A Faster FFT Implementation

Joymallya Chakraborty
North Carolina State University
jchakra@ncsu.edu

ABSTRACT

Fast and frugal tree is a binary classification tree. Chen D. et al. [2] have shown that FFT is comprehensible and extremely beneficial for defect prediction. As a course project for CSC-791 I have decided to explore different approaches to build FFT. Among sequential, parallel, and combination of sequential and parallel approaches, I found out that the third approach is the fastest.

KEYWORDS

FFT, defect Prediction,Python Multiprocessing

1 INTRODUCTION

In artificial intelligence and in heuristic decision-making, a fast-and-frugal tree is a type of classification tree. Fast-and-frugal trees can be used as decision-making tools which operate as lexicographic classifiers, and, if required, associate an action (decision) to each class or category. The original fast-and-frugal trees were introduced in 2003 by Laura Martignon et al. [12]. FFT is simple both in execution and in construction. It constitutes a kind of simple heuristic. Chen D. et al. [2] found that FFTs are remarkably effective. Their models are very succinct (5 lines or less describing a binary decision tree). These succinct models outperform state-of-the-art defect prediction algorithms defined by Ghortra et al. at ICSE'15. Also, when training data is restricted to operational attributes (i.e., those attributes that are frequently changed by developers), FFTs perform much better than standard learners. They had two major conclusions. Firstly, there is much that software analytics community could learn from psychological science. Secondly, proponents of complex methods should always baseline those methods against simpler alternatives. For example, FFTs could be used as a standard baseline learner against which other software analytics tools are compared.

1.1 Advantages of FFT

- Easy to implement - FFT implementation is really easy.
- Comprehensible - FFT is easy to describe.
- Succinct - FFT is small. So, description is just 4-5 lines(depending upon the height)
- Ensemble method - If depth is d, then 2^d trees are created. Each tree is applied on the training data and best tree is chosen depending upon the score (dis2heaven). That best tree is then tested on the test data.
- FFTs match the structure of SE data

Chen D. et al. [2] explained the reason why FFTs work well. During training phase, FFT explores 2^d models, then selects the models whose exit policies achieves best performances. The exit policies selected by FFTs are like a trace of the reasoning jumping around the data. Software data is "lumpy"; i.e., it divides into a few separate regions, each with different properties. Further, the number and importance of the "lumps" is specific to the data set and the goal criteria. In such a "lumpy" space, a learning policy like FFT works well since it's exit policies let a learner discover how to best jump between the "lumps". So, basically FFTs match the structure of SE data, that is why it performs well. Finally, FFTs can make good predictions even on the basis of a small amount of noisy data because they are relatively robust against a statistical problem known as overfitting. overfitting occurs when an algorithm has systematically lower accuracy in predicting new, unseen data compared to fitting old, known data. In contrast to regression (particularly in its classic, non-regularized form), FFTs tend to be robust against overfitting, and have been found to predict data at levels comparable with regression [11, 16]. So, FFT has so many advantages. It has mainly two limitations -

1.2 Limitations of FFT

- FFTs restrict number of conditions per rule to only one comparison - At each level only one attribute can be selected.
- The total number of rules is set to a small number (often just d ∈ 3, 4, 5). So, if a dataset has a large number of attributes, only 4-5 attributes can be used while building FFT. Therefore most of the attributes will be unused.

The rest of this report is structured as follows: Section 2 provides a background on FFT classification. Section 3 explains how a fast-and-frugal tree works, section 4 describes the evaluation measures for FFT. In section 5, FFT construction algorithms are vividly described. In section 6, the results are provided along with dataset description. Section 8 describes the future work and finally in section 9, conclusion and learning outcomes are given.

2 BACKGROUND & RELATED WORK

The fast-and-frugal decision tree was first introduced in 2003 by Laura Martignon, et al. [12]. In contrast to compensatory decision algorithms such as regression, FFTs allow people to make fast, accurate decisions based on limited information without requiring statistical training or a calculation device. Figure 1 (Source - Wikipedia) illustrates a fast-and-frugal tree for classifying a patient as "high risk" of having a heart stroke and thus having to be sent to the "coronary care unit" or "low risk" and thus having to be sent to a "regular nursing bed"(Green & Mehr, 1997).FFTs have been successfully used to both describe decision processes and to provide prescriptive guides for effective real-world decision making in a variety of domains, including medical [6, 7, 14, 15] legal

Foundations of Software Science, Fall 2018, Computer Science

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$9999.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

