Fast Frugal Trees vs. Random Forests: Repeatability in defect prediction

Aswin Anil Kumar
North Carolina State University
aanilku@ncsu.edu

Samim Mirhosseini
North Carolina State University
smirhos@ncsu.edu

Sreeram Veluthakkal
North Carolina State University
sveluth@ncsu.edu

ABSTRACT

We compare and report few key performance metrics that were missing in prior studies that compared Fast Frugal Trees (FFT) and Random Forests (among other learning models). This report aims to compare 'repeatability' and 'readability' of these popular learners in terms of computational cost (CPU, Memory footprint) and model size respectively in defect prediction of software projects.

For three datasets, across multiple iterations of various training and testing sample sizes, it was observed that

(1) FFTs was upto 3x faster than Random forests and creates a model of 6x lesser size.

(2) Random Forest models perform slightly better with respect to accuracy in prediction without any tuning.

KEYWORDS

Fast Frugal Trees, Random Forests, Readability, Repeatability, SMOTE, P-Opt

1 INTRODUCTION

Defect prediction usually trains on large amounts of multi-dimensional data as a result of the size of enterprise size code bases, their production pace and myriad of defect contributing factors like team size to lines of code or number of classes. Often the learner has to be run multiple times for each project due to any number of reasons like addition of features in data or significant changes to code base or even simply to test the accuracy and reproduce consistent results. This is what is referred to as repeatability of a model. Many papers compare the learners for application in this domain and look only at performance metrics like precision/recall.

Phillips et. al in their paper introduce a package for FFTs and compare its efficiency and prediction accuracy among others against other well-known models. The paper states 'low cost' of FFTs, but that and repeatability is not discussed, which is a key criterion, and is the focus here in our study.

The repeatability of any algorithm is directly tied to the speed in which it can produce an outcome and in the case of FFTs, Phillips et. al in their paper use two distinct measures for this: *mcu* and *pci*. MCU is mean number of cues used in arriving at a decision i.e. the number of nodes traversed before we arrive at the final decision. PCI is the percentage of cues or columns ignored.

As seen, fastness and frugality is being measured by the cues required and cues ignored respectively and not by the computational resources taken. Computational limitations are usually in terms of processing power and memory availability and their performance. While cues are a good measure, it may not necessarily translate into faster processing times. Testing stability of a conclusion via 'repeatable' runs is impossible with long running tasks. One problem is that it is looking only at one face of the coin. Does lesser number of cues essential translate to lesser computing resource requirements or faster learning (execution) times? In today's world of 'Internet of Things' where learning happens everywhere from coffee makers to automated cars, computing resources may be constrained and considering these factors are also extremely important in choosing a learner over another. It is also impossible for other researchers to test the original work. For instance, learning tuning parameters of software cloning detectors as proposed in [11] would take 15 years of CPU time, and it is impossible, economically as well, to repeat it. Thus, here the comparison between the two learning methods will be on 'task run time and other metrics' and baseline the computational requirements in a controlled and similar experimental environment across all runs.

Thus, research of this paper began with the question *which of these two learners are more repeatable and/or readable? What is the tradeoff in looking at just the accuracy of the learners for comparison?*

RQ1: If enough data is not available in the dataset to accurately study repeatability with growing problem size, how can data be synthetically created based on scientific methodologies while preserving the statistical properties?

Result 1

Modified SMOTE [13] to oversample all classes and not just minority class until the required sample size is achieved without creating a population that is significantly different.

RQ2: Is Fast Frugal Trees really more repeatable comparing to random forests?

Result 2

Fast frugal trees performed much faster on the data samples compared to random forests while also creating models that have a smaller memory footprint.

RQ3: If FFTs are faster than Random Forests, is there a performance trade off involved?

Result 3

With the default parameters, Random Forest have a slight advantage but parameter tuning and other optimizations may give different conclusions and is left for future exploration.

2 BACKGROUND AND MOTIVATION

2.1 Model readability

Model readability is useful especially for defending an output against the business user. Using a learner that makes a more readable model is preferred for showing a new idea, although it may not perform the best in some other criteria (No Free Lunch). Model readability is hard to achieve because as only certain learners like decision tree are readable by nature but even those become complex as the size and features of data increases in real world application. Another advantage of model readability is that it opens up our model for critique by the community, thus rapidly improving the model itself [12].

