RF and FFT on Studying reopened bugs in open source software

Satish Gurav
North Carolina State University
United States
sjgurav@ncsu.edu

Abhinandan Deshpande
North Carolina State University
United States
aadeshp2@ncsu.edu

Neel Kapadia
North Carolina State University
United States
ntkapadi@ncsu.edu

Abstract

Bug fixing accounts for a large amount of the software maintenance resources. Generally, bugs are reported, fixed, verified and closed. However, in some cases bugs have to be reopened. Reopened bugs increase maintenance costs, degrade the overall user-perceived quality of the software and lead to unnecessary rework by busy practitioners. We have considered two major software systems namely, Eclipse and Mozilla. We will be comparing different machine learning algorithm viz. Decision trees, Random Forests and Fast Frugal Trees. We will also discuss, critique and present various performance measures on which the algorithms will be judged. The basic aim is to play with the performance vs time trade-off in bug reopen analysis at the same time enhancing model readability.

We aim to answer the following questions:

1. Which algorithm runs faster?
2. Which algorithm gives us better precision while also providing low false alarm rates?
3. Are the results readable for naïve user?

Concepts Data Mining Algorithms, Decision trees, Random Forest, Fast frugal trees

Keywords Performance, Bug reports, Reopened bugs, Fast Frugal Trees, Random Forests, Decision Trees, SMOTE, Cross Validation, Confusion Matrix

1 Introduction

One of the major factors considered in software project management is the impact of bugs. Presence of bugs in a software can lead to huge delays and/or high costs in the software development life cycle. Even worse are the reopened bugs. They completely disturb the project life cycle. Also, the more the delay in detecting and solving the bugs, the more the resources are required to resolve it. Thus, predicting whether a bug will be reopened or not at an early stage is very essential to optimally manage resources allocated to a project.

To perform a study on reopened bugs, there are many different features which can be used. We selected the ones which we thought would make the most impact on the performance. We decided to choose a set of features which we intuitively thought will make the most impact on our study. Our study is based on the following features:

- Severity: the measure of how much impact that bug can cause to the project
- Priority: how urgent it is to resolve the bug
- Operating System: which OS is the project/module being built on
- Day: which day of the week was the bug resolved
- Time: what time of the day was the bug resolved
- Version: which version of the development environment is being used

Implementation:

- Datasets: Eclipse Bug Reports, Mozilla Bug Reports
- Validation method: Cross-validation on version of software used to resolve the bug
- Data Preprocessing: SMOTE on training data
- Models: Fast Frugal Trees, Decision Trees, Random Forests
- Performance Measures: Recall, False alarms, Precision, F1 score

Results:

On running Decision Trees, Random Forests and Fast Frugal Trees, we observed that decision trees give decent performance at a decent running time whereas fast frugal trees gave poorer performance but super fast running time and random forests gave high performance at the cost of run time.

2 Background and Motivation

2.1 Bug re-opening

In any software industry, the cost of resolving bugs is a major set-back, as the cost of fixing it increases exponentially, with the passage of development time. Here, the cost implies the human resources, time as well as the money spent on fixing the bug. The cost of fixing the bugs increases when we move from development to release. Hence, it is useful if we could anticipate whether the bug will be reopened, which can be used as an insight to re-check the solution before moving forward. In this case, we would want low false alarms, because it takes a lot of time to reevaluate the fix.

2.2 Related work

There is a paper Studying reopened bugs in open source software[1] which we referred to while working on our project. They authors of this paper have used decision tree for predicting whether the bug will be reopened or not. The authors indicated in the paper that the features they used mostly fall in four categories, viz, work habits, bug report, bug fix and team dynamics.

Work habits : Software developers are often overloaded with work. This increased workload affects the way these developers perform. For example, Sliwerski et al.(2005)[2] showed that code changes are more likely to introduce bugs if they were done on Fridays. Anbalagan and Vouk (2009)[3] showed that the time it takes to fix a bug is related to the day of the week when the bug was reported. Hassan and Zhang (2006)[4] used various work habit factors to predict the likelihood of a software build failure. These prior findings motivate them to include the work habit dimension in their study on reopened bugs. For example, developers might be inclined to close bugs quickly on a specific day of the week to reduce their work queue and focus on other tasks. These quick decisions may cause the bugs to be reopened at a later date.

