Software Defect Prediction with Metaclassification Algorithms

Yannis Mentekidis*, Themistoklis Papavasileiou*, Themistoklis Diamantopoulos†, Andreas Symeonidis†
†Intelligent Systems and Software Engineering Laboratory
*†Department of Electrical and Computer Engineering
Aristotle University of Thessaloniki
Thessaloniki, Greece
*{mentekid,pvthemis}@ece.auth.gr †{asymeon,thdiaman}@issel.ee.auth.gr

***Abstract*—Predicting the reliability of software projects is an interesting and challenging problem in Software Engineering. The problem involves predicting the presence or absence of a logical fault ("bug") in a piece of software using Machine Learning methods. Current solutions found in literature propose several algorithms for software defect predictions. Although these solutions are effective in some cases, they are sensitive to the special characteristics of each dataset and are affected by noisy data, such as software metrics. In this paper, we design and implement an algorithm that combines the output of a number of other classifiers to produce more accurate predictions. The evaluation of our algorithm with respect to common practice indicates that it offers a more effective alternative.**

***Keywords*—Software Reliability Prediction, Software Defect Prediction, Ensemble Classification Algorithms, Machine Learning**

I. Introduction

Software defect prediction has been an area of interest in the software engineering community for a considerable amount of time. Manpower allocation to test for or fix a bug constitutes a cost in terms of both monetary and time resources. In adition, testing typically consumes 40% to 50% of development effort [1]. Any reduction of that effort that eventually leads in a reduction of total cost is therefore sought after.

Many software metrics have been proposed for use with Machine Learning algorithms to solve the problem of Software Reliability Prediction. A prominent set of metrics is proposed by Chidamber and Kemerer (CK) [2]. These metrics characterize the cohesion and connections between classes in an Object-Oriented design. In a similar approach, D'Ambros *et al.* [3] propose the combination of these metrics with additional "OO Metrics" which characterize the software classes themselves.

The combination of these two groups of metrics, often called "CK+OO Metrics", has been used extensively in literature to predict the presence or absence of a bug in software projects. The problem with these methods is that they are succeptible to noisy data, and produce suboptimal results. For different software projects, different algorithms can be found to be more effective.

In this paper, we posit that combining the predictions of an ensemble of algorithms will outperform any of the singular approaches found in literature. We test different ensemble architectures and different aggregation algorithms. To compare our results to the state-of-the-art, we use the Eclipse JDT Core[1] dataset [3].

The rest of the paper is structured as follows. In Section II, we discuss related work done in the field of Software Reliability Prediction. In Section III, we propose the method of *metaclassification*, and in Section IV test metaclassification against other proposed methods presented in Section II. Section V concludes this paper by discussing insights on the results and presenting possible future research topics.

II. Related Work

The design of prediction models consists of two parts: (a) the dataset to be used, and (b) the algorithm used to fit its parameters to a dataset's features.

In Software Reliability Prediction, the origin of features has been an active field of research. While some argue that the changing rate of a component can predict more accurately the presence or absence of defects [4], [5], [6], others focus on the current state of the source code of the program, and require no history to predict whether or not a defect exists, instead using a variety of metrics as features [7], [8], [9].

Each approach defines metrics that capture the behaviour of a program according to a certain rationale, for example temporal or object-oriented metrics. In addition to the CK+OO metrics [3] discussed earlier, there exist predictors measuring changes in code [6] or the entropy of changes, hinting that complex changes are more error-prone than simple ones [10]. For different software projects, different sets of measures have been found to have stronger correlation with the presence or absence of bugs. As an example, the bugs in the JDT Core dataset, which we use for the experimental validation of our method, have been found to have a stronger correlation with the CK+OO metrics [3].

The problem of defect prediction can be regarded as either a regression or a classification problem. In the scope

[1] https://eclipse.org/jdt/core/

of this work, we view the problem as one of classification.