2.2 Learnability and repeatability of the results

It is important for a learner to use as less amount of RAM, disk and CPU time as possible. In real world applications, we often need to improve and reproduce results to prove accuracy or build improved versions of the learner. Thus, a fast and light learner is useful. This criterion is hard to achieve as faster or lighter learners usually accomplish this by ignoring sets of data or batching. These

Table 1: OO CK code metrics used for all studies in this paper. The last line shown, denotes the dependent variable.

amc	average method complexity	e.g., number of JAVA byte codes
avg, cc	average McCabe	average McCabe's cyclomatic complexity seen in class
ca	afferent couplings	how many other classes use the specific class.
cam	cohesion amongst classes	summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods.
cbm	coupling between methods	total number of new/redefined methods to which all the inherited methods are coupled
cbo	coupling between objects	increased when the methods of one class access services of another.
ce	efferent couplings	how many other classes is used by the specific class.
dam	data access	ratio of the number of private (protected) attributes to the total number of attributes
dit	depth of inheritance tree	
ic	inheritance coupling	number of parent classes to which a given class is coupled
lcom	lack of cohesion in methods	number of pairs of methods that do not share a reference to an case variable.
locm3	another lack of cohesion measure	if m, a are the number of $methods, attributes$ in a class number and μa is the number of methods accessing an attribute, then $lcom3 = \frac{1}{a}, j^a \mu a, j - m1 - m.$
loc	lines of code	
max, cc	maximum McCabe	maximum McCabe's cyclomatic complexity seen in class
mfa	functional abstraction	no. of methods inherited by a class plus no. of methods accessible by member methods of the class
moa	aggregation	count of the number of data declarations (class fields) whose types are user defined classes
noc	number of children	
npm	number of public methods	
rfc	response for a class	number of methods invoked in response to a message to the object.
wmc	weighted methods per class	
nDefects	raw defect counts	numeric: number of defects found in post-release bug-tracking systems.
defects present?	boolean	if $nDefects > 0$ then *true* else *false*

Source: Amritanshu Agrawal and Tim Menzies. 2018. Is "Better Data" Better Than "Better Data Miners"?

Dataset Name	Original Dataset			Mutated Dataset		
	Defect %	Non Defect %	Size	Defect %	Non Defect %	Size
Velocity	34	66	640	37	73	8912
Synapse	33.5	66.5	636	36.8	74.2	8493
Tomcat	9	91	859	11.5	89.5	8226

Table 2: Data set statistics. Data sets are sorted from high percentage of defective class to low defective class. Data comes from the SEACRAFT repository: http://tiny.cc/seacraft

methodologies are often not considered to create accurate results [5] and it is harder to prove and advocate its usage over a standard, well used, complex learner. This criterion is the main focus of this essay, and we will discuss in more details in key criteria section.

2.3 Defect Prediction

Bugs or defects in code are very common and software testing methodologies aim at maximum coverage as opposed to complete coverage. It is also an expensive process done under extreme pressure to ensure on time production release. Previous studies [14] have shown that assessment effectiveness increases exponentially with assessment effort [7]. Prediction models based on the topological properties of components within them are also seen to be accurate [18]. Thus, future defect locations can be guessed using past defects logs [16, 17]. These logs might be summarized by software components using some metrics like the CK metrics [19] (Table 1 [14]).

It has been found that such static code defect predictors are fast and effective [14] and that no significant differences exist in the cost effectiveness of static code analysis tools and static code defect predictors. [14,15]

2.4 Data mutation with SMOTE

Class imbalance in a data set is when some classes in a data set is under-represented in comparison to other classes. [20] The under-represented classes are called *minority* classes and over-represented classes are called *majority* classes. Synthetic minority over-sampling technique (SMOTE) handles class imbalance by changing the frequency of different classes of training data. [13]. Figure 1 shows how SMOTE works. The majority classes are sub samples by deleting few data samples while in super sampling, a data point in the minority class looks at it's k nearest neighbors and builds a fake member of this class between the itself and one of it's nearest neighbors. The distance function used is the minkowski distance function. The control parameters of SMOTE are 'k' that selects the nearest neighbors to use, 'm' the number of examples to generate and 'r', the distance function.