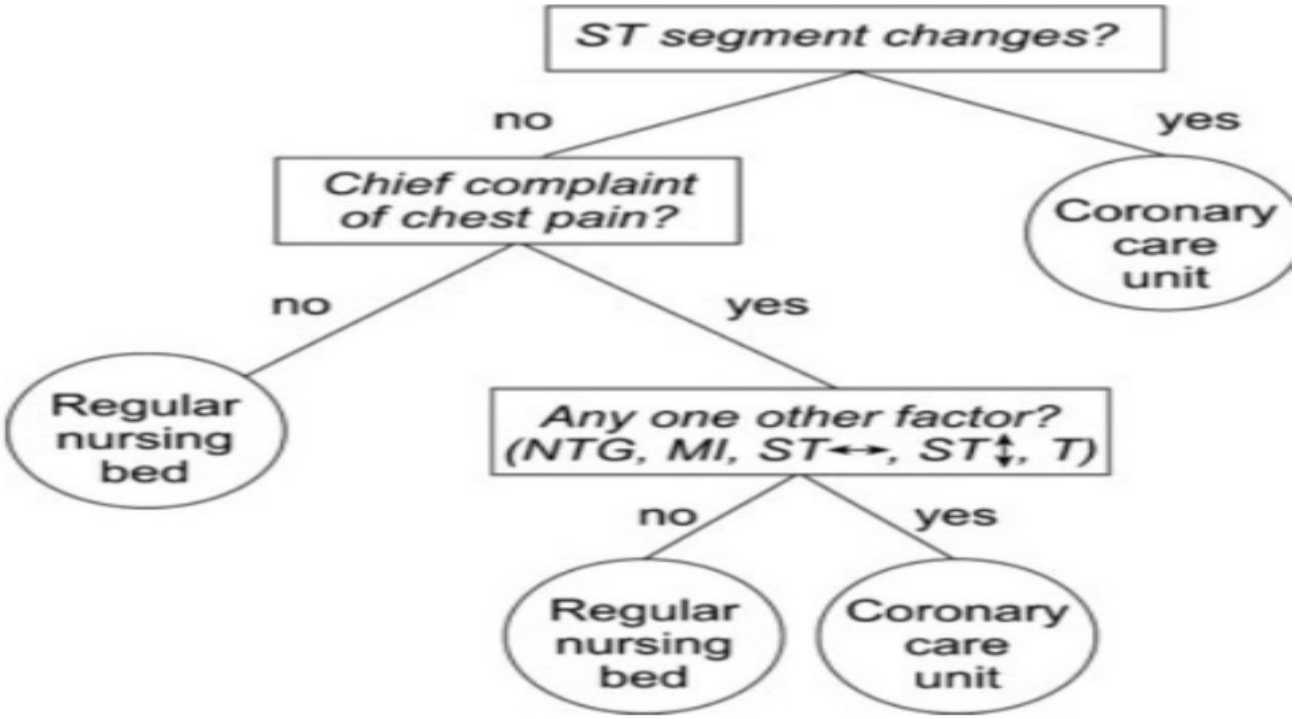

Figure 1: An example of Fast and Frugal Tree

[3–5], financial [1, 17] and managerial [9] decision making. Despite their proven effectiveness, FFTs were not used as often as other decision algorithms in earlier days. In 2017, Phillips, Neth, Woike and Gaissmaier [13] introduced the R package for FFTrees. They did a simulation study across ten real-world data sets. They found out that FFTrees can predict data as well as more complex classification algorithms, while remaining simple enough for anyone to understand and use. Figure 2. presents an FFT designed to classify patients as being at high or at low-risk for having heart disease [13]. The three nodes in the FFT correspond to the results of three medical tests: thal indicates the result of a thallium scintigraphy, a nuclear imaging test that shows how well blood flows into the heart while exercising or at rest. The result of the test can either be normal (n), indicate a fixed defect (fd), or a reversible defect (rd). The second node uses the cue cp, indicating a patient's type of chest pain, which can be either typical angina (ta), atypical angina (aa), non-anginal pain (np), or asymptomatic (a). Finally, ca indicates the number of major vessels colored by fluoroscopy, a continuous x-ray imaging tool, whose values can range from 0 to 3. To classify a patient with the FFT, begin with the first node (the parent node): If a patient's thal value is either rd or fd, then immediately classify the patient as high-risk, ignoring all other information about the patient. Otherwise, consider the next node: If a patient's cp value is aa, np, or ta, then immediately classify the patient as low-risk. Otherwise, consider the third and final node: If the ca value is positive, classify the patient as high-risk, otherwise classify the patient as low-risk. After the R package comes out, that FFT gained more popularity in the field of classification.