In this context, Naive Bayes [11] has been used on public domain datasets to detect complex components in the source code. Authors claim these components are more defect-prone than simpler ones, therefore making bug identification simpler. Logistic Regression has been found to be effective [7] in the Eclipse dataset, mapping failures to components, with experimental results suggesting that the complexity of code is a good predictor of defect presence. Decision Trees have also been applied to the problem [6], using a dataset extracted from Mozilla releases [12]. Genetic Fuzzy rule-based systems have also been employed, achieving not only effective classification in the Eclipse dataset [13], but also increasing interpretability of the results.

While these methods provide us with satisfactory results, they are often overfitted to the corresponding dataset and display high variance even when they only consider a fixed set of metrics [14]. As a consequence, one must take several decisions regarding the choice of algorithm in combination with the choice of metrics, resulting in time-consuming experimentation.

Thus, a collection of singular, diverse approaches entails a tradeoff, since different algorithms might make inaccurate predictions for different parts of the dataset. However, by combining the capabilities of these algorithms, we can avoid this tradeoff and achieve higher accuracy. In the following section, we design an ensemble algorithm that effectively confronts the aforementioned issues.

III. Metaclassifier Algorithms

A. Definitions

For the sake of clarity, we define the symbols used for the entirety of this Section.

For a training set, we define as N its cardinality (number of samples) and d the dimensionality. We denote as y a vector of size $N \times 1$ that contains class labels for each of the N training examples.

We call a group of algorithms an *ensemble* and denote by n the number of algorithms contained in an ensemble.

Finally, for a testing set, we define its cardinality as N'. The vector of class predictions made using the $N' \times d$ dataset is then called y'.

B. Method

The proposed method uses an algorithm ensemble. Each of the algorithms makes predictions on the same set of training examples. The metaclassification process then uses the results of these predictions, called the meta-dataset, instead of the actual dataset to make predictions.

To produce the meta-dataset, each participating algorithm in the ensemble is trained with part of the available data, and makes predictions on the rest of the dataset. By aggregating these results, we create a meta-dataset that has, as its features, the classification decision of each of n classifiers. The process is described in Figure 1.

To increase the cardinality of the meta-dataset, we use k-fold cross-validation[2] for each of the classifiers in the ensemble. For the purposes of this paper, we have picked k=10. This way, each algorithm is repeatedly trained on a different 90% of the data and makes predictions on the remaining 10%. Cross-validation ensures that all available data is used once as a testing set in a valid way, i.e, predictions are made on data that have not been used as training examples in the current fold. This is therefore an unbiased way to produce a larger meta-dataset without invalidating the results.

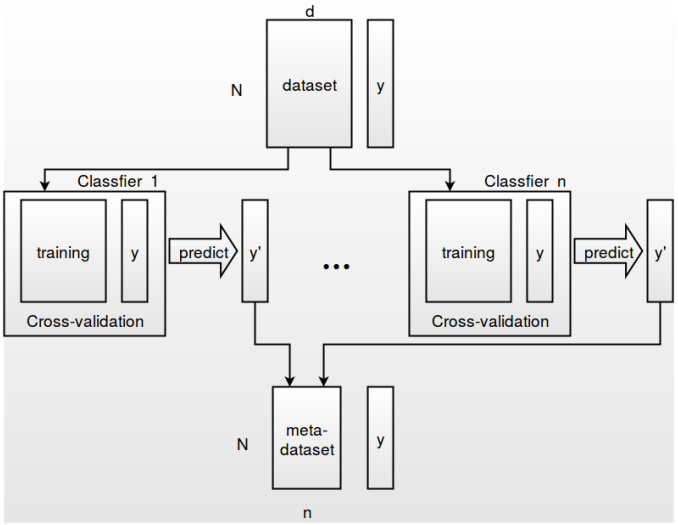

Fig. 1. Meta-dataset Creation Process

After the results of the ensemble have been collected, we train a classification algorithm (called *aggregator*) to learn from those results. This algorithm does not use the initial data to be trained, instead it only uses the meta-dataset produced earlier. For an ensemble of size n and an initial dataset of cardinality N, the meta-dataset will be of size $N \times n$, if cross-validation is used.