Bug Report : While reporting a bug, there is a proper documentation given along with it, in order to locate the bug. Several studies use that information to study the amount of time required to fix a bug (Mockus et al. 2002)[5]. For example, Panjer (2007)[6] showed that the severity of a bug has an effect on its lifetime. In addition, a study by Hooimeijer and Weimer (2007)[7]. showed that the number of comments attached to a bug report affects the time it takes to fix it. The authors believed that attributes included in a bug report can be leveraged to determine the likelihood of a bug being reopened. For example, bugs with short or brief descriptions may need to be re-opened later because a developer may not be able to understand or reproduce them the first time around.

Bug fix: The authors found out that the bug fix dimension uses factors to capture the complexity of the initial fix of a bug, which includes three factors viz., Reporter name, Fixer name and Reporter experience, that measure the time it took to fix the bug. Along with them, the status before the bug was reopened and the number of files changed to fix the bug also impact the output.

Team dynamics: The paper indicated that most of the times, it depends on the reporter, as he couldn't explain the problem properly. And in many cases, it is the inability of a certain member of the team who fixes the bug but has less attention to detail. This feature also aligns with the seniority of the developer, the more experienced is the developer, the lesser are the chances of the bug getting reopened.

2.3 Main idea

Time required and Readability of the results is the major concern in our project, as it should be easily explainable to the stakeholders. Hence, we chose fast frugal trees and random forests for the experiment. The paper that we referred, used decision trees for the prediction. So we made an attempt to increase either the precision(by using random forests) or the speed (by using the FFTs).

2.3.1 Fast Frugal Trees

A fast-and-frugal tree (FFT) is a set of hierarchical rules for making decisions based on very little information (usually 4 or fewer). Specifically, it is a decision tree where each node has exactly two branches, where one (or in the cast of the final node, both) branches is an exit branch.

This is an example of FFT:

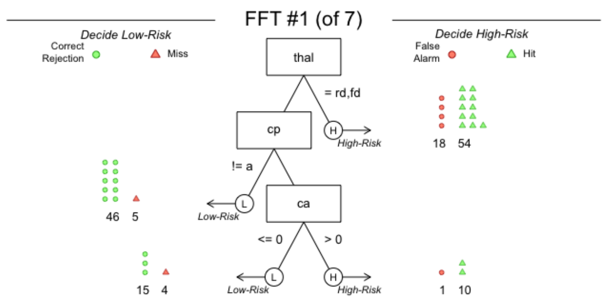


Fig. 1 Sample Fast frugal tree

FFT is generally used when the number of features is around 4. Hence, it is an ideal choice for this research topic. Fast Frugal Trees have the following properties:

- Ecologically rational (that is, they exploit structures of information in the environment).
- Founded in evolved psychological capacities such as memory and the perceptual system.
- Fast, frugal, and simple enough to operate effectively when resources might be limited.
- Precise enough to be modeled computationally.
- Powerful enough to model both good and poor reasoning.

2.3.2 Random Forest

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set.
An example of Random Forest for the golf dataset is given by:

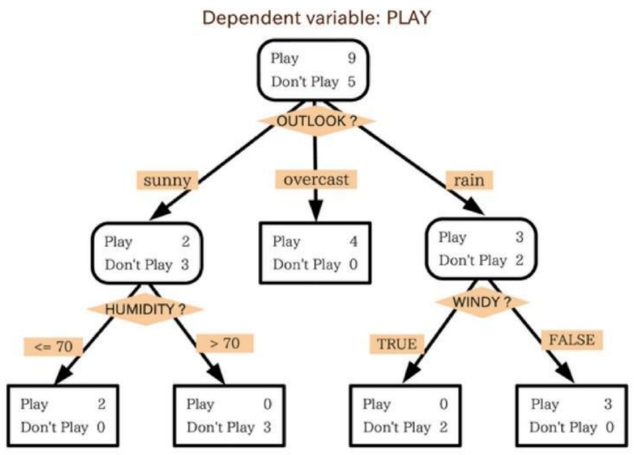


Fig. 2. Sample Random Forest

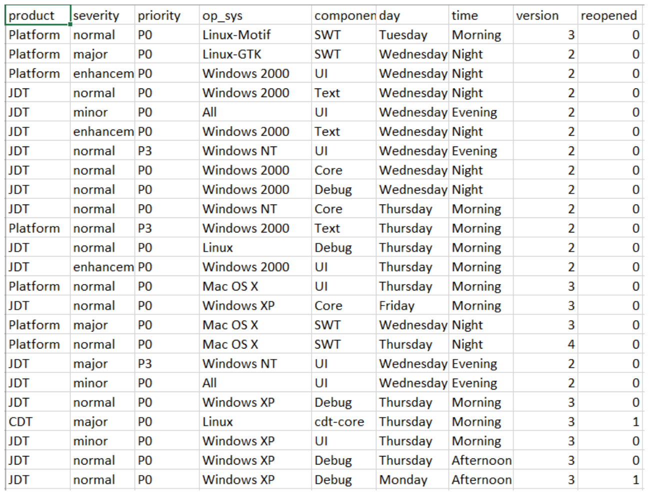
Hence, we implemented both of them and studied the difference in their performance on the following features:

1. Readability
2. Speed
3. Precision
4. False Alarm

3. Experimental Design

3.1 Dataset

The datasets we used in the experiment are of Eclipse and Mozilla. The features which we used are product, severity of the bug, priority of the bug, operating system in which the bug has occured, day of the week when it was fixed , time of the day when the bug was fixed. Figure 3 is the data snapshot of our data.



product	severity	priority	op_sys	componen	day	time	version	reopened
Platform	normal	P0	Linux-Motif	SWT	Tuesday	Morning	3	0
Platform	major	P0	Linux-GTK	SWT	Wednesday	Night	2	0
Platform	enhancem	P0	Windows 2000	UI	Wednesday	Night	2	0
JDT	normal	P0	Windows 2000	Text	Wednesday	Night	2	0
JDT	minor	P0	All	UI	Wednesday	Evening	2	0
JDT	enhancem	P0	Windows 2000	Text	Wednesday	Night	2	0
JDT	normal	P3	Windows NT	UI	Wednesday	Evening	2	0
JDT	normal	P0	Windows 2000	Core	Wednesday	Night	2	0
JDT	normal	P0	Windows 2000	Debug	Wednesday	Night	2	0
JDT	normal	P0	Windows NT	Core	Thursday	Morning	2	0
Platform	normal	P3	Windows 2000	Text	Thursday	Morning	2	0
JDT	normal	P0	Linux	Debug	Thursday	Morning	2	0
JDT	enhancem	P0	Windows 2000	UI	Thursday	Morning	2	0
Platform	normal	P0	Mac OS X	UI	Thursday	Morning	3	0
JDT	normal	P0	Windows XP	Core	Friday	Morning	3	0
Platform	major	P0	Mac OS X	SWT	Wednesday	Night	3	0
Platform	normal	P0	Mac OS X	SWT	Thursday	Night	4	0
JDT	major	P3	Windows NT	UI	Wednesday	Evening	2	0
JDT	minor	P0	All	UI	Wednesday	Evening	2	0
JDT	normal	P0	Windows XP	Debug	Thursday	Morning	3	0
CDT	major	P0	Linux	cdt-core	Thursday	Morning	3	1
JDT	minor	P0	Windows XP	UI	Thursday	Morning	3	0
JDT	normal	P0	Windows XP	Debug	Thursday	Afternoon	3	0
JDT	normal	P0	Windows XP	Debug	Monday	Afternoon	3	1

Fig. 3. Data snapshot

3.2 Handling data imbalance with SMOTE

In our two datasets, the ratio of the non-reopened to reopened bugs was around 90 : 10. Hence, we used SMOTE on the training data to make the data of both the samples to be equal.