Fast-and-frugal trees have been characterized mathematically as lexicographic classifiers and as linear classifiers with non-compensatory weights [11]. Their "topology" and construction have been analyzed using signal detection theory [10] and their performance and robustness when compared to regression and CARTs has been studied by Laskey and Martignon (2014) [8]. An extensive study on the robustness, the predictive value and sensitivity/specificity of Fast-And-Frugal trees compared to those of Naive Bayes and of Full Natural Frequency Trees has been carried out by Woike J. et al. [16].

3 HOW A FAST-AND-FRUGAL TREE WORKS

The basic elements on which to ground a binary classification are (sets of) cues. The fast-and-frugal tree establishes a ranking and, according to the ranking, a "topology" of the tree. Once the ranking is established, the fast-and-frugal tree checks one cue at a time, and at each step, one of the possible outcomes of the considered cue is an exit node which allows for a decision.

Fast-and-frugal trees can also be described in terms of their building blocks. First, they have a search rule: They inspect cues in a specific order. Second, they have a stopping rule: Each cue has one value that leads to an exit node and hence to a classification, and another value that leads to consulting the next cue in the cue hierarchy (the exception being the last cue in the hierarchy, which has two exit nodes). Finally, they have a classification rule.

The core algorithm is very simple:

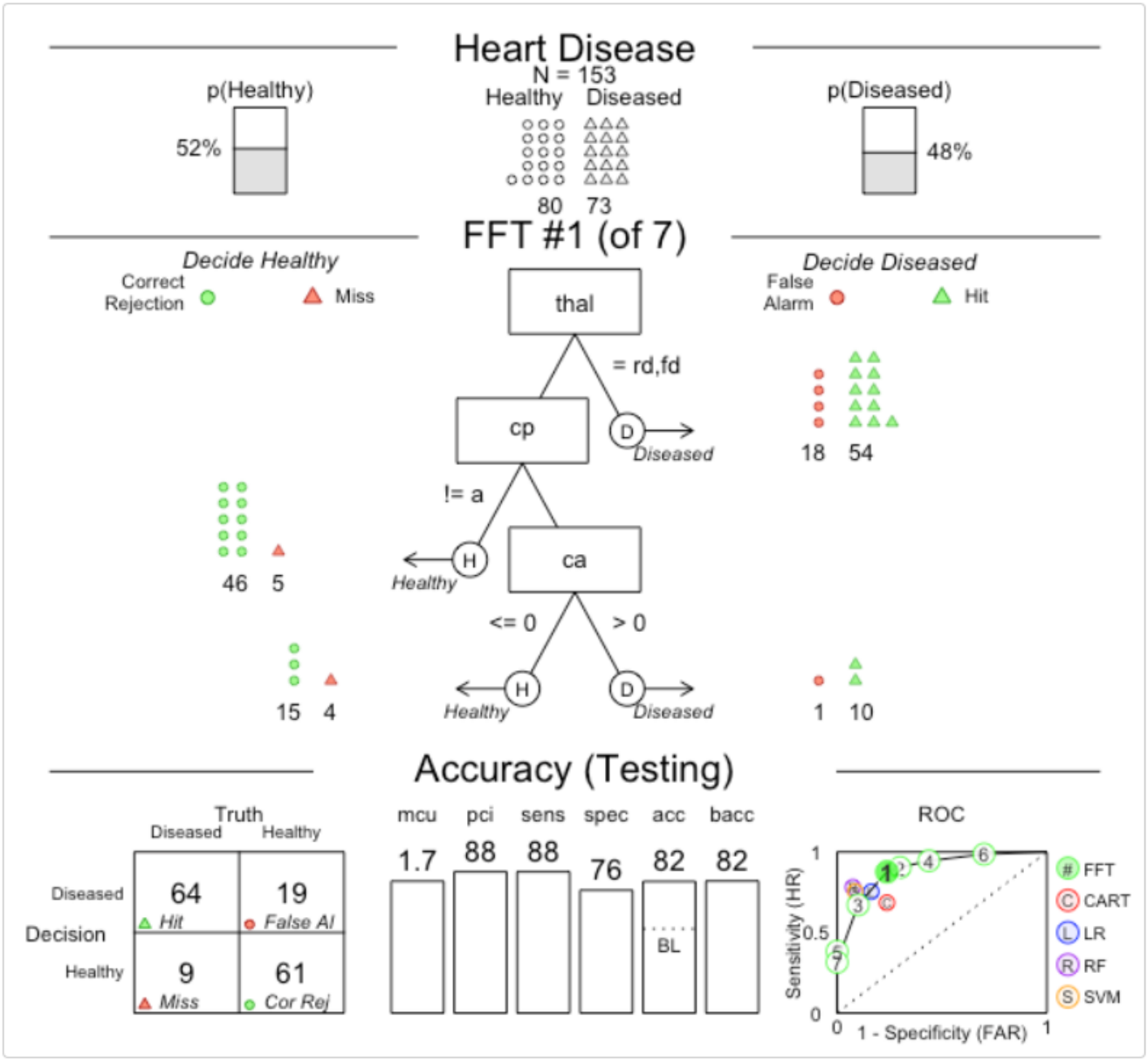

Figure 2: A fast-and-frugal tree (FFT) to predict heart disease status [13]

- First discretize all attributes; e.g., split numerics on median value.
- For each discretized range, find what rows it selects in the training data. Using those rows, score each range using some user-supplied score function e.g., recall, precision, false alarm.
- Divide the data on the best range.
- If the exit policy at this level is (0,1), then exit to (false,true) using the range that scores highest assuming that the target class is (false,true), respectively.
- If the current level is at d, add one last exit node predicting the opposite to step 4. Then terminate.
- Else, take the data selected by the non-exit range and go to step1 to build the next level of the tree.

4 EVALUATION MEASURES

A FFT is a decision tree with exactly two branches extending from each node, where either one or both branches is an exit branch leading to a leaf. For trees of depth d = 4, there are $2^4 = 16$ possible trees which are denoted as 00001, 00010, 00101,... , 11110. Here, the first four digits denote the 16 exit policies and the last digit denotes

the last line of the model (which makes the opposite conclusion to the line above).
For evaluation of FFTs, score function is called three times:

- Once to rank discretized ranges.