In its simplest form, the aggregator could be a typical majority vote scheme. In that case, for each example of new data, we would have n predictions by n algorithms. We would then pick the class that was voted for by the majority of the algorithms. If a voting scheme was used, the creation of the meta-dataset and the ensemble's training would not be necessary, since majority vote does not require fitting any parameters.

The method we propose uses different forms of aggregator algorithms instead of voting. We use classification algorithms as aggregators, since these algorithms successfully distinguish the features of the meta-dataset that contain useful information about the output. Thus,

[2]In k-fold cross-validation, a dataset is split into k equal parts. In each of k rounds, an algorithm is trained using k-1 of these parts, and then makes predictions using the remaining 1 part. In the end of the k rounds, cross-validation ensures that each data entry has been used exactly once in a test set, and k-1 times as part of a training set. Cross-validation is the alternative to the hold-out method, in which a fixed part of the data is used as a training set and the rest of the data as a test set.

the aggregator can distinguish between more and less successful algorithms for the task at hand. As an example, in the case of Logistic Regression, different weights are assigned to different algorithms of the ensemble. These weights are determined during the training phase based on the performance of each algorithm, and can therefore minimize or maximize the algorithm's influence of the final decision. A similar method has been presented in [15], where it is found that Logistic Regression outperforms the voting method for the Higgs dataset.

Algorithm 1 describes the process of creating the meta-dataset. The algorithm receives as input the training set and corresponding class labels, as well as n untrained classifiers. Each classifier is trained 10 times on different subsets of the dataset using 10-fold cross-validation. The N predictions the classifier made are then stored as a new column of the metadataset, which is returned after all n algorithms have been trained this way.

Algorithm 2 describes how the aggregator is trained. The aggregator uses Algorithm 1 to produce the meta-dataset from the initial data, and then uses the meta-dataset to learn the required parameters. Effectively, the aggregator will fit its parameters based on the predictions made by each classifier in the ensemble.

Algorithm 3 describes how the metaclassifier makes predictions on new data. The metaclassifier uses the trained ensemble to make predictions, thus creating a new, unlabeled meta-dataset. The aggregator is then used to predict the labels for the N′ new samples, using the model it learned when trained with algorithm 2.

At first sight, the proposed method is time-consuming: Each of the algorithms in the ensemble has to be trained 10 times to produce the meta-dataset, which will take a considerable amount of time. However, there are two advantages that make this method useful in practice.

At first, note that metaclassification only needs a handful of training algorithms to work, an assumption confirmed by our experimental evaluation of our method (see Section IV). Furthermore, the proposed method is completely task-parallel for up to n cores, meaning the individual algorithms can be trained in different processing nodes or cores, each using a copy of the initial dataset, requiring minimal cross-core communication. The lack of need of communication is important, making a parallel version of the proposed method trivial to design. This feature makes metaclassification schemes very useful when dealing with very large datasets in distributed environments.

Algorithm 1: Meta-dataset creation algorithm

Input : An $N \times d$ dataset, an $N \times 1$ column with true labels, and n classification algorithms
Output: An $N \times n$ meta-dataset

```
for Each algorithm in the ensemble do
    for Each of k folds of cross-validation do
        Train the algorithm with this fold's training subset;
        Classify the remaining data;
    end
    Store the N classification results in a new column of the meta-dataset;
end
```

Algorithm 2: Metaclassifier Training

Input : An $N \times d$ dataset, an $N \times 1$ column with true labels, and n classification algorithms
Result: The aggregator is trained

1 Use Algorithm 1 to create a meta-dataset from the initial dataset;
2 Use the $N \times n$ size meta-dataset and true labels to train the *aggregator*;

IV. Results

A. Performance Indicators

The dataset we use to test our model on is the Eclipse JDT Core[16]. It contains the CK+OO metrics (17 features) as well as bug tracking information (class label) for 997 classes of the Eclipse project. Of these, approximately 80% were found to be bug-free (class 0) and the remaining 20% was found to contain bugs (class 1). Dealing with an unbalanced dataset, it follows that its partitions, namely the training and testing set will not be balanced as well. To address this issue, we apply the SMOTE oversampling technique [17] on the training set to achieve balance in relation to the bug existence in the records. The testing set is left as is.

