Choosing the Best Set of Features for a ML Model for Software Risk Prediction Based on Model Accuracy

Sugato Chakrabarty[1], Jhansi Rani Prathuri[2], Kavitha P[3], Richard Delwin Myloth[4], and Pavan Rajkumar Magesh[5]

[1,2]Professor
[3]Assoc. Professor
[4,5]Under Graduate Student
[1,2,3,4,5]Department of Computer Science and Engineering, CMR Institute of Technology, Bengaluru, India

August 29, 2020

Abstract

Trying to minimize the number of software failures is one of the most important goals in the area of software project management. In software development, which is a part of software project management, incremental changes are typically made to an existing set of code and documents that may be initially empty. This goal becomes more crucial due to short delivery schedules that constrain coding, inspection and testing. In this paper, we use an approach to determine an optimal or nearly optimal set of features to use in creating or training a machine learning model. We do this by creating subsets of a base set of features in this domain, training a ML model on the set of features and training them on the data set. We can then pick the set that gives the best results.

Keywords: Machine Learning, Software Engineering, Risk Prediction, Naïve Bayes, Neural Nets, Random Forest, Shapley

1 Introduction

Though software engineering is one of the most challenging and difficult of all engineering disciplines, this is often not recognized as such, because most of the software is hidden from our eyes in our daily lives.

Software development typically proceeds as a series of changes to a base set of software. Except for new projects, where the base set is initially empty, most

projects, however, involve incremental changes to an existing, perhaps large, set of code and documentation. Software developers usually make changes to the code by fixing defects or bugs, adding new functionality, improving performance or reliability, or rewriting the software to improve its changeability or readability, among other tasks. Each change carries with it some possibility of failure, leading in turn to possible software project failures. These failures are often a result of insufficient and ineffective risk information regarding the future. To overcome this, software risk prediction should be performed in advance to allow project managers insight into providing more valuable information for decision making, such as scope coverage, resource allocation, and schedule changes.

Reducing the number of software failures, thus, becomes one of the most challenging problems of software produc tion. It is even more important when tight delivery schedules restrict coding, inspection, and test ing intervals. In this paper, we specifically deal with one aspect of the above-mentioned scenario, namely predicting, using Machine Learning techniques, the risk associated with changing a specific software code segment given a set of parameters associated with that segment. Knowing this risk helps project managers to put in place possible risk mitigating tasks (e.g. increasing testing or rescheduling the launch of the software).

The remainder of this paper is organized as follows. In the section immediately below, we review related work. In the sections that follow, the methodology adopted and the results of running the Machine Learning models are shown. Finally, in the last section, the concluding remarks are presented.

2 Related Work

A number of investigations have been done that determine several product measures that predict the likelihood of a code having a fault.

A common approach is to use several product measures— determined from a short description of the code—as predictors of fault likelihood, with code size (that is, the number of lines of code) as one of the most important fault prediction measures.

The paper by An, Gustafson, and Melton [1], Basili and Perricone [2], and Hatton [3] connect the defect frequency of code to the file size. An, Gustafson, and Melton[1] also use the degree of nesting to predict the potential for a file to have a fault.

Measures of code complexity, such as McCabe's cyclomatic complexity [4] and Halstead's program volume [5] are other product measures that have been linked to failure possibilities. Empirical studies of product measures and fault rates have been conducted and are described by Schneidewind and Hoffman,[6] Ohlsson and Alberg,[7] Shen et al.,[8] and Munson and Khoshgoftaar.[9]

Another way to measure modeling fault rates uses data from the change and defect history of the program. Yu, Shen, and Dunsmore [10] and Graves et al.[11] use this idea for fault prediction, while Basili and Perricone [2]do a comparison based analysis between new code units and those that borrow code

from other places.

The software reliability literature contains various studies [12, 13, 14, 15, 16, 17] that roughly calculate the number of faults remaining in a software system. This may help in predicting the number of faults in the future.

Some studies have also been done with regard to software projects risk prediction [18, 19, 20]. In particular, research hasbeen conducted on prediction models, thatattempt to predict thefailure likelihood of components of a newly developed software system.Since a software system usually has components described by metrics such as the code complexity and number of pre-release changes, statistical models, such as a decision tree or logistic regression, may be used based on these metrics for predicting a component defect or system failure.

