When making up data is a good idea: On the advantages of partially synthetic training sets for software analytics

Amritanshu Agrawal
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
Email: aagrawa8@ncsu.edu

Raghavendra Prasad Potluri
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
Email: rpotlur@ncsu.edu

***Abstract*—The accurate prediction of where faults are likely to occur in code can help direct test effort, reduce costs and improve the quality of software. One of the approaches to tackle the problem, is based on relying on code metrics especially CK metrics. The aim of the project was to reproduce a package showing comparative study of 6 learners for defect prediction using CK metrics. We successfully made a python pip package and also found an astonishing result. Using SMOTE brought out a clear winner of which learner to use. Other users will now be able to use this package and refute our results as well as get future results in no time. There are still various improvements that can be done which we will be publishing in our next revised package.**

***Keywords*—Defects prediction, code metrics, classification.**

I. Introduction

Software defect prediction has been an important research topic in the software engineering field for more than 30 years. It has generated widespread interest for a considerable period of time. The driving scenario is resource allocation: Time and manpower being finite resources, it makes sense to assign personnel and/or resources to areas of a software system with a higher probable quantity of defects. Current defect prediction work focuses on (i) estimating the number of defects remaining in software systems, (ii) discovering defect associations, and (iii) classifying the defect-proneness of software components, typically into two classes defect-prone and not defect-prone.

There have been vast amount of studies done to find the best defect prediction performing model. But literature suggests, that no single prediction technique dominates and making sense of the many prediction results is hampered by the use of different data sets, data pre-processing, validation schemes and performance statistics. We highly agree to this given so many variations available in the data and there are so many classification techniques available like Statistical, Clustering, Rule-Based, Neural Networks, Nearest Neighbour, Support Vector Machines, Decision trees, ensemble methods, to name a few.

This project deals with the third type of problem for code metrics which is classifying the defect-proneness of software components, typically into two classes defective and not defective. Ghotra et al. [7] did a comparative study on various learners for defect prediction. They found out that mainly 6 learners have been performing well namely Naive Bayes, Logistic regression, Support Vector Machines, Nearest Neighbor, decision tree and Random forest. Our project considered Ghotra et al. results as the baseline results.

For the reproduction package we generalised our datasets to be comprised of CK metrics [4]. The CK metrics aim at measuring whether a piece of code follows OO principles. It contains a check of these OO design attributes:

- **Weighted Methods for Class (WMC)** The sum of the complexities of each method in a class. If all the method complexities are considered equal and have the value of 1 (as proposed in the chidamber94), then WMC equals the number of methods in a class.
- **Depth of Inheritance Tree (DIT)** Number of classes that a particular class inherits from.
- **Number of Children (NOC)** The count of immediate subclasses of a class.
- **Response for Class (RFC)** The number of elements in the response set of a class. The response set of a class (as defined in chidamber94) is the number of methods that can potentially be executed in response to a message received by an object of that class.
- **Lack of Cohesion of Methods (LCOM)** For a class C, LCOM is the number of method pairs that have no common references to instance variables minus the number of method pairs that share references to instance variables.
- **Coupling Between Objects (CBO)** For a class C, CBO is the number of classes that are coupled to (i.e. call a function or access a variable of) C.

We created a Pip Package generalised to run any CK metrics based dataset and compare results against 6 learners. Since the classes are imbalanced (less defective class about 15 percent), we used SMOTE [3] (only on Training Data) which is a synthetic minority over-sampling technique.

The remainder of this report is organized as follows. Section II gives a brief related work on defect prediction. Since we found an astonishing results on smote, section III tells about

SMOTE. Experimental setup is provided in section IV. Then results are discussed in Section V. Final conclusion is being discussed in section VI. And section VII talks about future work.

II. Related Work

There are works on defect prediction which employs statistical approaches, capture-recapture (CR) models, and detection profile methods (DPM) [19]. The second type of work borrows association rule mining algorithms from the data mining community to reveal software defect associations [20]. A variety of approaches have been proposed to tackle the third type of problem, relying on diverse information, such as code metrics [13], [5], [9], [16], [18] (lines of code, complexity), process metrics [10] (number of changes, recent activity) or previous defects [12].