In an unbalanced dataset, high accuracy does not necessarily indicate a successful model. A naive approach of classifying each example as bug-free scores an 80% accuracy, but this is obviously an ineffective model. Therefore, we consider a prediction model as acceptable if its accuracy is greater than that of the naive approach, and further assess the acceptable models according to their precision. Consequently, we seek to keep a model's accuracy above 80% while discovering as many bugs as possible.

For the aforementioned reasons, we employ the *accuracy* and *precision* metrics. Accuracy measures the ratio of correctly classified observations, or more specifically the sum of true positive and true negative values over the total number of samples. Precision concerns the ratio of correctly detected defects (true positives) and is calculated as the number of true positives divided by the sum of true positives and false negatives. In our dataset, precision is

Algorithm 3: Metaclassifier prediction algorithm

Input : $N' \times d$ new data, the same n classification algorithms
Output: An $N' \times 1$ column with class predictions for the new data

```
for Each algorithm in the ensemble do
    Make predictions for the N' × d new data;
    Store predictions in one of n N' × 1 columns;
end
```

5 Use the $N' \times n$ new dataset to predict the label for the N' new examples;

maximized if all of the bugs are detected.

B. Experimental Results

To demonstrate the efficiency of the proposed method, we conduct three experiments and present their results. First, we need to demonstrate how the consistency of the ensemble affects the results, i.e. to analyze how the number of classifiers influences the effectiveness of the method. Second, it is important to demonstrate the effect that a different choice of aggregator would have on performance. Last, we compare the results of the proposed method with the results of other known literature methods.

The experiments were conducted using Python's SciPy libraries [18], which include a number of algorithms and a framework for creating new ones. We have used the provided algorithms and have implemented the proposed metaclassifier ourselves[3].

In the following paragraphs, we present the results of three experiments. During the first two experiments, we try different versions of the proposed model to assess sets of parameters with superior performance. To do that objectively, we use 10-Fold cross-validation on the data to ensure that the evaluation is objective. We repeat the test of each parameter set 30 times, and show the resulting precision and accuracy as boxplots describing the 1st, 2nd and 3rd Quartile of the data. In this process, we use the first 799 entries of the Eclipse Source Code Metrics dataset.

Having chosen the most effective set of parameters, we can now proceed to evaluating the proposed method against other methods suitable for the task. In this part, we will use the previous 799 entries for training our models, and the remaining 198 entries of the dataset to evaluate them. Using a different set of data than the one used to fit parameters is essential to producing results that are not the product of overfitting a model to the dataset.

1) Ensembles: In the first experiment, we evaluate the performance of the Metaclassifier using different ensembles. We have devised 5 different ensembles, each with different characteristics. The "Big" ensemble contains 15 classifiers, including different versions of kNN, Decision Trees, SVMs and other classifiers. The "Small" ensemble contains only 6 classifiers which we have observed to have good results on the problem, while the "Good & Bad" ensemble adds to those another 3 classifiers which we have found to not perform adequately to the task. Finally, ensemble "Bad" only contains these 3 classifiers, along with another one we have also found to perform poorly, and ensemble "No Bias" only contains classifiers which have a 1:1 class weight for positive and negative predictions, i.e. the class imbalance is not handled. Table I describes which classifiers can be found in each ensemble.

Figures 2 and 3 depict the values of accuracy and precision respectively for the five ensemble configurations. As we can see from both figures, the results are encouraging as we have achieved the bare minimum requirement of retaining an adequately high accuracy. Indeed median accuracy does not drop below 80% for any of the ensembles.