- Then once again to select the best FFT out of the 2^d trees generated during training.
- Then finally, score is used to score what happens when that best FFT is applied to the test data.

Chen D. et al [2] calculated **dis2heaven** and **Popt** using two score functions.

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

$$FAR = \frac{TruePositive}{TruePositive + FalseNegative}$$

A perfect learner will have Recall = 100% and no false alarms. These two are combined into a "distance to heaven" measure called dis2heaven that reports how far a learner falls away from the ideal point(heaven) of Recall=1 and FAR=0.

$$dis2heaven = \sqrt{\frac{(1 - Recall)^2 + FAR^2}{2}}$$

- For dis2heaven, the lower values are better
- For Popt , the higher values are better

5 FFT CONSTRUCTION

Here I will describe the procedure of building FFT. If depth of each tree is d, then total 2^d trees will be generated. Here, I took d = 4, so 2^4 = 16 trees will be generated. The algorithm of generating trees is already mentioned in the second section. Here, I will describe three different ways of implementing the algorithm.

5.1 Sequential Approach

The sequential approach is building trees one by one. Sequence of trees will be - 00001,00010,00101,.....,11110. Here, the first four digits denote the 16 exit policies and the last digit denotes the last line of the model (which makes the opposite conclusion to the line above). But some tricks can be used as well -

- **Grow the i'th tree (i is odd) and use it for (i+1)' th tree** - We can divide 16 trees into 8 consecutive pairs. For each pair, the first three levels are same. So, we can store the results for first three levels while building the first tree and reuse that result in building the second tree.
- **Store median chops** - While building each tree, we are doing supervised discretization. In case of numeric columns, we are computing the median. For each tree, while building the first level, we are computing the median of all the numeric columns. While building the first tree if we can store the median results, we can reuse those while building subsequent trees. We can not store medians for each level, because in each level we are dealing with a different set of data.
- **Use subroutines** - We can use subroutines to reduce recoding.
- **Calculate tree score , if not good , jump (GREEDY)** - There are 16 trees. We can divide these 16 trees into 4 groups. Trees starting with 00(Trees 1 - 4),01(Trees 5 - 8),10(Trees 9 - 12) and 11(Trees 13 - 16). We will start building from tree 1 (0000). After building, we will compute the score of the tree. If score is not good, we will jump 4 steps and start building tree 5. If tree 5 is not of good score, we will again jump 4 steps and start building tree no. 13. This is extremely greedy approach, so the chances of missing the best tree is also there. I did not get enough time to implement this approach in this course project.