Some other research [2] indicate that it is possible to predict which components are likely locations of defect occurrence using a component's development history, and dependency structure. Two key properties of software components in large systems are dependency relationships (which components depend on or are dependent on by others), and development history (who made changes to the components and how many times). Thus we can link software components to other components a) in terms of their dependencies, and also b) in terms of the developers that they have in common. Prediction models based on the topological properties of components within them have proven to be quite accurate [24].

Result by Tantithamthavorn et al. [22] also suggested that every dataset comes with different attributes. And also classification techniques often have configurable parameters that control characteristics of these classifiers that they produce. Now time has come to even think about hyperparameter optimization of these techniques and come up with an automated process [6], [1] to tune these parameters for every dataset.

And lastly we found a paper from Ghotra et al. [7] on "Revisiting the impact of classification techniques on the performance of defect prediction models". To compare the performance of defect prediction models, they used the Area Under the receiver operating characteristic Curve (AUC), which plots the false positive rate against the true positive rate. They ran the Scott-Knott test to group classification techniques into statistically distinct ranks

III. Case Study of SMOTE in Defect Prediction

There have been various oversampling and undersampling techniques available. And SMOTE [3] has become increasingly popular in recent times. To find out the effects of smote in Software defect prediction, we found out few papers. Some researchers [8], [23] studied various undersampling and oversampling technique and compared the results with Naive Bayes and random forest. Some found out that AdaBoost.NC had a better performance than the rest while others are planning to use SMOTE. Overall oversampling and undersampling of data has been useful.

Pelayo et al. [15] studied the effects of percentage of oversampling and undersampling done. They found out that different percentage of each helps improve the accuracies of decision tree learner for defect prediction using CK metrics.

IV. Experimental Setup

A. *Data*

We used the data sets available in promise repository[1]. Totally 14 data sets are used. These datasets have been collected from the work done by Jureczko et al. [11]. Namely these datasets are:

- ANT
- ARC
- CAMEL
- IVY
- JEDIT
- LOG4J
- POI
- REDAKTOR
- SYNAPSE
- TOMCAT
- VELOCITY
- XALAN
- XERCE

B. *Preprocessing*

CK metrics comprises mostly of numerals. But to generalize the package we added pre-processing component. We ignore any string columns in the data. We assume the last column in the data sets is the class label. Originally, the target class contains number of defects. We converted them into binary, i.e if target class has defect then it represents 1 otherwise its 0. The package assumes user has preprocessed the data before passing it to the learners.

C. *Classifiers*

We used six classifiers which are mentioned in the baseline paper [7]. Beside every learner we wrote some important parameter which are hardcoded as suggested by Ghotra et al.

- **Support Vector Machine (Linear Kernel)** In machine learning, support vector machines (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier.
- **Logistic Regression** In statistics, linear regression is an approach for modeling the relationship between a scalar dependent variable y and one or more explanatory variables (or independent variables) denoted X.

[1] http://openscience.us/repo/defect/ck/

- **Naive Bayes** In machine learning, naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes theorem with strong (naive) independence assumptions between the features.
- **K Nearest Neighbors (K=8)** In pattern recognition, the k-Nearest Neighbors algorithm (or k-NN for short) is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space.
- **Decision Trees (CART, Split Criteria=Entropy)** A decision tree is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm.
- **Random Forest (Split Criteria=Entropy)** 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.

We are using K=8 for K Nearest Neighbours because it was proposed to perform better. Also for Decision trees and Random Forest we are using Entropy as split criteria. Scikit-Learn [14] provides this feature of selecting the split criteria. Since the classes are highly imbalanced we want training and testing set to have same proportions of both the classes. That's why we used stratified 5- Fold Cross-validation as default.

D. *Package*

We implemented the package in Python 2.7 but added support for Python 3.x as well. Our package depends on scikit-learn and NLTK. It has been written in pep8 standards to ensure it can be deployed in pip. Users can import Learner module, which requires file name as mandatory parameter, folds and splits as optional parameters. The default values of folds and splits are 5, and 5 respectively. The smoting can be turned on or off by passing smoting flag to the learner. We are using "ball tree" nearest neighbor algorithm for smoting. The run method accepts a list of learners to be used.

```
Sample Execution:
    import Learner
    learner = Learner("./data/ant.csv")
    learner.run()
    learner.stats()
```

The execution procedure is

- Csv file input is parsed. Converting integers to float.
- The data is now shuffled and dropped into bins using StratifiedKFold.
- The unbalanced class is smoted, depends on the user choice.
- Data is passed to each learner and its predicted value for target class is captured.
- A stats file (Scott-Knott) is used to calculate the measures using the predicted target class.
- The stats are aggregated in result object and displayed upon execution of all learners.

