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 criteria 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) (2) (3) To Do

KEYWORDS

Fast Frugal Trees, Random Forests, Readability, Repeatability, SMOTE, P-OPT To Do

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.

FFTs are very popular since they are simple and enables quick and accurate decisions using limited information. Researchers have developed and analyzed FFT toolboxes [5] and compared it against other popular learners in terms of their accuracy in prediction. Speed as a performance criteria is defined as number of cues needed to build the tree, and since FFTs need lesser cues by design, they are concluded to be faster. 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.

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

Configuring SMOTE [13] to oversample all classes and not just minority class until the required sample size is achieved.

RQ2:

Result 2

RQ3:

Result 3

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"?

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 2 shows how SMOTE works. The majority classes are sub samples by deleting few data

```
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"?

samples while in super sampling, a data point in the minority class looks at it's k nearest neighbors and builds a fake member 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 distribution of the original data should be preserved 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

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
Camel	4	96				

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://tinv.cc/seacraft

```
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.

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.

applied. Note that test set data is not oversampled using SMOTE.

2.5 Performance Criteria

2.6 Computing Requirements

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. For implementations of these learners, we used the open source tool. 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 P-OPT values.

3.1 Statistical Analysis

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.

5 THREATS TO VALIDITY

6 CONCLUSION

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. In ICSE. ACM, 531–540.

[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]

[22]

[23]