[3]https://github.com/mentekid/Metaclassifier

TABLE I. Ensemble Architecture

	Big	Small	Good & Bad	Bad	No Bias
12NN, Chebyshev	•	•	•		•
9NN, Canbera	•	•	•		•
Gini Tree	•				•
Entropy Tree	•	•	•		•
RBF SVM	•	•	•		•
Naive Bayes	•		•	•	•
Random Forest, Gini	•				•
Random Forest, Entropy	•	•	•		•
Logistic Regression	•		•	•	•
Gini Tree, 1:10	•			•	
Entropy Tree, 1:10	•				
RBF SVM, 1:10	•				
RF, Gini, 1:10	•				
RF, Entropy, 1:10	•	•	•		
Log Regression, 1:10	•		•	•	

Furthermore one can also observe that all ensembles fare almost equally; this is a useful indication that implies that the overhead computation can be reduced significantly by limiting the ensemble to classifiers that are known to be good at solving the given problem.

Interesting conclusions can also be drawn from the "Bad" ensemble, which has performed only slightly worse than the rest. As indicated by these results, our method can even boost poorly performing algorithms such as the ones included in this ensemble, significantly raising their accuracy. This means that one could use the proposed method to improve the results in a problem where no adequate classification algorithm has been developed.

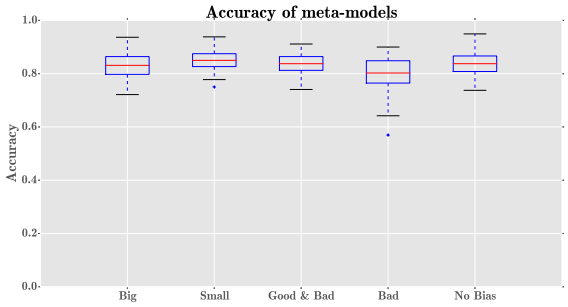

Fig. 2. Experimental accuracy of different ensembles

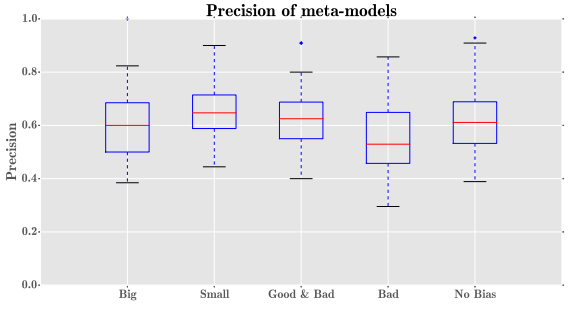

Fig. 3. Experimental precision of different ensembles

2) Aggregators: The second experiment attempts to specify which classification algorithms are good candidates for the role of the aggregator. In this part, we also test the naive method of simple majority vote, and observe its shortcomings. For this experiment, we use the "Small" ensemble described in Table I and assess the following aggregators for the Model: Logistic Regression, Gaussian Naive Bayes, Decision Tree, k-Nearest Neighbors (kNN), SVM and Majority Vote.

Concerning accuracy, the differences among the algorithms are almost insignificant. The selection of aggregator does not greatly affect the accuracy of the metamodel. It can be observed in Figure 4 that a Gaussian Naive Bayes aggregator fares slightly better than the competition, having SVM as a close second. kNN (biased 2:1 towards the negative class) and the voting scheme closely follow. All 6 tested models achieve an accuracy of over 80%, so we can't disqualify any.

It is noteworthy, though, that having similar accuracy does not necessarily imply similar precision. As shown in Figure 5, which depicts the precision for the 6 models, the last four aggregators have a better median performance than either Logistic Regression or Gaussian Naive Bayes. Of these choices, we will retain the Support Vector Machine aggregator as our choice, since it appears to both achieve high precision scores in several runs as well as not drop as low as Decision Trees and kNN in its worst runs.