There are certain helper functions for user to just calculate one or more of the measures, recall, f1 score, accuracy, precision and false alarm. A helper function to display available learners is also implemented.

E. *Evaluation Measures*

Since, this is a multi-class classification problem, we represent the predictions using a confusion matrix where a 'positive' output is the class under study and a 'negative' output is the non defective class. The confusion matrix is shown in figure 1.

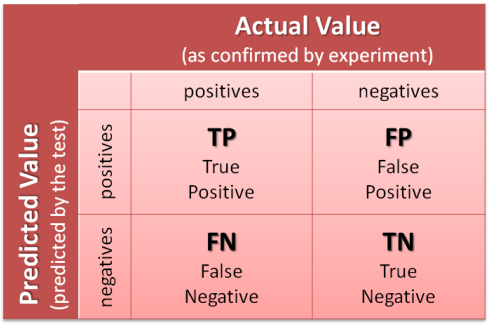

Figure 1: Confusion Matrix

We define the measures as

- **Recall** is the fraction of relevant instances that are retrieved.

$$Recall(rec) = \frac{TP}{TP + FN}$$

- **Precision** is the fraction of retrieved instances that are relevant.

$$Precision(prec) = \frac{TP}{TP + FP}$$

- **F1 Score** A measure that combines precision and recall which is the harmonic mean of precision and recall.

$$F1 = \frac{2 * prec * rec}{prec + rec}$$

- **Accuracy** is a weighted arithmetic mean of Precision and Inverse Precision (weighted by Bias) as well as a weighted arithmetic mean of Recall and Inverse Recall.

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

- **False Alarm** is the ratio of false positive to predicted negative total.

$$Falsealarm(pf) = \frac{FP}{FP + TN}$$

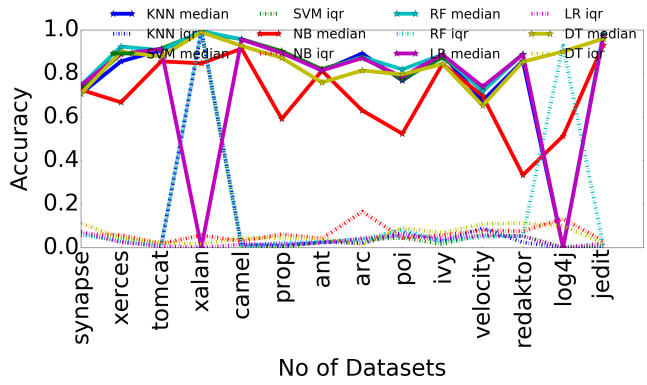

Figure 2a: Accuracy Without smote.

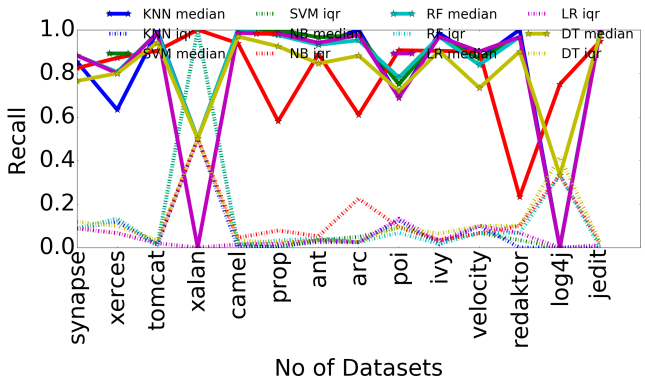

Figure 2b: Recall Without smote.