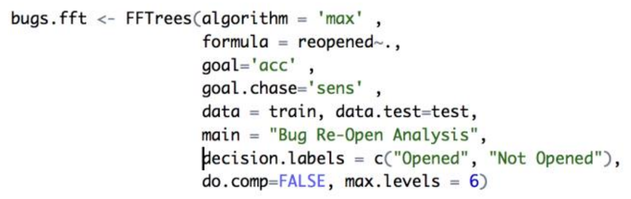
SMOTE handles class imbalance by changing the frequency of different classes of the training data[8]. The algorithm's name is short for "synthetic minority over-sampling technique". When applied to data, SMOTE sub-samples the majority class (i.e., deletes some examples) while supersampling the minority class until all classes have the same frequency. In the case of software defect data, the minority class is usually the defective class. During supersampling, a member of the minority class finds k nearest neighbors. It builds a fake member of the minority class at some point in-between itself and one of its nearest neighbors. During that process, some distance function is required which is the minkowski_distance function.

3.3 Implementation

We have different versions of Eclipse and Mozilla and we found out that we could use round robin on the data sets. We trained on the previous versions and tested on the newer version. Hence, first we separate the training and test data and then apply SMOTE on the training data.
Then we apply different algorithms on this new training data which has equal number of reopened and non-reopened bugs. We will start with fast frugal trees.

3.3.1 Fast Frugal Trees

The code snippet in R is attached below,

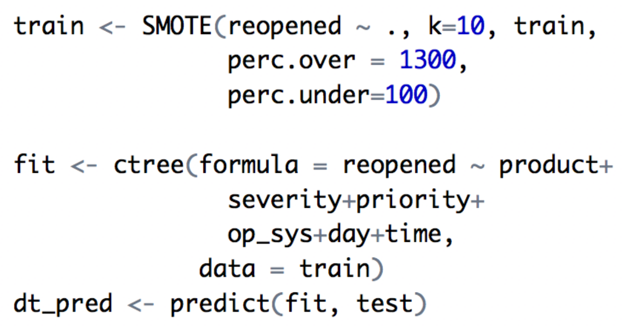


```
bugs.fft <- FFTrees(algorithm = 'max' ,
                    formula = reopened~.,
                    goal='acc' ,
                    goal.chase='sens' ,
                    data = train, data.test=test,
                    main = "Bug Re-Open Analysis",
                    decision.labels = c("Opened", "Not Opened"),
                    do.comp=FALSE, max.levels = 6)
```

Fig 4. FFT code snapshot

In Fig.4 ,
1.Algorithm is the one used to determine the best tree amongst the many trees created in the process.
2.The goal value is the goal that we want to maximize.
3.The 'goal.chase' is the value that we need to maximize within the goal. Here, we have preferred *sensitivity.*
4.Data is to specify the training data.
5.Data.test is set to the test data
6.max.levels can be used to specify the maximum levels in a tree. Default value for this is 4.

3.3.2 Decision Trees



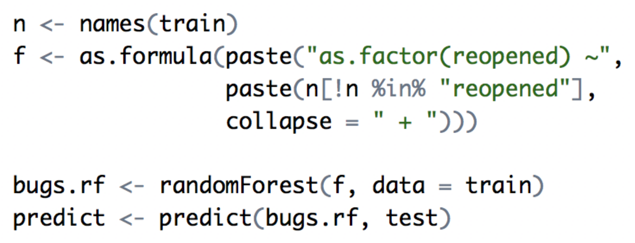
```
train <- SMOTE(reopened ~ ., k=10, train,
               perc.over = 1300,
               perc.under=100)

fit <- ctree(formula = reopened ~ product+
               severity+priority+
               op_sys+day+time,
             data = train)
dt_pred <- predict(fit, test)
```

Fig 5. Decision trees

We used '*ctree*' function in R language which creates a decision tree, where it takes the following attributes as its input,
1.Data: The training data to train on
2.Formula: It takes the dependent and independent variables.
Then we use the '*predict*' function to test the already created tree. We give tree and the test data as input and it gives us the prediction list of output.