Gupta et al. [21] gives a quantitative model for risk control for a software project called SRAEM (Software Risk Assessment and Estimation Model). Much of the research conducted by industry involve using the data mining of software repositories (MSRs), such as DTSs and version control systems. In this context, Hassan[22] gives the concept of the entropy of changes, which is a measure of code change complexity. Kim et al.[23] propose the change classification technique, which involves learning historical change patterns of bugs, in order to predict whether a new code change will lead to bugs. Zimmermann et al. [24] predict defect proneness based on the defect information extracted from the CVS/SVN repositories. However, most of these techniques try to build a classification model to analyze the code or developer behavior and thereby predict whether the resulting code will have a bug.

Some work has also been made to automate the software testing process using Machine Learning (ML) which happens to be a sub domain of AI. In [25],[26] , evolutionary algorithms have been used to automate generation of test case. In [27], Artificial Neural Networks (ANN) have been used to build a model to determine the effectiveness of the generated test cases. Briand et al. in [28] has proposed a method based on the C4.5 decision tree algorithm to predict potential bugs in a software system. Research seems to suggest that machine learning techniques offer a promising approach to automate the testing processes.

3 Methodology

In this section the overall methodology is discussed. The basic idea is to create machine learning models to predict the risk of introducing a bug to a software code section, when making a change in it, and then compare these models in terms of several criteria.

We use supervised machine learning techniques and so there happen to be features used to create the machine learning model and certain steps that are typically followed in creating such a model. The steps are as follows:

1. Gathering the data: This step is very crucial as the quality and quantity of data gathered will directly determine how good the predictive model will turn out to be. In our case, the data happen to be software code sections from

github. An example of a software commit in git is shown in Figure 1.

2. Preparing the data: After the training data is gathered, the next step of machine learning needs to be done. This happens to be data preparation, where the data is loaded into a suitable place and then prepared for use in machine learning training. In addition to other tasks, the data now also has to be split into two parts. The first part that is used in training our model, will be the majority of the data set, usually around 80 percent (the training set) and the second will be used for evaluating the trained model's performance, usually around 20 percent (the test set).

In this step, the relevant features of the data set are also decided. This is important because choosing the appropriate set of features determines the accuracy of the output. For our work, the features chosen are subsets of the following features 1) lines of code added, 2) lines of code deleted, 3) number of files changed, 4) author experience 5) commit message categorization and 6) commit message length.

It ought to be mentioned here that the size of the code (e.g. the number of lines of code) is not used as a feature to consider since we are only interested in the effect of the changes made and not on the effect of the existing state of affairs.

We also decided to use the categorical variable called risk taking the value of 1 (i.e. equal to or greater than 50 percent probability of failure) and 0 (less than 50 percent probability of failure) as the output variable.

3. Choosing the model: The next step is to choose a model among the many machine learning models available. In our case, we've chosen the random forest model which yields the best performance according to previous studies (possible self citation)

4. Training the model: In this step, often considered to be perhaps the most important step, the model is trained or created with the training data. This essentially means that some kind of function is created based on which an input can be mapped to an output.Some models also need the user to adjust certain control parameters.

5. Evaluating the model: In this step,once training is complete, the model is evaluated for accuracy or goodness. This is where the test set put aside earlier comes into play. Evaluation allows the testing of the model against data that has never been seen and used for training and is meant to be representative of how the model might perform when used in the real world. There are several ways in which the accuracy of the model may be measured or evaluated. We will use the measures of Precision and Recall, Receiver Operating Characteristic or ROC, and Confusion Matrix to measure the accuracy of the models.

We have followed the above steps for four sets of features. These are lines of code added, lines of code deleted, number of files changed, author experience, commit message categorization, commit message length, lines of code added, number of files changed, author experience and lines of code added, lines of code deleted, number of files changed, author experience, commit message categorization, commit message length. We show the results for each of these three sets in the next section.

4 Results

In this section, the results obtained by running the dierent machine learning models based on the dierent sets of features are given in Table 1. An example of a commit has already been shown in Figure 1. Table 2 shows the performance of the ML models created based on the feature sets shown in Table 1, These performance values are found by taking the average values of the performances Nets and Naive Bayes. Based on the values in Table 2, it is clear that the accuracy of set 2 is about the same as that of set 1, while the accuracies of set 3 and set 4 are much lower.