5.2 Parallel Approach

Sixteen trees are independent. So, those trees can be built in parallel. Python multiprocessing is not very easy, because of GIL. The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter. This means that only one thread can be in a state of execution at any point in time. The impact of the GIL is not visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code. Since the GIL allows only one thread to execute at a time even in a multi-threaded architecture with more than one CPU core, the GIL has gained a reputation as an "infamous" feature of Python. Still, there are two ways to do parallel programming in python. Python community claim that these two methods can overcome the performance bottleneck caused by GIL.

- **Python joblib** - joblib.Parallel uses the 'loky' backend module to start separate Python worker processes to execute tasks concurrently on separate CPUs. This is a reasonable default for generic Python programs but can induce a significant overhead as the input and output data need to be serialized in a queue for communication with the worker processes.
  I tried to implement FFT using joblib.Parallel. But instead of being faster, I saw the program got slower. There are two main reasons for that.
  - **Parallel pool creation overhead:**The problem here is that creating a parallel pool is costly. Each time a new pool is created, it takes some time which cumulatively affect the total execution time of the program.
  - **Dispatching overhead:** it is important to keep in mind that dispatching an item of the for loop has an overhead (much bigger than iterating a for loop without parallel). Thus, if the individual computation items are very fast, this overhead will dominate the computation.
- **Python Multiprocessing** - multiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

I successfully used multiprocessing approach. I divided sixteen trees into four groups and created four sub-processes. The sub-processes were pinned to four different cores.

5.3 Combination of sequential and Parallel

In my implementation, I combined the sequential and parallel approach.

- **Apply the odd, even approach and store median chops (sequential)** - I paired up sixteen trees into eight groups (as mentioned above) and while building the first tree I saved the median chops of numeric attributes. In the process of building subsequent trees, those values were used.
- **Use multiprocessing (parallel)** - I divided sixteen trees into 4 groups. The main tree building process was divided into four sub-processes. Four sub-processes were executed on different cores.

6 DATASET DESCRIPTION & RESULTS

I used defect prediction dataset to check execution time of my FFT implementation. The dataset consists four public GitHub projects - abinit,libmesh,lammps and mdanalysis. The dataset has following features :-

- All the attributes are CK (Chidamber and Kemerer) metrics. CK metrics measure unique aspects of the object oriented approach. It is used to measure the complexity of the design.
- Class (Buggy/Not buggy) - Whether file is buggy or not
- Time dependent dataset (No K-fold) - The data is in chronological order. So, K-fold cross validation could not be used. For all four projects, I used first 80% data for training purpose and rest 20% data for testing purpose. Table 1. describes the spread of class labels for these four projects.

Table 1: Spread of Class labels

Class	abinit	libmesh	lammps	mdanalysis
0 (Not Buggy)	4024	6462	7437	2802
1 (Buggy)	572	651	186	379

In my experiment I have taken sequential approach as my baseline and then improved it using above mentioned tricks. I have run both versions of FFT for the four datasets - abinit,lemmps,libmesh and mdananlysis. Figure 3. shows the results for the baseline approach and my implementation. My implementation achieved 7-10% improvement on average. As I have used multiprocessing in my approach, it makes execution time worse in case where individual tasks are not large. For abinit dataset, sequential approach takes longest execution time. For that dataset my implementation shows best improvement. For other three datasets, the improvement is not significant enough.

7 DISCUSSION

Figure 4. shows the function call sequence of FFT building algorithm. build_trees() is the main function which is responsible for building sixteen trees. Inside build_trees(), growEven() and growOdd() functions are called depending upon the index of the tree. Inside growEven() function, tree is built from scratch and first three levels are stored. Inside growOdd() function, tree is built from level three. growEven() and growOdd() both functions internally call call_eval_point_split(). call_eval_point_split() calls eval_point_split() which calls eval_decision(). eval_decision() is the function which calculates the performance of the decision or evaluates how good the decision is. eval_decision() does not call any other function. Table 2. shows the profiling result for abinit dataset. The description of each coulmn is given here -

- ncalls - for the number of calls
- tottime- for the total time spent in the given function (and excluding time made in calls to sub-functions)
- percall - is the quotient of tottime divided by ncalls
- cumtime - is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions
- percall - is the quotient of cumtime divided by primitive calls
- filename:lineno(function) - provides the respective data of each function

After drilling down , it is seen that eval_decision() is taking a lot of time. It is not calling any other function, so it should be written using a different approach to reduce the execution time.