The description so far has been about the classic version of SMOTE [13] and there have been various versions that followed from various researches (like [14]). In this research, we are not trying to handle the class imbalance with SMOTE. Most of data sets used in this study have a good balance between the two classes as shown in Table 2. SMOTE is now modified to do Synthetic over-sampling of all classes and not just the minority classes. The algorithm remains the same where the data points look at k nearest neighbors of *that class* and builds a *mutated* member, but in this case, it is done for all classes until the sample size reaches 'm' i.e. instead of the while in the definition of SMOTE where Minority < m condition is checked, the add something_like step is carried out for the entire data set. The distribution of the original data should be preserved

```
def SMOTE(k=2, m=50%, r=2): # defaults
  while Majority > m do
    delete any majority item
  while Minority < m do
    add something_like(any minority item)

def something_like(X0):
  relevant = emptySet
  k1 = 0
  while(k1++ < 20 and size(found) < k) {
      all = k1 nearest neighbors
      relevant += items in "all" of X0 class}
  Z = any of found
  Y = interpolate (X0, Z)
  return Y

def minkowski_distance(a,b,r):
  return Σ_i absa_i − b_i^r 1r
```

Figure 1: Pseudocode of SMOTE
Source: Amritanshu Agrawal and Tim Menzies. 2018. Is "Better Data" Better Than "Better Data Miners"?

```
rank ,                          name ,    med   ,  iqr
-----------------------------------------------------
   1 ,                           wmc ,    5   ,    6 (*--           |             ), 2.00,  4.00,  5.00,  7.00, 20.00
   1 ,                          wmc_m ,    5   ,    5 (*--           |             ), 2.00,  3.00,  5.00,  6.00, 19.00

rank ,                          name ,    med   ,  iqr
-----------------------------------------------------
   1 ,                           cbo ,    7   ,    7 (- *-----      |             ), 2.00,  5.00,  7.00, 10.00, 22.00
   1 ,                          cbo_m ,    8   ,    7 (- * ----      |             ), 2.00,  5.00,  8.00, 11.00, 23.05

rank ,                          name ,    med   ,  iqr
-----------------------------------------------------
   1 ,                           rfc ,    16   ,   23 ( * ---        |              ), 3.00,  8.00, 16.00, 26.00, 51.00
   1 ,                          rfc_m ,    17   ,   22 (- *--        |              ), 3.00,  9.00, 17.00, 25.00, 50.00

rank ,                          name ,    med   ,  iqr
-----------------------------------------------------
   1 ,                           loc ,    86   ,  195 (*             |              ), 5.00, 31.00, 86.00, 184.00, 417.00
   1 ,                          loc_m ,    88   ,  190 (*             |              ), 5.00, 34.00, 90.00, 178.00, 437.00

rank ,                          name ,    med   ,  iqr
-----------------------------------------------------
   1 ,                           amc ,    12   ,   25 ( *---         |              ), 0.00,  5.00, 12.50, 26.00, 49.17
   1 ,                          amc_m ,    15   ,   25 ( * --         |              ), 0.00,  5.53, 15.50, 28.00, 49.80
```

Figure 2: Mutated Data set statistics.
NOTE: Only the top features which show differences in one or more IQRs from the data set Velocity are shown.

so that the learning is accurate. Statistical measures like inter-quartile range analysis to study statistical significance of the difference between the two data sets (the original and the mutated) are to be applied. Note that test set data is not oversampled using SMOTE.

2.5 Computing Requirements

In this paper, we mainly focus on repeatability and performance criteria. Repeatability in learners is important because as stated by Fu et al. [6] "[…] long training time limits the ability of (a) researchers to test the stability of their conclusion via repeated runs with different random seeds; and (b) other researchers to repeat, improve, or even refute that original work.

Repeatability Criteria:

To measure repeatability of random forests and fast frugal trees, we considered three measures 1) Process memory 2) Model size 3) CPU time. Below we explain each of these measures and their significance:

Process memory:

Process memory is the peak memory usage of our scripts that makes the models and does prediction. Process memory is measured in Kilobytes. We measure process memory to be able to compare the implementations of the packages and compare how efficient they are. But we also want to measure the model size which is described below.

Model size:

Model size is the amount of space the models created by each learner takes on the memory. Model size can give us a better idea about how the two methods compare while making the implementation of each package have smaller effect on the results. Model size is measured in Bytes.

CPU time:

CPU time is the amount of time that the learner takes to make the model and do the predictions. When researching the repeatability of a model, CPU time is one of the main things that we need to consider. CPU time is measured in seconds.

	Actual	
Prediction	false	true
defect-free	*TN*	*FN*
defective	*FP*	*TP*

Table 3: Confusion Matrix

2.6 Performance Criteria

Even though our key criteria are readability and repeatability, we cannot ignore the performance of our models. A really fast and highly readable model is of no use if does not deliver at least a reasonable level of performance.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

$$F1 = \frac{2 * Precision * Recall}{Recall + Precision}$$

The performance metrics we are using here are precision, recall and the F1 Score.