3.3.3 Random Forest



```
n <- names(train)
f <- as.formula(paste("as.factor(reopened) ~",
                paste(n[!n %in% "reopened"],
                collapse = " + ")))

bugs.rf <- randomForest(f, data = train)
predict <- predict(bugs.rf, test)
```

Fig. 6 Random Forest code

We used 'randomForests' function from the 'randomForest' library in R. We created 500 trees for each iteration. The random forests take a long time to build each tree and come to a conclusion.

4. Results

We compared all the three algorithms on the following features:

1. Readability
2. Speed
3. Precision
4. False Alarm

Readability:
Readability is a feature which is undoubtedly better in Fast Frugal Trees. Fig. 7, demonstrates the ease of understanding in FFTs.

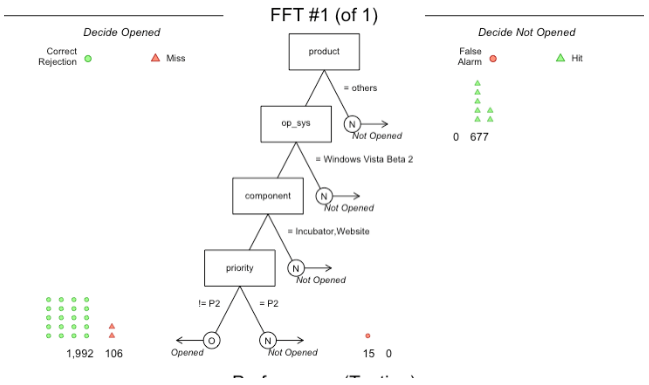


Fig. 7. Readability of FFT

Decision trees are relatively better than Random Forests in terms of readability.

Speed:
All the algorithms were run on a MacBook pro 2.3GHz Dual-Core processor with 16GB RAM. The average time for each algorithm is:

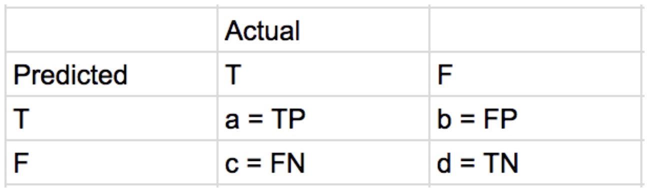
RF(7mins 28sec) > DT (5min 12sec) > FFT (3min 40sec)

Hence, FFTs are faster than the other two, but the difference between FFT and DT is not very significant.

Other parameters for comparison are clubbed in one graph.

4.1 Separate Results for one tree

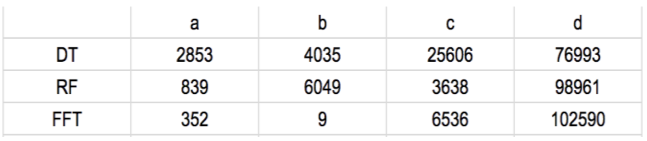
After trying all the algorithms on the two datasets, viz. Eclipse and Mozilla, we got the confusion matrix for all of them. Figure 8 represents the confusion matrix of FFT, decision trees and random forests. This is what we considered as our confusion matrix,



	Actual	
Predicted	T	F
T	a = TP	b = FP
F	c = FN	d = TN

Fig.8 Confusion matrix for reference

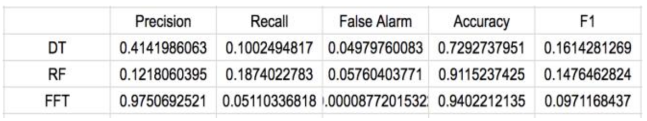
We got the following values of a,b,c and d for each of the following algorithms. The values in the Fig. 9 are for the 1st version. We have many such values, one for each version.



	a	b	c	d
DT	2853	4035	25606	76993
RF	839	6049	3638	98961
FFT	352	9	6536	102590

Fig. 9 Confusion matrix values for each algorithm

Using the confusion matrix in the Fig. 9 , we calculate precision, recall, false alarm, accuracy and F1 score. Fig. 10 is a snapshot of the calculated values for the 1st version.