Table 1: Feature Sets

Set Number	Features
1	lines added, lines deleted, no. files changed, author experience, commit categorization, message length
2	lines added, lines deleted, no. files changed
3	author experience, message length, commit categorization
4	lines added, no. of files changed, author experience

Table 2: Performance of feature sets

Set Number	Accuracy	Recall	Precision	F1 Score
Set 1	72.3	71.9	73.0	72.5
Set 2	69.9	62.2	74.2	67.7
Set 3	60.7	60.5	61.3	60.9
Set 4	58.3	59.5	58.7	59.1

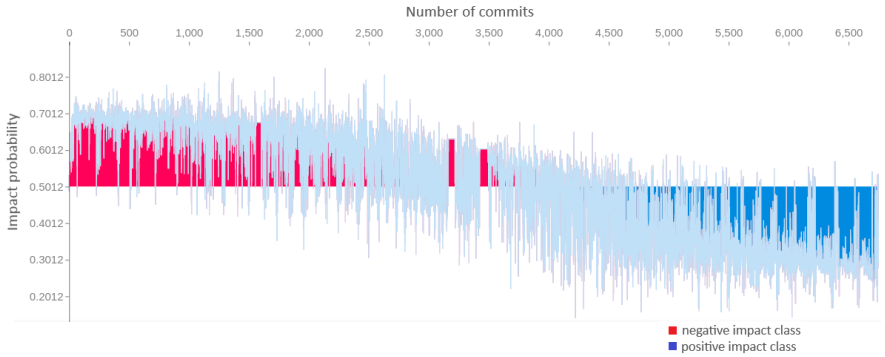

Figure 1: Impact of lines added

This agrees with the impacts of each of the features from the overall set (i.e. set number 1 in Table 1). These impacts are shown in Figure 1 to Figure

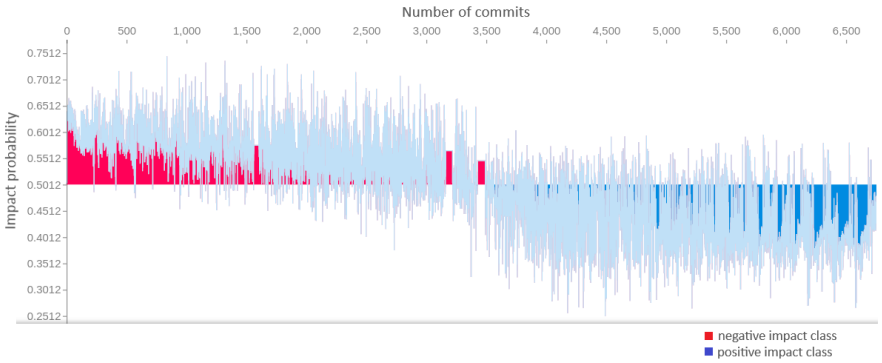

Figure 2: Impact of lines deleted

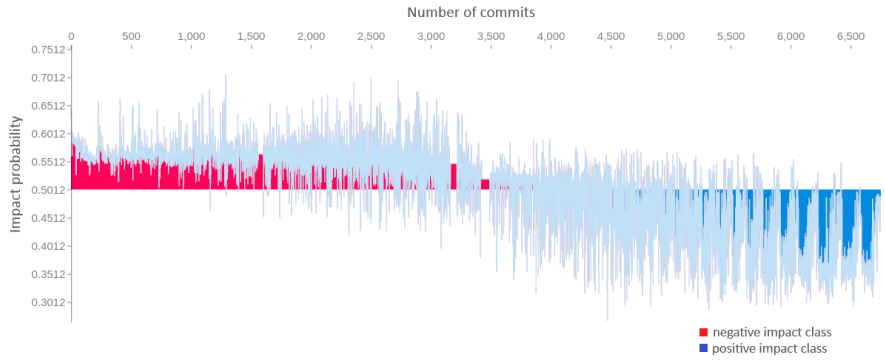

Figure 3: Impact of Files Changed

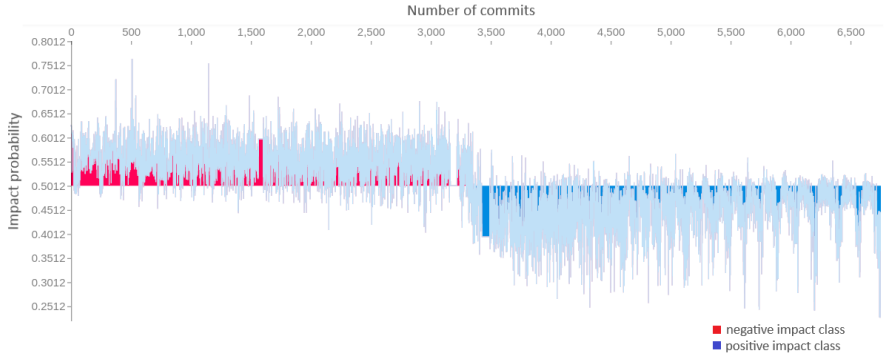

Figure 4: Impact of Message Length

6. Figures 1-6 show the impact of each feature on each sample of the entire data set with the samples plotted on x-axis while their predicted probability