8 LIMITATION & FUTURE WORK

I have taken sequential approach as my baseline and tried to improve the execution time of building FFT trees using some tricks. I have also applied multiprocessing to divide the task among different cores. Multiprocessing makes program execution time worse if individual tasks are not long enough. So, my implementation will perform worse than baseline in case of smaller datasets. I may have to modify my code a bit to make further tuning. Here I am summarizing the tasks I want to do in future to further extend/improve this work -

- Try with smaller datasets to find out the adverse effect of multiprocessing.
- Try with datasets which are not related to defect prediction.
- Implement FFT algorithm using Python set union and intersection.
- Implement the greedy approach which I mentioned in section 5.
- Implement eval_decision() function using different approach

9 CONCLUSION

In this course project, I have taken sequential FFT as baseline and tried to make it faster. I used some special tricks in sequential approach and python multiprocessing technique. The improvement is not significant. Still I think this project was extremely beneficial for my career. The learning outcomes of this project are -

- Python Coding
- Python multiprocessing
- Python profiling
- Familiarity with ARC
- Familiarity with Classification Tree

In future I will try to dive deep in the field of data science research with the knowledge I acquired from this class project.

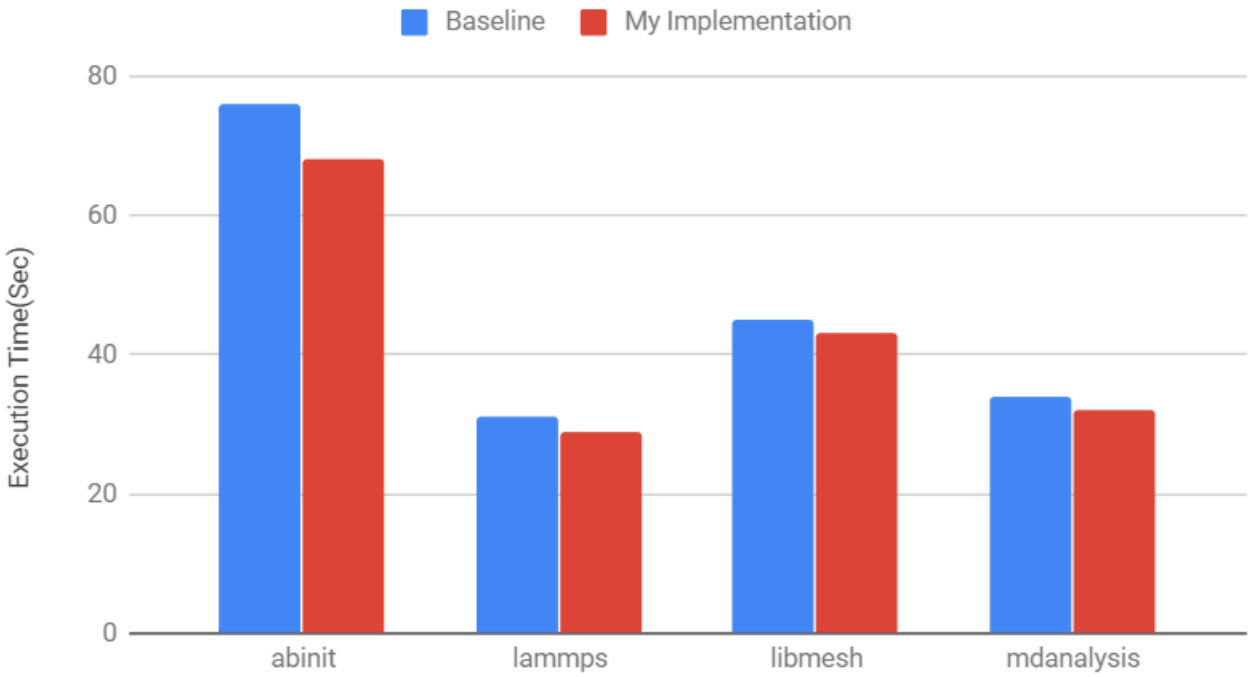

Figure 3: Results showing execution time for Baseline and my approach

Table 2: Python Code Profiling

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
6992	0.342	0.000	57.550	0.008	FFT.py:166(eval_decision)
6720	0.055	0.000	60.723	0.009	FFT.py:195(eval_point_split)
120	0.037	0.000	65.397	0.544	FFT.py:216(call_eval_point_split)
160/32	0.001	0.000	54.307	1.700	FFT.py:233(growEven)
64/32	0.000	0.000	12.201	0.381	FFT.py:258(growOdd)
4	0.000	0.000	66.608	16.652	FFT.py:39(build_trees)