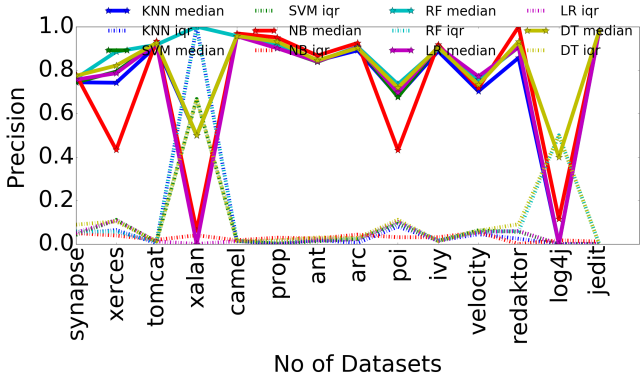

Figure 2c: Precision Without smote.

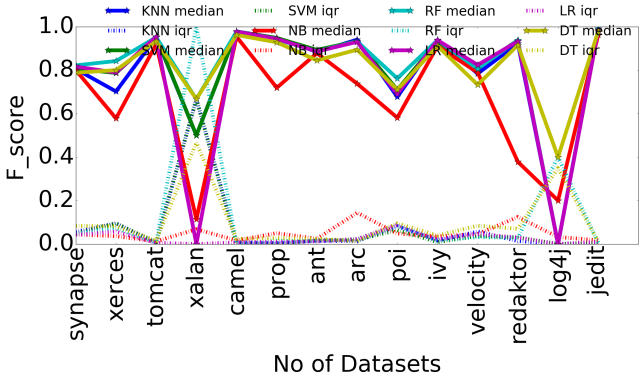

Figure 2d: F1 score Without smote.

Figure 2: Results comparing 6 learners for each measure without smote

F. The Scott-Knott Test

Scott-Knott Effect Size Difference (ESD) test is an enhancement of the standard Scott-Knott test (which cluster distributions into statistically distinct ranks [17]), which makes no assumptions about the underlying distribution and takes the effect size into consideration [21]. After the predicted values are computed, we pass the accumulated values to a Scott-Knott Test which compares 6 learners for each of these measures F-Score, Recall, Accuracy, Precision and False Alarm. The results are available to user by helper methods.

G. The visualizations

We have included a visualization script, when executed with a pickle file displays the pretty visualization of all the learners. Currently this is not implemented as a part of output. A user has to dump the results in a pickle file and run the script manually.

V. Results

Figures 2 and 3 represents all the datasets without performing smote and with smote respectively. 4 different graph represents each evaluation measure. X-axis represents all the 14 datasets and y-axis represents scores. Each splod line represents median value for 6 learners and dashed line represents variance of 25 numbers (ie, 75th percentile - 25th percentile). From the graphs we can see how stable our results are since variance is less than 10%.

Now to see exact difference between smote and without smote, see figures 4 and 5. These figures show each learners all evaluation scores and column represents each dataset median values. Now to read these figures take any colum, and select 1 evaluation measure and follow its background color in the same column. The bold mark for such color represents which learner preforms the best. From figure 4, we observed that Logistic regression, random forest and nearest neighbor has almost equal proportions of winners among different datasets which was the expected results from our baseline paper [7].

But we found astonishing results while reading the numbers from figure 5 which is with smote. We observed that for recall, f1 score and accuracy, Random forest performed the best. Also, we got less variance in smote results than the non-smote.

The run times for most of the data sets except 2 data sets Tomcat and Xalan took close to 20 minutes. For Tomcat and Xalan it took close to 2 hours. The package was deployed on HPC machines.

VI. Conclusion

We were able to reproduce the baseline paper ”Revisiting the impact of classification techniques on the performance of defect prediction models” with the same conclusions what

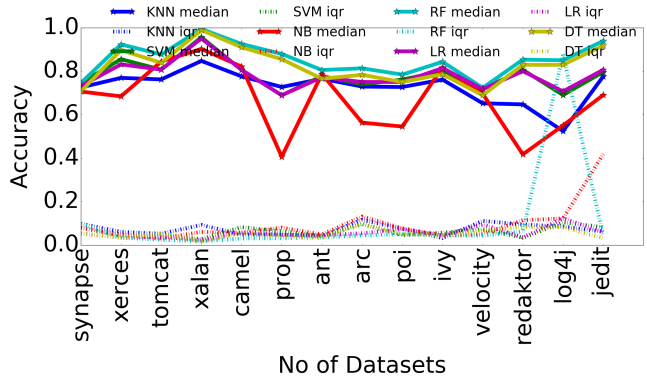

Figure 3a: Accuracy With smote.