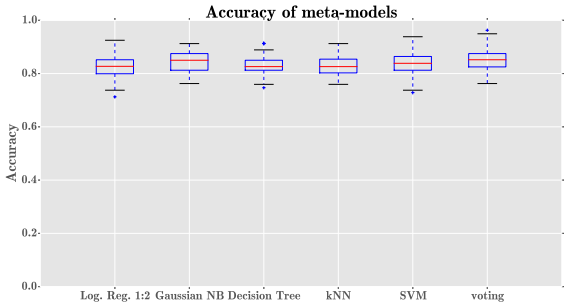

Fig. 4. Experimental accuracy of different aggregators

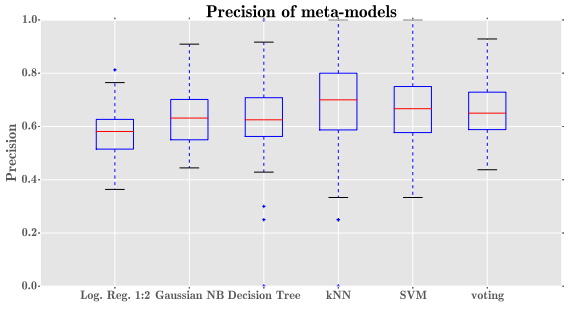

Fig. 5. Experimental precision of different aggregators

3) Other Algorithms: Finally, we are interested in comparing our results against other algorithms that are frequently used to approach this problem. Based on our previous experiments, we have picked the 'Small' ensemble and paired it with the Support Vector Machine aggregator. This is the combination of parameters that seem to produce the highest chance of discovering bugs, and therefore the highest Precision score, while maintaining a satisfactory Accuracy.

We evaluate our algorithm against 4 other classifiers: a kNN classifier, a Decision Tree, a Gaussian Naive Bayes, and an SVM. The selection of these algorithms covers approaches and methods used by current literature, such as the ones discussed in Section II.

Concerning accuracy, in Figure 6 we can observe that the Metaclassifier correctly classifies 83% of the dataset, faring equally with the kNN method.

In terms of precision, however, it is obvious that the proposed method outperforms the kNN as well as the SVM and Naive Bayes classifiers. Furthermore, our method performs better than Decision Trees, without compromising accuracy.

In Figure 7 we observe the much better performance of the proposed method than that of kNN and SVM. Metaclassification also performs slightly better than the Decision Tree and Naive Bayes with respect to Precision.

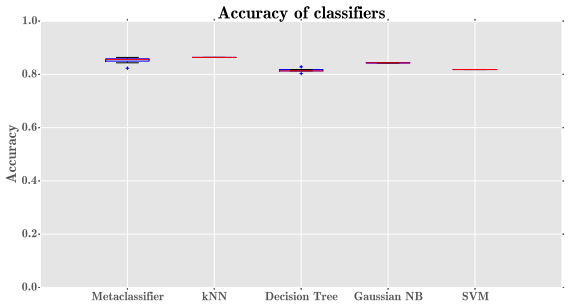

Fig. 6. Experimental accuracy of different classifiers

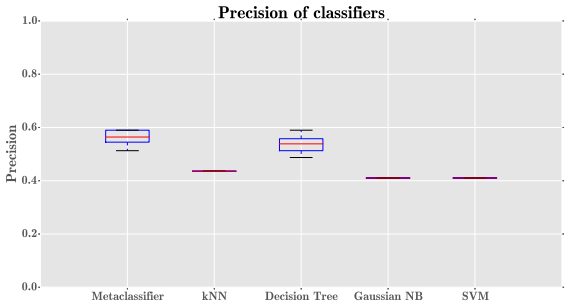

Fig. 7. Experimental precision of different classifiers

V. Conclusions

Current literature on software defect prediction has led to the development of several methods for predicting defects in software using source code metrics. These methods, however, are sometimes sensitive to specific datasets. Additionally, they do not fully exploit the underlying

relations of these metrics, thus their accuracy and precision levels can be further improved.