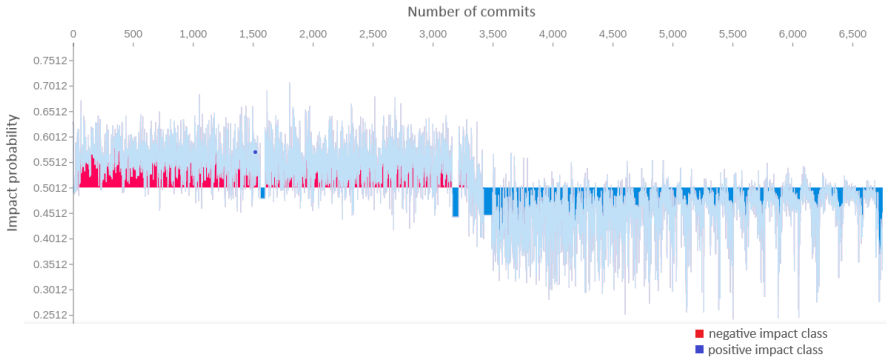

Figure 5: Impact of Author Experience

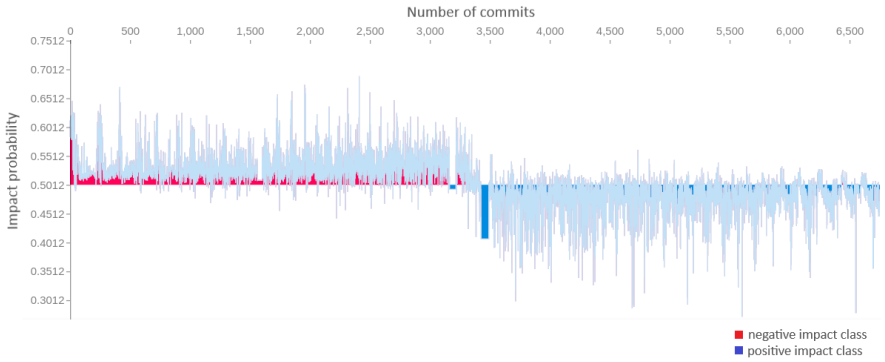

Figure 6: Impact of Commit Category

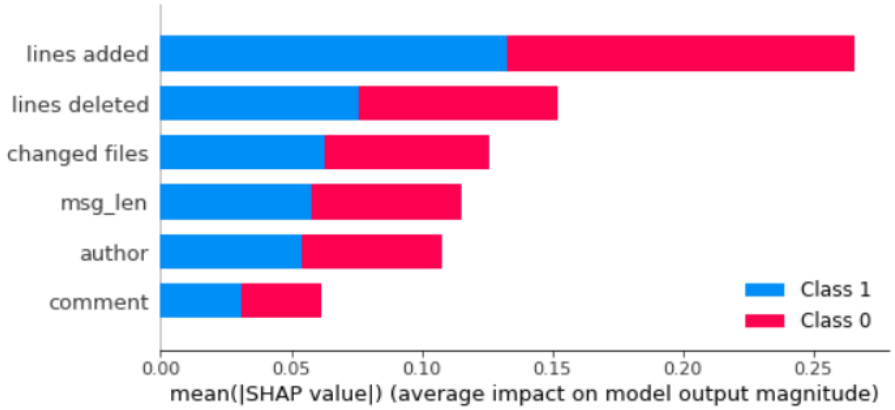

Figure 7: Overall impact analysis

is depicted on the y-axis. Red indicates a contribution to the positive class, while blue does the opposite. It is clear from these Figures that the red parts in Figures 1 and 2 are more than the red parts in the other Figures. Also, Figure 7 in the end depicts an overall view of each of the feature's impacts. Figures 1 to 7 are based on the SHAP (SHapley Additive exPlana- tions) that happens to be a game theoretic approach to explain the output of any machine learning model. It connects optimal credit allocation with local explanations based on the classic Shapley values [29], [30] from game theory and their extensions. Another approach could be to do an impact analysis of each of the features and then choose those which have maximum impact. One way to do this would be to decide a threshold n say on the number of features to be considered, sort the features based on their impacts and take the top n features based on these impacts.