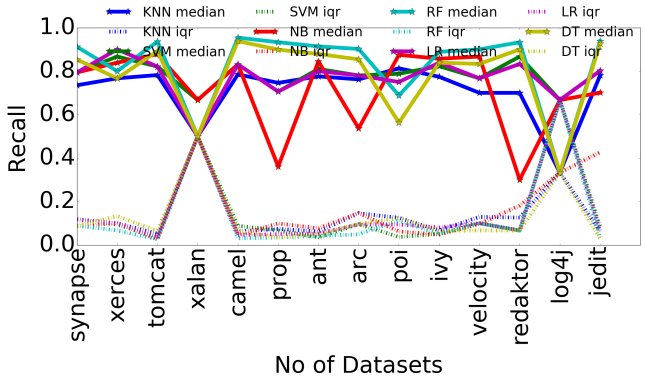

Figure 3b: Recall With smote.

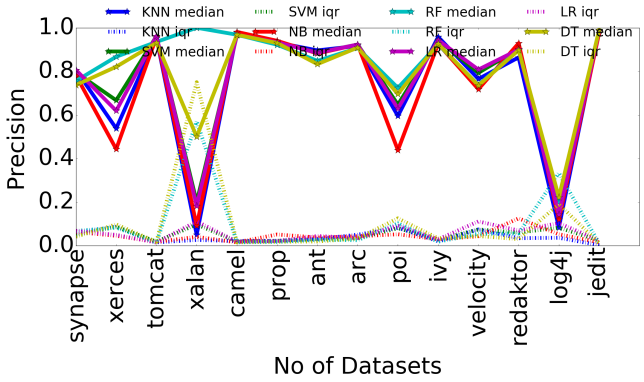

Figure 3c: Precision With smote.

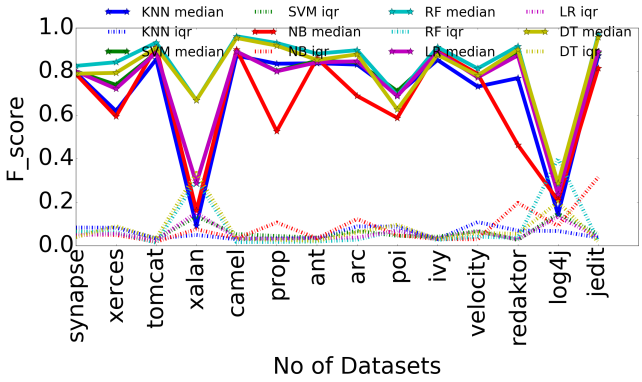

Figure 3d: F1 score With smote.