Precision is the ratio of actual bugs to all predicted bugs. High precision indicates low false positive rates. Recall is the ratio of predicted bugs to all actual bugs. The F1 score combines both precision and recall and is the Harmonic mean of Precision and Recall.

We are not considering accuracy because accuracy gives a good picture only when the dataset is symmetric and there is very little class imbalance. Accuracy can be misleading when there is a class imbalance.

But measuring these characteristics aren't enough. We should also take into account the effort involved in obtaining the results. This is called effort aware defect prediction. We measure the standard statistics at 20% of all the lines of code. Essentially, we will be measuring how much defects the learner is able to detect given 20% of the data. This cutoff is not a random or magic number, according to Ostrand et al. [22], in a software engineering project, 20% of lines of code contain on an average 80% of the defects.

In effort aware defect prediction, we compare our learner with a hypothetical optimal model. The optimal model is built by sorting the data in increasing order of lines of code and by having the records with bugs at the front. This represents model is optimal because the we are able to find more number of defects while going through less lines of code.

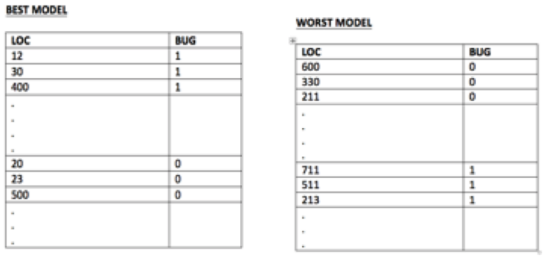

Figure 3: Data arrangement for best and worst-case models

The worst model is built by sorting the records in decreasing order of lines of code. Also, the records with bugs comes after all the records without bugs. This represents the worst-case model because, we have to run through a lot of the code in order to find a few bugs.

The closer our model is close to the optimal model, the better the model is in terms of effort taken to predict the bugs. The optimal model, worst model and our learner are compared in a chart called the effort based cumulative lift chart. Figure 4 shows an effort based cumulative lift chart (Wei Fu et al. [21])

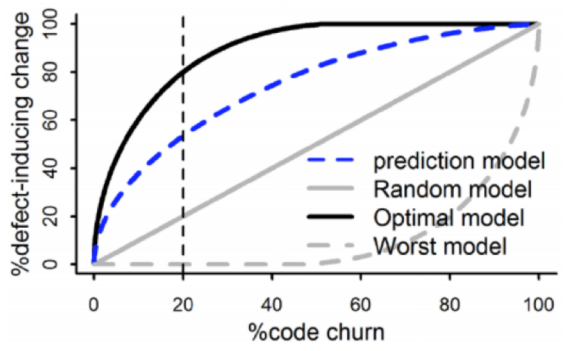

Figure 4: Effort based cumulative lift chart. [21]

One of the new parameters introduced in effort aware defect prediction is Popt which is defined as:

$$Popt = 1 - \Delta opt$$

Here, Δopt is the area between the charts for the optimal model and our model.

There is also a normalized version of Popt defined as:

$$P_{opt}(m) = 1 - \frac{S(optimal) - S(m)}{S(optimal) - S(worst)}$$

Where S(optimal), S(m) and S(worst) are the area under the curve for the optimal model, predicted model and the worst-case model.

3 EXPERIMENTAL DESIGN

This experiment reports on the repeatability and readability of FFTs and RF when used for defect prediction on the data sets shown in Table 2.

To compare repeatability criteria between random forest and fast frugal trees, first we create a model using each learner and then do a cross validation on each model. For this experiment, we used three datasets as shown in Table 2 from the Seacraft repository. To accomplish this, we wrote R scripts that use packages "randomForest", "FFTrees" to create random forest and Fast frugal trees respectively. "randomForest" has been a very well-known R package for many years, but "FFTree" was recently released by Philip in his paper [source]. Both of the packages allow some configuration before they make the model. In this project, we used 1000 trees and 25 nodes for random forest, and 1 tree with 4 levels for FFTrees.

As described in the previous sections we defined three measures which we will use to compare the repeatability of random forest and fast frugal trees. These measures are Process memory, Model size, and CPU time which we assess for each learner. The methods that we use to calculate these measures are described below.

Process memory: We used an open source utility called "memusg" [source] that measures the peak memory usage of any process. memusg is a very simple utility; as an input it gets a shell command and runs it, and at the same time it finds the peak memory usage of that command.

Model size: We created our models in R and we also used `object.size(model)` function of R to find how much space does the model take in the memory. Model size allows us to more precisely compare the actual models rather than comparing the R packages that create these models.