In this work, we have proposed a new classification algorithm, called metaclassification, which uses the results of different classification algorithms trained on a problem, to produce more accurate results. Upon evaluating our method, we conclude that it is robust and produces stable results for a number of different ensembles. Our method is also quite efficient with respect to ensemble methods, since it requires a small number of models. The effectiveness of our method both in terms of precision and in terms of accuracy was empirically supported using the CK+OO metrics of the Eclipse JDT Core dataset.

The presented idea can be explored further. An interesting question is whether we can further aid the aggregator by including, for each prediction, the confidence level of the algorithm making the prediction. In the future, we plan to include this option in the provided source code. Another area to be explored is the amount of classifiers required for the algorithm to efficiently work. By further experimentation we can pinpoint a number of classifiers for which the proposed metaclassification scheme produces diminishing returns, or a lower bound for which the metaclassifier behaves no better than other algorithms. In the future, we also plan to apply the metaclassification method to other similar datasets, and see if it outperforms other well-known approaches.

References

[1] B. Beizer, *Software Testing Techniques (2Nd Ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[2] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, Jun 1994.

[3] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Engg.*, vol. 17, no. 4-5, pp. 531–577, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9173-9

[4] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, "Change bursts as defect predictors," Nov 2010, pp. 309–318.

[5] T. Khoshgoftaar, E. Allen, N. Goel, A. Nandi, and J. McMullan, in *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, "Detection of software modules with high debug code churn in a very large legacy system," Oct 1996, pp. 364–371.

[6] R. Moser, W. Pedrycz, and G. Succi, in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08."A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," New York, NY, USA: ACM, 2008, pp. 181–190. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368114

[7] T. Zimmermann, R. Premraj, and A. Zeller, in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '07."Predicting defects for eclipse," Washington, DC, USA: IEEE Computer Society, 2007, pp. 9–. [Online]. Available: http://dx.doi.org/10.1109/PROMISE.2007.10

[8] N. Nagappan, T. Ball, and A. Zeller, in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. "Mining metrics to predict component failures," New York, NY, USA: ACM, 2006, pp. 452–461. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134349

[9] K. E. Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *J. Syst. Softw.*, vol. 56, no. 1, pp. 63–75, Feb. 2001. [Online]. Available: http://dx.doi.org/10.1016/S0164-1212(00)00086-8

[10] A. Hassan, "Predicting faults using the complexity of code changes," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, May 2009, pp. 78–88.

[11] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.10

[12] P. Knab, M. Pinzger, and A. Bernstein, in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06, "Predicting defect densities in source code files with decision tree learners," New York, NY, USA: ACM, 2006, pp. 119–125. [Online]. Available: http://doi.acm.org/10.1145/1137983.1138012

[13] T. Diamantopoulos and A. Symeonidis, in *Proceedings of the Fourth International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, ser. RAISE '15. "Towards interpretable defect-prone component analysis using genetic fuzzy systems," Piscataway, NJ, USA: IEEE Press, 2015, pp. 32–38. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820668.2820677

[14] Y. Suresh, L. Kumar, and S. K. Rath, "Statistical and machine learning methods for software fault prediction using ck metric suite: A comparative analysis," *ISRN Software Engineering*, 2014.

[15] I. Arnaldo, K. Veeramachaneni, A. Song, and U.-M. O'Reilly, "Bring your own learner: A cloud-based, data-parallel commons for machine learning," *Computational Intelligence Magazine, IEEE*, vol. 10, no. 1, pp. 20–32, Feb 2015.

[16] M. D'Ambros, M. Lanza, and R. Robbes, in *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)* "An extensive comparison of bug prediction approaches,". IEEE CS Press, 2010, pp. 31 – 41.

[17] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Int. Res.*, vol. 16, no. 1, pp. 321–357, Jun. 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=1622407.1622416

[18] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–, [Online; accessed 2016-01-27]. [Online]. Available: http://www.scipy.org/