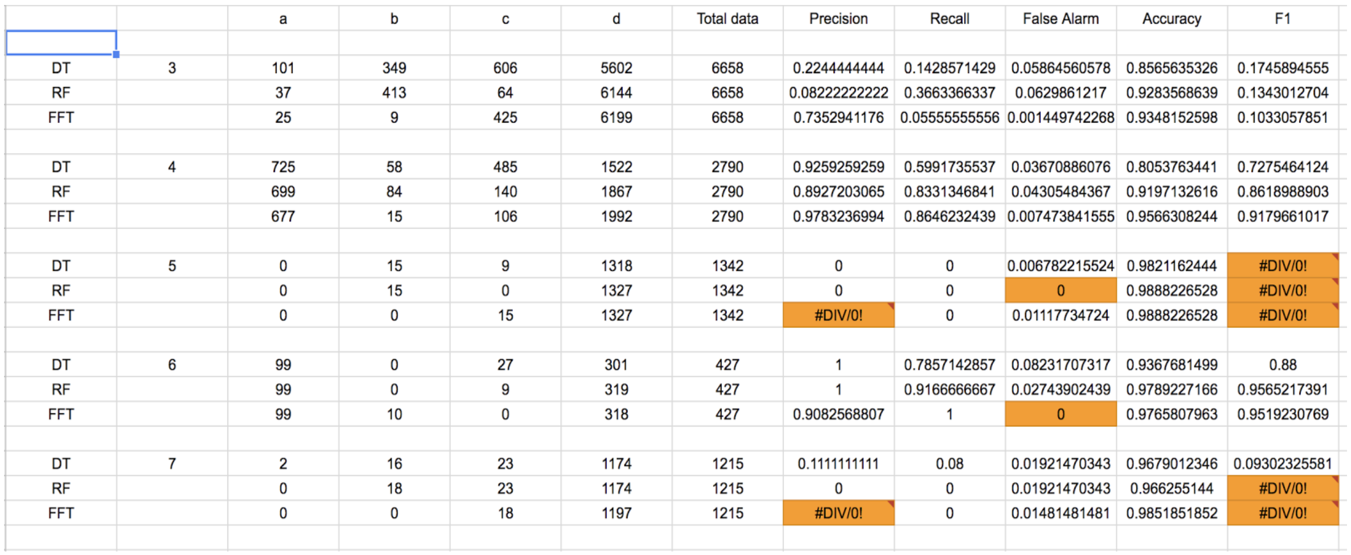


	Precision	Recall	False Alarm	Accuracy	F1
DT	0.4141986063	0.1002494817	0.04979760083	0.7292737951	0.1614281269
RF	0.1218060395	0.1874022783	0.05760403771	0.9115237425	0.1476462824
FFT	0.9750692521	0.05110336818	).0000877201532	0.9402212135	0.0971168437

Fig. 10 Comparison values for 1st version

Fig.10 is the snapshot of the confusion matrix for the cross-validation. The highlighted cells are the ones where there are anomalies. We can see that these anomalies can be seen in the cases when there are fewer data points. We would like to mention here that, such cases are not considered while calculating the aggregate values of the performance index, since such data can skew the results to unknown extremes.

In the next section, we are reporting the aggregate values of all the trees for each dataset, and we will comment on the performance of each of the algorithms cumulatively.