CPU time: We use the "time" utility of linux/unix to measure the time that the process for making random forest and fast frugal tree models.

To automate these measurements, we wrote shell scripts to run these tests and save a report file at the end. To make sure the results are consistent we ran these scripts 5 times and calculated average of the results; We also kept the computer hardware the same between all the runs.

To summarize, the experimental steps carried out were as follows:

(1) Randomized the data set order five times to reduce the probability that some random ordering of examples in the data will affect the results.

(2) Each time, did multiple iterations of training and testing for incremental data set sizes to study the variations in performance metrics. Note that the mutated data created by SMOTE was used only for training.

(3) Gathered the performance metrics to study repeatability and readability, and then looked at the prediction results to arrive at Popt values.

3.1 Statistical Analysis

When comparing the results of the mutation using the modified SMOTE, it needs to be tested that the mutated data is not statistically significant from the original so that the learning is accurate. Figure 2 shows the IQR comparison between the two for Velocity data set.

4 RESULTS

RQ1: If enough data is not available in the dataset to accurately study repeatability with growing problem size, how can data be synthetically created based on scientific methodologies while preserving the statistical properties?

Yes. As described above, using a slightly modified SMOTE, more data can be 'created' to create a significant size of data set that enable us to study the results with respect to repeatability and readability. As shown in Figure 2 the distribution of classes remains closely similar, and as seen in Table 3, the median and other inter quartile ranges across all features in the data set remain similar. It can be confidently stated that the mutation does not affect the performance of the learning.

RQ2: Are Fast Frugal Trees more repeatable compared to Random Forests?

As discussed above, repeatability is dependent on the memory/disk usage and the time it takes to run the learner. We ran our experiment on the three different datasets which have different distributions, and for each dataset, assessed our three measures of CPU time, model size and process memory that we defined previously. Figures 5 show the results for the all datasets (Velocity, Synapse and Tomcat).

For the CPU times (Figure 5), although for small sample sizes FFT used more memory, as we increase the sample size, the time that it takes for RF to make the model increased faster comparing to FFT and for sample size > 2000 it always took longer for RF to complete.

For model size (Figure 5), FFT always created models that are smaller in size comparing to RF. This is to

be expected because FFT creates one tree vs RF that creates 1000 trees. The model size can also be used to argue FFT is also more readable than RF.

The process memory (Figure 5) was surprisingly different from what we expected. The trend of process memory between both FFT and RF is similar but RF is using less memory. We predict this can be caused because of the implementations of the packages and think more investigation of the source code will be needed in future work.

RQ3: If FFTs are faster than Random Forests, is there a performance trade off involved?

To compare the performance of FFTs against Random Forests, we used the same datasets we used to compare readability and repeatability. The mechanism used to