Figure 3: Results comparing 6 learners for each measure with smote

Learners	Measures	synapse	xerces	tomcat	xalan	camel	prop	ant	arc	poi	ivy	velocity	redaktor	log4j
KNN	Recall	0.85	0.63	0.99	0.5	1	1	0.93	1	0.69	0.98	0.87	1	0
	Precision	0.74	0.74	0.91	0.5	0.96	0.9	0.84	0.89	0.68	0.89	0.7	0.86	0
	F_score	0.81	0.7	0.95	0.5	0.98	0.95	0.88	0.94	0.68	0.94	0.78	0.92	0
	Accuracy	0.71	0.85	0.91	0.99	0.96	0.9	0.81	0.89	0.77	0.89	0.67	0.86	0
Support Vector Machine	Recall	0.88	0.8	0.99	0.5	1	1	0.96	0.98	0.75	0.97	0.87	0.97	0
	Precision	0.74	0.79	0.92	0.5	0.96	0.9	0.84	0.91	0.68	0.91	0.76	0.91	0
	F_score	0.81	0.79	0.95	0.5	0.98	0.95	0.89	0.93	0.71	0.93	0.81	0.94	0
	Accuracy	0.72	0.89	0.91	0.99	0.96	0.9	0.82	0.88	0.77	0.87	0.73	0.89	0
Naive Bayes	Recall	0.82	0.87	0.9	1	0.94	0.58	0.89	0.61	0.91	0.9	0.9	0.23	0.75
	Precision	0.78	0.43	0.93	0.06	0.97	0.95	0.87	0.93	0.43	0.92	0.72	1	0.12
	F_score	0.8	0.58	0.92	0.11	0.95	0.72	0.88	0.74	0.58	0.91	0.79	0.38	0.2
	Accuracy	0.72	0.67	0.85	0.85	0.91	0.59	0.81	0.63	0.52	0.84	0.7	0.33	0.51
Random Forest	Recall	0.88	0.81	0.99	0.5	1	0.97	0.93	0.95	0.78	0.97	0.84	0.97	0
	Precision	0.76	0.89	0.92	1	0.96	0.91	0.84	0.91	0.73	0.9	0.76	0.91	0
	F_score	0.82	0.84	0.95	0.67	0.98	0.94	0.89	0.93	0.76	0.94	0.81	0.94	0
	Accuracy	0.74	0.92	0.91	0.99	0.96	0.89	0.81	0.88	0.82	0.89	0.72	0.89	0
Logistic Regression	Recall	0.88	0.8	0.99	0	0.98	0.98	0.94	0.98	0.69	0.97	0.9	0.97	0
	Precision	0.76	0.79	0.92	0	0.96	0.9	0.84	0.91	0.7	0.91	0.77	0.91	0
	F_score	0.82	0.79	0.95	0	0.98	0.94	0.89	0.93	0.69	0.94	0.82	0.93	0
	Accuracy	0.74	0.89	0.91	0	0.95	0.89	0.81	0.87	0.78	0.89	0.74	0.89	0
Decision Trees	Recall	0.77	0.8	0.94	0.5	0.97	0.92	0.84	0.88	0.72	0.9	0.73	0.9	0.33
	Precision	0.77	0.82	0.93	0.5	0.96	0.93	0.84	0.9	0.72	0.91	0.73	0.93	0.4
	F_score	0.79	0.8	0.93	0.67	0.96	0.93	0.84	0.89	0.71	0.91	0.73	0.92	0.4
	Accuracy	0.71	0.9	0.88	0.99	0.93	0.87	0.76	0.81	0.8	0.84	0.65	0.85	0.9

Figure 4: Without Smote Results

Learners	Measures	synapse	xerces	tomcat	xalan	camel	prop	ant	arc	poi	ivy	velocity	redaktor	log4j
KNN	Recall	0.73	0.77	0.78	0.5	0.79	0.75	0.78	0.76	0.81	0.77	0.7	0.7	0.33
	Precision	**0.81**	0.54	0.95	0.05	**0.98**	0.93	**0.9**	**0.92**	0.59	**0.96**	0.77	0.86	0.08
	F_score	0.79	0.62	0.85	0.09	0.87	0.83	0.84	0.83	0.69	0.85	0.73	0.77	0.14
	Accuracy	0.72	0.77	0.76	0.85	0.78	0.73	0.77	0.73	0.73	0.76	0.65	0.65	0.52
Support Vector Machine	Recall	0.79	0.87	0.82	**0.67**	0.83	0.71	0.81	0.78	0.79	0.82	0.77	0.87	**0.67**
	Precision	0.8	0.67	**0.96**	0.2	0.97	0.93	0.89	**0.92**	0.65	0.94	0.79	0.9	0.14
	F_score	0.8	0.74	0.89	0.29	0.89	0.8	0.85	0.84	**0.71**	0.88	0.78	0.89	0.23
	Accuracy	**0.74**	0.85	0.81	0.96	0.81	0.69	0.78	0.74	0.76	0.8	**0.72**	0.81	0.69
Naive Bayes	Recall	0.79	0.84	0.88	**0.67**	0.83	0.36	0.84	0.54	**0.88**	0.86	0.87	0.3	**0.67**
	Precision	0.78	0.44	**0.96**	0.09	**0.98**	**0.94**	0.88	**0.92**	0.44	0.93	0.72	**0.93**	0.12
	F_score	0.79	0.59	0.91	0.16	0.9	0.53	0.86	0.69	0.59	0.89	0.79	0.46	0.21
	Accuracy	0.71	0.68	0.84	0.9	0.82	0.41	0.79	0.56	0.55	0.82	0.7	0.42	0.55
Random Forest	Recall	**0.91**	0.8	**0.94**	0.5	**0.95**	**0.93**	**0.91**	**0.9**	0.69	**0.89**	**0.9**	**0.93**	0.33
	Precision	0.76	**0.87**	0.94	**1**	0.97	0.92	0.85	0.91	**0.72**	0.92	0.74	0.9	0.2
	F_score	**0.82**	**0.84**	**0.93**	**0.67**	**0.96**	**0.93**	**0.88**	**0.9**	0.69	**0.91**	**0.81**	**0.92**	0.25
	Accuracy	**0.74**	**0.92**	**0.88**	**0.99**	**0.93**	**0.88**	**0.81**	**0.81**	**0.78**	**0.84**	**0.72**	**0.85**	**0.85**
Logistic Regression	Recall	0.79	**0.9**	0.82	0.5	0.83	0.71	0.8	0.78	0.75	0.84	0.77	0.83	**0.67**
	Precision	**0.81**	0.62	**0.96**	0.18	0.97	0.93	0.89	**0.92**	0.63	0.94	**0.81**	0.9	0.14
	F_score	0.8	0.72	0.89	0.29	0.89	0.8	0.84	0.85	0.69	0.89	0.77	0.87	0.23
	Accuracy	0.73	0.83	0.81	0.95	0.81	0.69	0.77	0.75	0.75	0.81	**0.72**	0.8	0.71
Decision Trees	Recall	0.85	0.77	0.89	0.5	0.94	0.9	0.88	0.85	0.56	0.84	0.83	0.9	0.33
	Precision	0.74	0.82	0.93	0.5	0.97	0.93	0.83	0.91	0.7	0.93	0.73	0.9	**0.22**
	F_score	0.79	0.79	0.91	**0.67**	0.95	0.92	0.85	0.88	0.62	0.88	0.78	0.9	**0.29**
	Accuracy	0.71	0.9	0.84	**0.99**	0.91	0.85	0.77	0.78	0.75	0.79	0.69	0.83	0.83