REFERENCES

[1] David Aikman, Mirta Galesic, Gerd Gigerenzer, Sujit Kapadia, Konstantinos Katsikopoulos, Amit Kothiyal, Emma Murphy, and Tobias Neumann. 2014. Taking Uncertainty Seriously: Simplicity versus Complexity in Financial Regulation. *Bank of England Financial Stability Paper* 28 (05 2014). https://doi.org/10.2139/ssrn.2432137

[2] Di Chen, Wei Fu, Rahul Krishna, and Tim Menzies. 2018. Applications of psychological science for actionable analytics. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018* (2018). https://doi.org/10.1145/3236024.3236050

[3] Mandeep Dhami and Peter Ayton. 2001. Bailing and jailing the fast and frugal way. *Journal of Behavioral Decision Making* 14 (04 2001), 141–168. https://doi.org/10.1002/bdm.371.abs

[4] Mandeep K. Dhami. 2003. Psychological Models of Professional Decision Making. *Psychological Science* 14, 2 (2003), 175–180. https://doi.org/10.1111/1467-9280.01438 arXiv:https://doi.org/10.1111/1467-9280.01438 PMID: 12661681.

[5] Mandeep K. Dhami and Clare Harries. 2001. Fast and frugal versus regression models of human judgement. *Thinking & Reasoning* 7, 1 (2001), 5–27. https://doi.org/10.1080/13546780042000019 arXiv:https://doi.org/10.1080/13546780042000019

[6] Fischer JE, Steiner F, Zucol F, and et al. 2002. Use of simple heuristics to target macrolide prescription in children with community-acquired pneumonia. *Archives of Pediatrics Adolescent Medicine* 156, 10 (2002), 1005–1008. https://doi.org/10.1001/archpedi.156.10.1005 arXiv:/data/journals/peds/5052/poa20037.pdf

[7] Mirjam A. Jenny, Thorsten Pachur, S. Lloyd Williams, Eni Becker, and JÃijrgen Margraf. 2013. Simple rules for detecting depression. *Journal of Applied Research in Memory and Cognition* 2, 3 (2013), 149 – 157. https://doi.org/10.1016/j.jarmac.2013.06.001

[8] Kathryn Laskey and Laura Martignon. 2014. Comparing Fast and Frugal Trees and Bayesian Networks for Risk Assessment. (07 2014).

[9] Shenghua Luan and Jochen Reb. 2017. Fast-and-frugal trees as noncompensatory models of performance-based personnel decisions. *Organizational Behavior and Human Decision Processes* 141 (07 2017), 29–42. https://doi.org/10.1016/j.obhdp.2017.05.003

[10] Shenghua Luan, Lael J. Schooler, and Gerd Gigerenzer. 2011. A signal-detection analysis of fast-and-frugal trees. *Psychological review* 118 2 (2011), 316–38.

[11] Laura Martignon, Konstantinos Katsikopoulos, and Jan Woike. 2008. Categorization with Limited Resources: A Family of Simple Heuristics. *Journal of Mathematical Psychology* 52 (12 2008), 352–361. https://doi.org/10.1016/j.jmp.2008.04.003

[12] Laura Martignon, Oliver Vitouch, Masanori Takezawa, and Malcolm Forster. 2003. *Naive and Yet Enlightened: From Natural Frequencies to Fast and Frugal Decision Trees.* 189–211 pages. https://doi.org/10.1002/047001332X.ch10

[13] Nathaniel D. Phillips, HansjÃČÂűrg Neth, Jan K. Woike, and Wolfgang Gaissmaier. 2017. FFTrees: A toolbox to create, visualize, and evaluate fast-and-frugal decision trees. *Judgment and Decision Making* 12, 4 (2017), 344–368. https://EconPapers.repec.org/RePEc:jdm:journl:v:12:y:2017:i:4:p:344-368

[14] Nate Silver. 2012. The signal and the noise: why so many predictions fail–but some don't. *New York: Penguin Press* (2012).