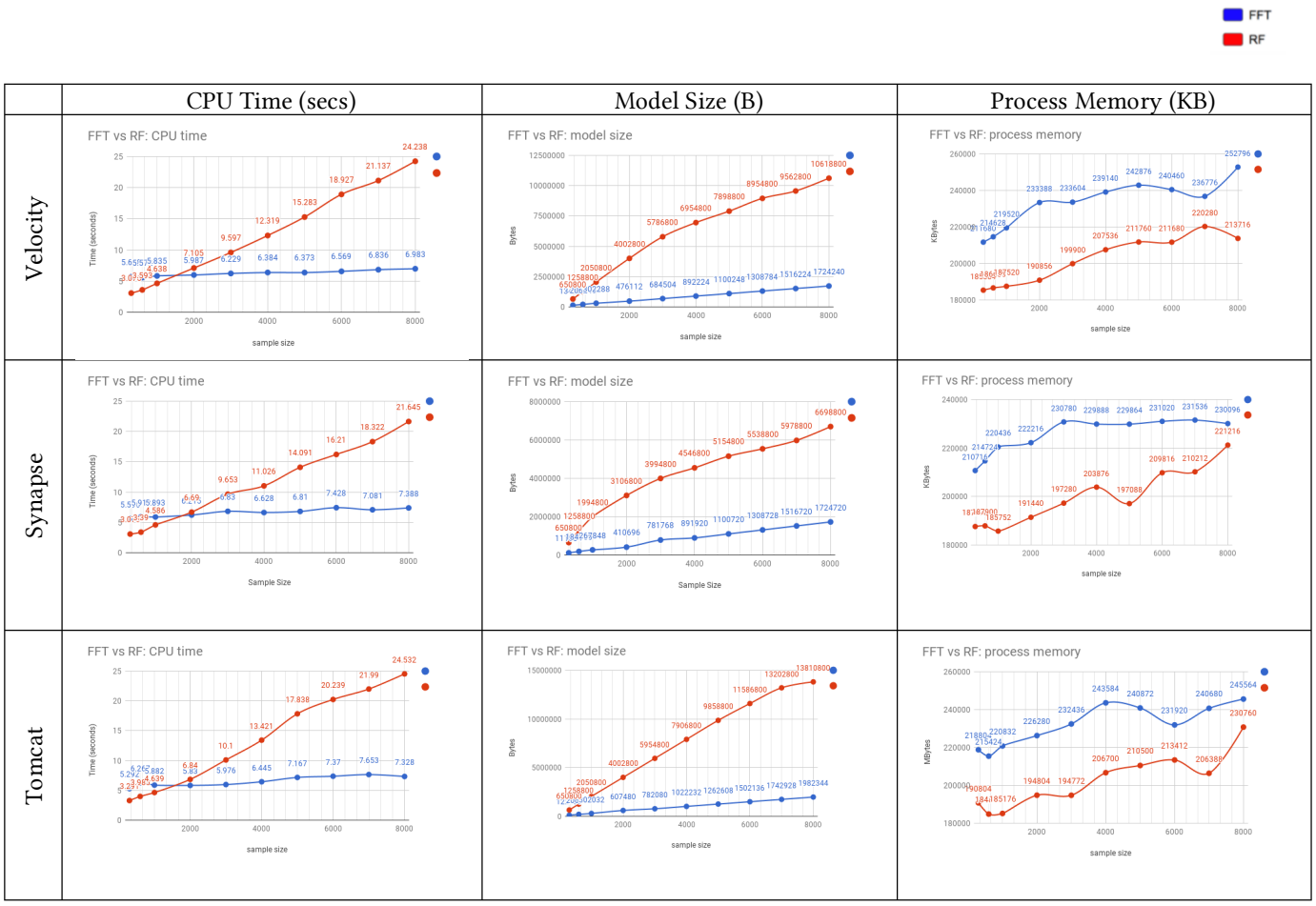

Figure 5 Runtime statistics

calculate Popt is based on the paper by Wei Fu at al. [21]. The parameters used are also defined in section 2.2 of this paper. The effort cumulative lift chart we got as shown in Figure 6.

We can see that the results are almost comparable with the Random Forest having the upper edge. There was only a difference of 0.04 when comparing Popt for both the models.

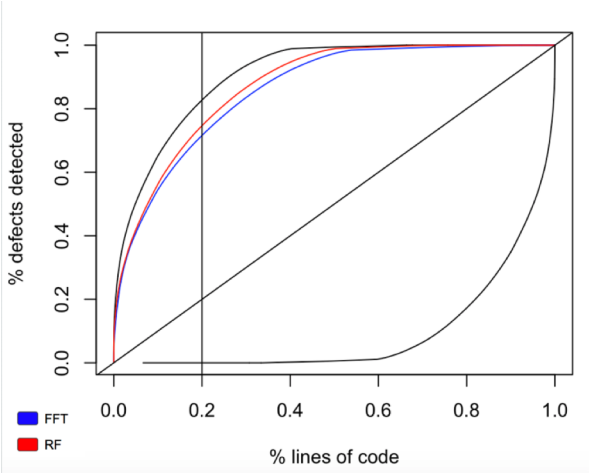

Figure 6: Popt: Random Forests vs FFT

To test the stability of our reading, we took the reading with increasing data set size and as seen from Fig 7, the Popt value remains stable across data set size 1000 to 8000.

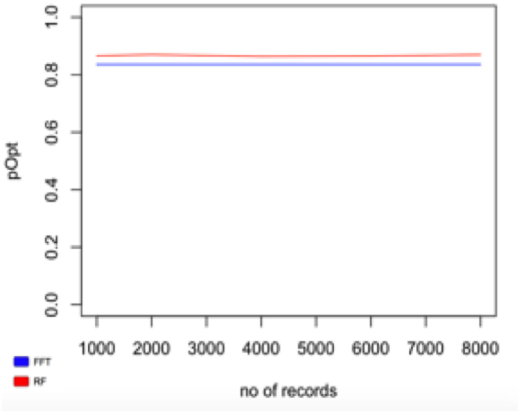

Figure 7: Popt vs. Sample sizes

All these measurements were taken with default values for all the parameters. Since readability and repeatability rather than performance was our key criteria, we did not do parameter tuning for the models. However, this is something that can be done as a future extension for the project.