5 Conclusions

In this paper, a way to determine an optimal or nearly optimal set of features to use in creating or training a machine learning model is given. This is done by creating subsets of a base set of features in this domain, training ML models on these sets of features and then picking the set that gives the most accurate results. Alternatively, an approach could be to do an impact analysis based on Shapley values of each of the features and then choose those which have maximum impact.

References

[1] K. H. An, D. A. Gustafson, and A. C. Melton, "A model for software maintenance," *IEEE Comp. Soc. Press, Proc. of the Conf. on Soft. Maintenance, Austin, Texas*, pp. 57–62, 1987, September.

[2] V. R. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation0," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[3] L. Hatton, "Reexamining the fault density component size connection," *IEEE software*, vol. 14, no. 2, pp. 89–97, 1997.

[4] T. McCabe, "A complexity measure ieee," *Transactions on software engineering, SE-2 (4)*, pp. 308–320, 1976.

[5] M. H. Halstead *et al.*, *Elements of software science*, vol. 7. Elsevier New York, 1977.

[6] N. F. Schneidewind and H.-M. Hoffmann, "An experiment in software error data collection and analysis," *IEEE Transactions on Software Engineering*, no. 3, pp. 276–286, 1979.

[7] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, 1996.

[8] V. Y. Shen, T.-J. Yu, S. M. Thebaut, and L. R. Paulsen, "Identifying error-prone software—an empirical study," *IEEE Transactions on Software Engineering*, no. 4, pp. 317–324, 1985.

[9] J. Munson and T. Khoshgoftaar, "Regression modelling of software quality: empirical investigation," *Information and Software Technology*, vol. 32, no. 2, pp. 106–114, 1990.

[10] T.-J. Yu, V. Y. Shen, and H. E. Dunsmore, "An analysis of several software defect models," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1261–1270, 1988.

[11] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on software engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[12] Z. Jelinski and P. Moranda, "Software reliability research," in *Statistical computer performance evaluation*, pp. 465–484, Elsevier, 1972.

[13] G. J. Schick and R. W. Wolverton, "An analysis of competing software reliability models," *IEEE Transactions on Software Engineering*, no. 2, pp. 104–120, 1978.

[14] S. N. Mohanty, "Models and measurements for quality assessment of software," *ACM Computing Surveys (CSUR)*, vol. 11, no. 3, pp. 251–275, 1979.

[15] S. G. Eick, C. R. Loader, M. D. Long, L. G. Votta, and S. Vander Wiel, "Estimating software fault content before coding," in *Proceedings of the 14th international conference on Software engineering*, pp. 59–65, 1992.

[16] D. A. Christenson and S. T. Huang, "Estimating the fault content of software using the fix-on-fix model," *Bell Labs Technical Journal*, vol. 1, no. 1, pp. 130–137, 1996.

[17] A. L. Goel, "Software reliability models: Assumptions, limitations, and applicability," *IEEE Transactions on software engineering*, no. 12, pp. 1411–1423, 1985.

[18] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE transactions on software engineering*, vol. 33, no. 1, pp. 2–13, 2006.

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

[20] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*, pp. 284–292, 2005.

[21] D. Gupta and M. Sadiq, "Software risk assessment and estimation model," in *2008 International Conference on Computer Science and Information Technology*, pp. 963–967, IEEE, 2008.

[22] A. E. Hassan, "Predicting faults using the complexity of code changes," in *2009 IEEE 31st international conference on software engineering*, pp. 78–88, IEEE, 2009.

[23] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[24] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pp. 9–9, IEEE, 2007.

[25] M. A. Ahmed and I. Hermadi, "Ga-based multiple paths test data generator," *Computers & Operations Research*, vol. 35, no. 10, pp. 3107–3124, 2008.

[26] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and software technology*, vol. 43, no. 14, pp. 841–854, 2001.

[27] A. von Mayrhauser, C. Anderson, and R. Mraz, "Using a neural network to predict test case effectiveness," in *1995 IEEE Aerospace Applications Conference. Proceedings*, vol. 2, pp. 77–91, IEEE, 1995.

[28] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*, pp. 137–146, IEEE, 2007.