Figure 5: Smote Results

ghotra et al. found which are results without smote. But after applying smote, Random Forest outperformed every other learner. To control the high variance and most data sets have minority defective class, we highly recommend to use smoting. Comparing the run times and performance, we suggest to use Random Forest if the data sets are not big as it has larger runtime overhead.

VII. Future Work

In future, we can improve this pip package once other users start reporting issues. Additional features can be added such as:

- We can think of implementing cross project defect prediction.
- We can have binary classification, as well as to predict quantities of defects which is regression based model.
- The algorithm for smoting is hard-coded to "ball tree", this can be parametrized.
- The Split criteria and K value in K Nearest Neighbours are hard-coded, these can be parametrized.
- The learners currently doesn't support any tuning, which can implemented.
- Pretty visualizations can be added.

References

[1] A. Agrawal, W. Fu, and T. Menzies. What is wrong with topic modeling?(and how to fix it using search-based se). *arXiv preprint arXiv:1608.08176*, 2016.

[2] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *2009 20th International Symposium on Software Reliability Engineering*, pages 109–119. IEEE, 2009.

[3] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[5] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41. IEEE, 2010.

[6] W. Fu, T. Menzies, and X. Shen. Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76:135–146, 2016.

[7] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 789–800. IEEE Press, 2015.

[8] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. Using the support vector machine as a classification method for software defect prediction with static code metrics. In *International Conference on Engineering Applications of Neural Networks*, pages 223–234. Springer, 2009.

[9] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[10] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[11] M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 9. ACM, 2010.

[12] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[13] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[15] L. Pelayo and S. Dick. Applying novel resampling strategies to software defect prediction. In *NAFIPS 2007-2007 Annual Meeting of the North American Fuzzy Information Processing Society*, pages 69–72. IEEE, 2007.

[16] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[17] A. J. Scott and M. Knott. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, pages 507–512, 1974.

[18] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[19] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu. A general software defect-proneness prediction framework. *IEEE Transactions on Software Engineering*, 37(3):356–370, 2011.

[20] Q. Song, M. Shepperd, M. Cartwright, and C. Mair. Software defect association mining and defect correction effort prediction. *IEEE Transactions on Software Engineering*, 32(2):69–82, 2006.

[21] C. Tantithamthavorn, S. McIntosh, A. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 2015.

[22] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*, pages 321–332. ACM, 2016.

[23] S. Wang and X. Yao. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443, 2013.

[24] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540. ACM, 2008.