As discussed in section 2.2, we also measured the precision, recall and F1 score for both the models and the results are shown below. We took 8 data points for plotting each of these parameters. We measured the performance metrics with after training both the models with 1000, 2000, 3000, 4000, 5000, 6000, 7000 and 8000 records at a time, and the results are shown in the Fig 8.

From our experiments, we can see that both the models give almost similar performance with Random Forests maintaining the slight advantage. One observation is that Random Forests have a higher precision but lower recall when compared to FFTs but the difference is very small. To validate our results, we also measured these metrics with two other datasets: Synapse and Camel and the results are shown in Table 4.

We can see that higher Popt is always achieved by Random Forest and two out of three datasets by a significant margin. For the Velocity data set, we get comparable results but for Camel and Synapse, Random Forests score much better. When comparing F1 score, we

have comparable results with the Velocity and Camel datasets but for the Synapse dataset, Random Forests fare much better.

We did not include more datasets in our study because the other datasets in the seacraft repository were either too small or had a very significant class imbalance (most of them had less than 10% defects and we did not want to test on SMOTEed data). From the results we have,

	Velocity		***Synapse***		***Camel***	
	FFT	***RF***	***FFT***	***RF***	***FFT***	***RF***
P_{opt}	0.83	0.87	0.54	0.83	0.67	0.92
Precision	0.64	0.70	0.12	0.33	0.33	0.20
Recall	0.68	0.64	0.94	0.94	0.84	0.25
F1	0.66	0.67	0.21	0.48	0.32	0.29

Table 4: Performance Metrics

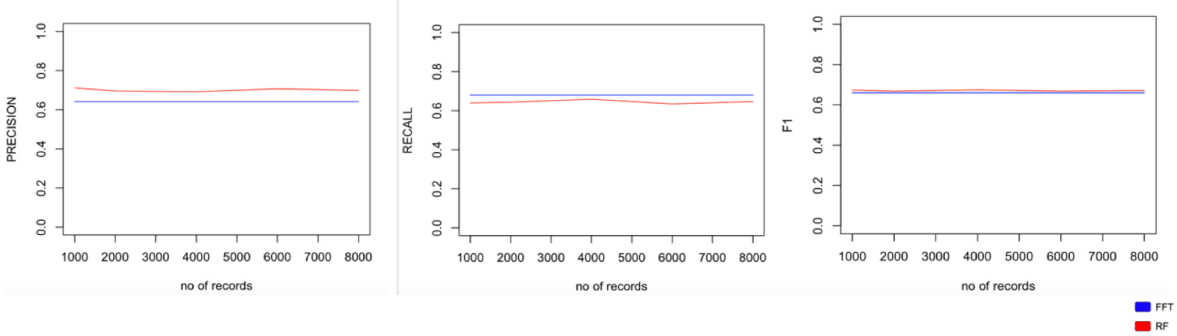

Figure 8: Performance Metrics: Random Forests vs FFTs

Random Forests perform slightly better when compared to FFTs. But since we performed all the tests without any parameter tuning/optimization, we cannot conclusively say that Random Forests always perform better than FFTs, all we can say is that with default parameters, Random Forests have a slight edge over FFTs.

5 THREATS TO VALIDITY

The conclusions in this paper are result of our empirical study and the following points should be considered when referring to these results:

Bias in sample data order: sample data bias can affect results of any classification, to counter this we select the sample data randomly so we don't get the same exact data in each run. We also run our experiments five times and find the average of results to further mitigate this sample order effect. A better way to counter bias would be doing cross validation.

The ***FFT/RF package implementations*** is another threat to validity of these results is the implementation of the R packages that we used for the experiments, specifically "process time" results from RQ2 could be affected by this. For example, we found an unexpected result for the process memory which can be caused by the implementation of the FFT package, specially because it is a fairly new package and might still need some optimizations.

Tuning the parameters and using better algorithms can possibly change the results that we get. We tried different algorithms for that are available in FFT package and as explained by Philip et al. [5], max and zig-zag were significantly (30x) faster comparing to fan algorithm which is the default in FFTrees package. However, for the purpose of this experiment we used the default ifan algorithm in FFT, and only set the number of trees/nodes in RF.

6 CONCLUSION

When it comes to readability of the model and run times, FFT is the clear winner as they perform significantly better in these areas. While testing with the maximum number of records from our data, CPU utilization for FFTs is one third that of Random Forests and the model size is one sixths that of Random Forests.