		a	b	c	d	Total data	Precision	Recall	False Alarm	Accuracy	F1
DT	3	101	349	606	5602	6658	0.2244444444	0.1428571429	0.05864560578	0.8565635326	0.1745894555
RF		37	413	64	6144	6658	0.08222222222	0.3663366337	0.0629861217	0.9283568639	0.1343012704
FFT		25	9	425	6199	6658	0.7352941176	0.05555555556	0.001449742268	0.9348152598	0.1033057851
DT	4	725	58	485	1522	2790	0.9259259259	0.5991735537	0.03670886076	0.8053763441	0.7275464124
RF		699	84	140	1867	2790	0.8927203065	0.8331346841	0.04305484367	0.9197132616	0.8618988903
FFT		677	15	106	1992	2790	0.9783236994	0.8646232439	0.007473841555	0.9566308244	0.9179661017
DT	5	0	15	9	1318	1342	0	0	0.006782215524	0.9821162444	#DIV/0!
RF		0	15	0	1327	1342	0	0	0	0.9888226528	#DIV/0!
FFT		0	0	15	1327	1342	#DIV/0!	0	0.01117734724	0.9888226528	#DIV/0!
DT	6	99	0	27	301	427	1	0.7857142857	0.08231707317	0.9367681499	0.88
RF		99	0	9	319	427	1	0.9166666667	0.02743902439	0.9789227166	0.9565217391
FFT		99	10	0	318	427	0.9082568807	1	0	0.9765807963	0.9519230769
DT	7	2	16	23	1174	1215	0.1111111111	0.08	0.01921470343	0.9679012346	0.09302325581
RF		0	18	23	1174	1215	0	0	0.01921470343	0.966255144	#DIV/0!
FFT		0	0	18	1197	1215	#DIV/0!	0	0.01481481481	0.9851851852	#DIV/0!

Fig. 11 Snapshot of the confusion Matrices

4.2 Aggregated Results

4.2.1 Eclipse

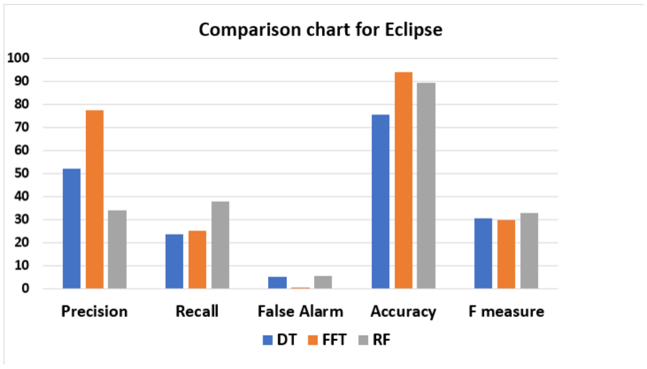


Fig. 12 Comparison chart for Eclipse dataset

From the Fig. 12, we can see that all the algorithms perform more or less in a similar manner. The average precision of FFT is better than others, while the recall is compromised. Balancing the precision and recall, all the three algorithms have similar values for F measure.

Except precision, none of the other parameters are significantly distinct. Now, we will see the results when we run the same code on the other dataset.

4.2.2 Mozilla

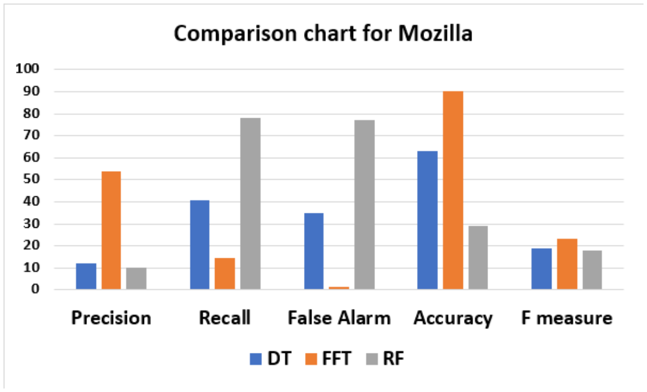


Fig. 13. Comparison chart for Mozilla

In the Mozilla dataset, we can observe that the decision trees and random forests are behaving similarly, while the FFT is behaving in opposite manner.

Precision of FFT is significantly higher than DT and RF. While Recall of RF is way higher than the other two. Combined accuracy of FFT is significantly higher than DT and RF.

The F1 score is more or less the same for all the algorithms.

Hence, we can confidently recommend FFT in this application of machine learning as it is faster and more readable. And the performance measures are not significantly different.