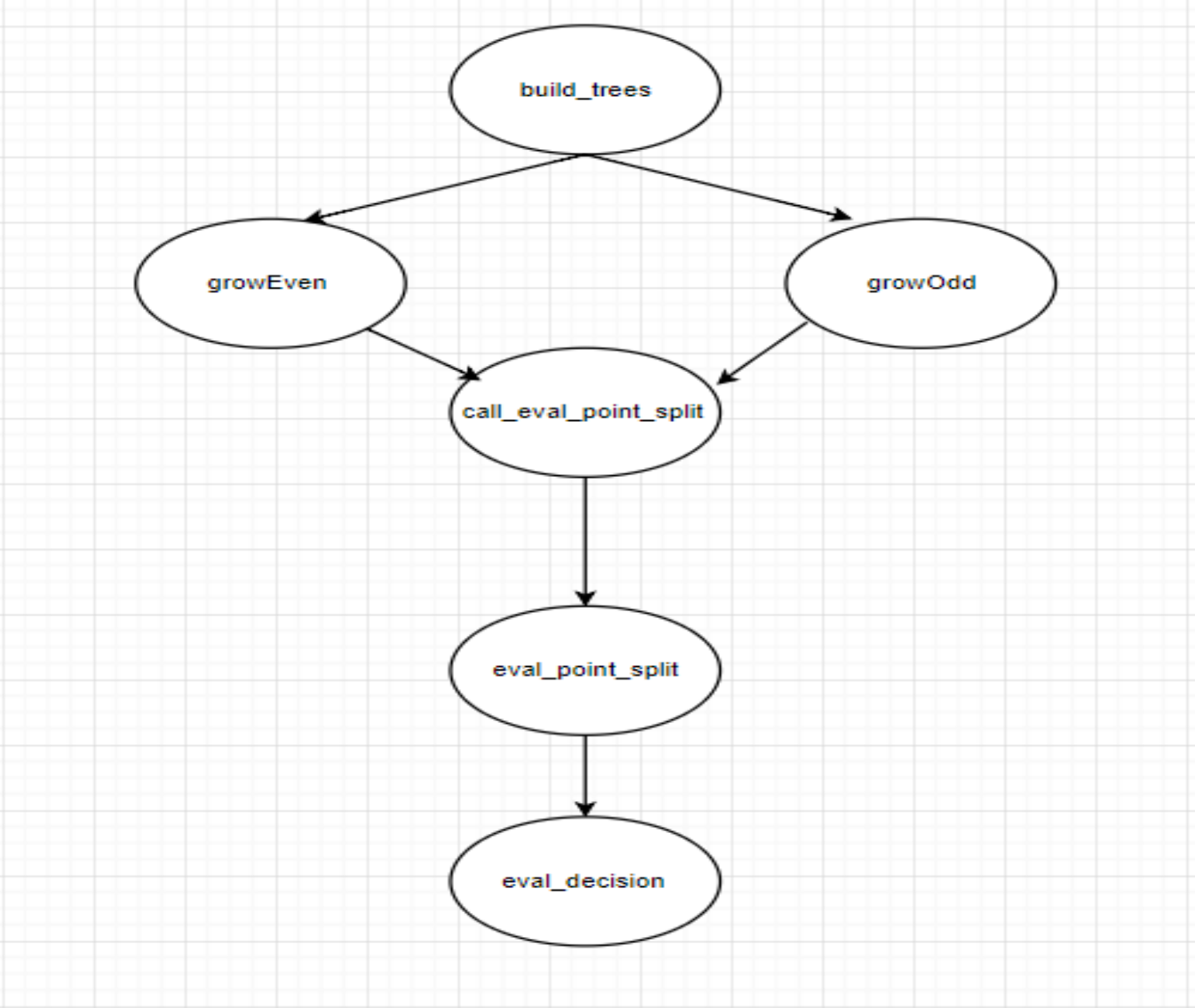

Figure 4: Python Code Function Call Flow

[15] Odette Wegwarth, Wolfgang Gaissmaier, and Gerd Gigerenzer. 2009. Smart strategies for doctors and doctors-in-training: heuristics in medicine. *Medical education* 43 8 (2009), 721–8.

[16] Jan Woike, Ulrich Hoffrage, and Laura Martignon. 2017. Integrating and Testing Natural Frequencies, NaÃŕve Bayes, and Fast-and-Frugal Trees. *Decision* 4 (06 2017). https://doi.org/10.1037/dec0000086

[17] Jan Woike, Ulrich Hoffrage, and Jeffrey Petty. 2015. Picking profitable investments: The success of equal weighting in simulated venture capitalist decision making. *Journal of Business Research* 68 (04 2015). https://doi.org/10.1016/j.jbusres.2015.03.030