From the results, we can see that FFTs have a bigger memory footprint than Random Forests even though the model size for FFTs are six times smaller than that of Random Forests. This is something that can be explored in the future. If we have control over finer details of the FFT, we could have got different results.

When comparing model performance form the results we can see that better results are achieved by Random Forests in two out of three datasets by a significant margin. For the third data set, we get comparable results. But, in order to make a string conclusion, we should explore parameter tuning and other optimizations for both the models, otherwise it won't be a fair comparison. This is something that can be done as a future enhancement for this project.

REFERENCES

[1] Hodge, V.J. and Austin, J. (2004) *A survey of outlier detection methodologies. Artificial Intelligence Review*, 22 (2). pp. 85-126.

[2] Singh, Sachin, Pravin Vajirkar, and Yugyung Lee. "*Context-based data mining using ontologies.*" Conceptual Modeling-ER 2003(2003): 405-418.

[3] Daniel Barbara. 2002. *Applications of Data Mining in Computer Security.* Sushil Jajodia (Ed.). Kluwer Academic Publishers, Norwell, MA, USA.

[4] Gordon E Moore and others. 1998. *Cramming more components onto integrated circuits.* Proc. IEEE 86, 1 (1998), 82–85.

[5] Phillips, N. D., Neth, H., Woike, J. K., & Gaissmaier, W. (2017). FFTrees: *A toolbox to create, visualize, and evaluate fast-and-frugal decision trees. Judgment and Decision Making*, 12(4), 344-368. Retrieved from http://journal.sjdm.org/17/17217/jdm17217.pdf

[6] Wei Fu, Tim Menzies. 2017. *Easy over Hard: A Case Study on Deep Learning.* In Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8, 2017 (ESEC/FSE'17), 12 pages. DOI: 10.1145/3106237.3106256

[7] Wei Fu, Tim Menzies, Xipeng Shen. *Tuning for Software Analytics: is it Really Necessary?* CoRR, abs/1609.01759.

[8] Nathaniel D. Phillips. 2017. *FFTrees: Fast-and-frugal decision trees.* (August 2017). Retrieved October 23, 2017 from https://cran.r-project.org/web/packages/FFTrees/vignettes/guide.html

[9] Jureczko, M., Spinellis D. 2010. *Using Object-Oriented Design Metrics to Predict Software Defects. In Models and Methods of System Dependability.* Oficyna Wydawnicza Politechniki Wrocławskiej. 69-81. http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html

[10] Marian Jureczko and Lech Madeyski. 2010. *Towards identifying software project clusters with regard to defect prediction.* In Proceedings of the 6th International Conference on Predictive Models in Software Engineering (PROMISE '10). ACM, New York, NY, USA, Article 9 , 10 pages. DOI=http://dx.doi.org.prox.lib.ncsu.edu/10.1145/1868328.1868342

[11] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. 2013. *Searching for better configurations: a rigorous approach to clone evaluation.* In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 455–465.

[12] Menzies, Tim. Foundations of Software Science, txt.github.io/fss17/

[13] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. Journal of artificial intelligence research 16 (2002), 321–357

[14] Amritanshu Agrawal and Tim Menzies. 2018. Is "Better Data" Better Than "Better Data Miners"?. In Proceedings of International Conference on Software Engineering, Gothenburg, Sweden, May 2018 (ICSE'18), 12 pages. https://doi.org/10.475/123_4

[15] Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar Devanbu. 2014. Comparing Static Bug Finders and Statistical Prediction (ICSE). ACM, New York, NY, USA, 424–434. DOI:http://dx.doi.org/10.1145/2568225.2568269

[16] Cagatay Catal and Banu Diri. 2009. A systematic review of software fault prediction studies. Expert systems with applications 36, 4 (2009), 7346–7354

[17] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A systematic literature review on fault prediction performance in software engineering. IEEE TSE 38, 6 (2012), 1276–1304

[18] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs.

[19] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. IEEE Transactions on software engineering 20, 6 (1994), 476–493.

[20] Haibo He and Edwardo A Garcia. 2009. Learning from imbalanced data. IEEE Transactions on knowledge and data engineering 21, 9 (2009), 1263–1284.

[21] Wei Fu, Tim Menzies. 2017. Revisiting Unsupervised Learning for Defect Prediction. In Proceedings of 2017 11th Joint Meeting of the European So.ware Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of So.ware Engineering, Paderborn, Germany, September 4-8, 2017 (ESEC/FSE'17), 12 pages. DOI: 10.1145/3106237.3106257

[22] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Predicting the location and number of faults in large software systems. IEEE Transactions on Software Engineering, 31(4):340–355, 2005.