5. Threats to Validity

In this section, we discuss the possible threats to validity of our study,

Threats to Construct Validity consider the relationship between theory and observation, in case the measured variables do not measure the actual factors. Some of the re-opened bugs considered in our study were re-opened more than once. In such cases, we predict for the first time the bug was re-opened. In future studies, we plan to investigate bugs that are re-opened several times. In the future, we also plan to incorporate one more parameters, fixer, which is nothing but the person who has done the fixing. Also, if the reopened bugs was reopened again and the fixer value was changed then we also plan to incorporate it into our study. In the future, we plan to use heuristics that may improve the accuracy of the fixer name factor.

Also, when we considered time factor we did not consider time-zone information as it was not available to us. But incorporating that in the future may influence our study.

Threats to Internal validity refers to whether the experimental conditions makes a difference or not, and whether there is sufficient evidence to support the claim being made.

In our study, when we divided the data sets on versions, we found out that many of the version do not contain proper information and thus we ignored them. The percentage of the data that we ignored was almost the half original data, but as existing paper mentioned, it is a common phenomenon in studies using bug reports.

Threats to External Validity consider the generalization of our findings. In this study, we used two large, well established Open Source projects to conduct our case study. Although these are large open source projects, our results may not generalize (and as we have seen do not generalize) to all open source or commercial software projects.

As existing paper used decision tree to predict reopened bugs, we also ran the same on our dataset so that we have a base system on which we can compare other ML algorithms that we used. We also made use of FFTrees to predict as it was selected for its speed and low memory footfrint. Also, RF was used as a third algorithm to achieve accuracy and avoid overfitting.

In our manual examination of the re-opened bugs, we only examined a sample of bug reports available for Eclipse and Mozilla. The purpose of this analysis was to shed some light on the type of information that we were able to get from the re-opened bug reports. Our findings may not generalize to all re-opened bugs.

6. Conclusion and Future Work

Prior work on bug reopen analysis showed a precision of ___, recall of ___ and an accuracy of __ using Decision Trees. We achieved comparable results (or even better) using FFTs and RFs. FFTs help achieve readability and reduces time taken to build the model. RFs take longer time to run but give a better accuracy. One interesting result which we achieved was reduction of false alarms by FFTs. Fast Frugal trees have a lower accuracy because of a higher number of misses.

So, to conclude we achieved better results by using the following techniques:

- Applying SMOTE on training data: In software bug analysis data, on an average 7 out of 100 bugs are reopened. Thus, there is an imbalance in the output classes. This leads to poor training of the model thus giving low performance. SMOTE helps balance the data amongst output classes and hence the model is trained better.
- Cross validation on versions: One of the features in software bug analysis data is version of the software on which the bug was resolved. Say there are 5 versions in the data. We trained on 4 versions and tested on the 5th, then we trained on other 4 and tested on the other 5th and so on. The final performance was the average of each test performance. Cross validation on versions help train the model on the whole data and test also on entire data.

The future scope of the project includes training the model on even more different datasets and performing Round-Robin validation on these datasets. Round-robin validation will help evaluate the performance on many datasets.

7. References

[1] Studying reopened bugs in open source software

[2] S´ liwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: MSR ’05: proceedings of the 2005 international workshop on mining software repositories, pp 1–5

[3] Anbalagan P, Vouk M (2009) “Days of the week” effect in predicting the time taken to fix defects. In: DEFECTS ’09: proceedings of the 2nd international workshop on defects in large software systems, pp 29–30

[4] Hassan AE, Zhang K (2006) Using decision trees to predict the certification result of a build. In: ASE ’06: proceedings of the 21st IEEE/ACM international conference on automated software engineering, pp 189–198

[5] Mockus A, Fielding RT, Herbsleb JD (2002) Two case studies of open source software development: Apache and mozilla. ACM Trans Softw Eng Methodol 11(3):309–346

[6] Panjer LD (2007) Predicting eclipse bug lifetimes. In: MSR ’07: proceedings of the fourth international workshop on mining software repositories, p 29

[7] Hooimeijer P, Weimer W (2007) Modeling bug report quality. In: ASE ’07: proceedings of the twenty-second IEEE/ACM international conference on automated software engineering, pp 34–43

[8] 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