Assessing Ensemble Learning Techniques in Bug Prediction

Zsolt János Szamosvölgyi$^{1[0000-0002-4216-1705]}$, Endre Tamás Váradi$^{1[0000-0001-8274-8992]}$, Zoltán Tóth$^{1[0000-0002-2268-1877]}$, Judit Jász$^{1,2[0000-0001-6176-9401]}$, and Rudolf Ferenc$^{1[0000-0001-8897-7403]}$

[1] University of Szeged, Hungary
[2] FrontEndART Ltd., Hungary
{szamos,idarav,zizo,jasy,ferenc}@inf.u-szeged.hu

Abstract. The application of ensemble learning techniques is continuously increasing, since they have proven to be superior over traditional machine learning techniques in various domains. These algorithms could be employed for bug prediction purposes as well. Existing studies investigated the performance of ensemble learning techniques only for PROMISE and the NASA MDP public datasets; however, it is important to evaluate the ensemble learning techniques on additional public datasets in order to test the generalizability of the techniques. We investigated the performance of the two most widely-used ensemble learning techniques AdaBoost and Bagging on the Unified Bug Dataset, which encapsulates 3 class level public bug datasets in a uniformed format with a common set of software product metrics used as predictors. Additionally, we investigated the effect of using 3 different resampling techniques on the dataset. Finally, we studied the performance of using Decision Tree and Naïve Bayes as the weak learners in the ensemble learning. We also fine tuned the parameters of the weak learners to have the best possible end results.
We experienced that AdaBoost with Decision Tree weak learner outperformed other configurations. We could achieve 54.61% F-measure value (81.96% Accuracy, 50.92% Precision, 58.90% Recall) with the configuration of 300 estimators and 0.05 learning rate. Based on the needs, one can apply RUS resampling to get a recall value up to 75.14% (of course losing precision at the same time).

Keywords: AdaBoost · Bug Prediction · Resampling · Unified Bug Dataset

1 Introduction

As bugs having a high cost, especially in later stages of the development of a software, it is crucial to reveal and eliminate as many of them as we can early in order to keep the maintenance costs low [4]. Evidently, this fact implies defect prediction being one of the most active research area [16, 25].

Recently, ensemble learning techniques have gained attention as they have performed better than traditional machine learning approaches in various domains covering a broad-spectrum [20]. Not surprisingly, ensemble learning techniques have been applied for defect prediction as well [12]. AdaBoost [23] is one of the most widely adopted ensemble learning method for bug prediction [12, 21, 27], where different base learning algorithms were investigated, such as Naïve Bayes, Logistic Regression, Multi-Layer Perceptron, Support Vector Machine, Decision Tree and more [18, 28].

Machine learning algorithms take the input as a series of feature vectors, which means that one has to produce these numerical values for each entry (usually files, classes or methods). Hence, researchers tend to reuse existing datasets in order to reduce the amount of work to be done and increase the reproducibility of their approaches. One common form of input data is where entries are described with software product metrics such as in the PROMISE [22] or the NASA MDP bug datasets [19]. These datasets were used by numerous studies [12, 27, 28]. In our approach, we used a different dataset, namely the Unified Bug Dataset [10], to investigate whether the achieved results hold in general. The Unified Bug Dataset contains bug related information for class level entries, amongst others. The dataset brings different software product metric-based datasets together, such as the PROMISE dataset [22], Bug Prediction dataset [7], and the GitHub Bug Dataset [26].

In this study, we investigate whether AdaBoost is superior over Bagging, both applying decision tree and Naïve Bayes algorithms as their weak learners. Related papers investigated parameter tuning only in a limited fashion or not experimented with at all. We ran a fine-grained search for finding the best parameter setup which includes tuning *n estimators*, *learning rate*, *max depth*, *min samples leaf*, and the *criterion* (gini or entropy). Note that the latter 3 are parameters for decision trees only.

Since bug datasets usually suffer from imbalance in their data, we also investigated the effect of using various resampling methods. We employed *SMOTE* [5], *RUS* [13], and a custom one used in the Deep Water Framework [11].

Based on the above mentioned aspects and deficiencies, we composed the following research question to be answered in this paper:

RQ 1: Does AdaBoost performs better than other classifier methods for bug prediction?
RQ 2: Is there any resample technique which performs consistently better than others?
RQ 3: Which is the best weak learning algorithm and which parameter configuration is the most powerful?

The rest of the paper is organized as follows. In Section 2, we enumerate the related papers, then we show the tools and techniques in details of which our approach consists in Section 3. Next, we evaluate our approach and answer our research questions in Section 4. The threats to validity are listed in Section 5. Finally, Section 6 concludes the paper and gives future work directions.

2 Related Work

In this section, we present the most related works to our study.

As Catal et al. showed [3], the most widely used machine learning algorithms for bug prediction are Logistic Regression, Naïve Bayes, Decision Tree, and Random Forest. Recently, ensemble learning techniques have started to be adopted in the context of bug prediction. Nevendra and Singh showed that AdaBoost with Extra Tree base learner could improve the performance of bug prediction [17], but this technique focuses on the bug count (regression) instead of binary classification.

AdaBoost was also used to predict defects in an imbalanced dataset [12, 21, 27]. Gao and Yang tried to use Back Propagation Neural Network to fight the imbalance in the data. We rather applied Decision Tree and Naïve Bayes base learners and investigated whether a resampling technique can help achieving better performance in the end. The work of Wang and Yao [27] is more close to ours as they investigated the usability of various resampling techniques. Beside AdaBoost, Bagging is also used as an ensemble learning technique [14], but these were not studied together in these papers. This missing comparison was provided by Peng et al. [18] by using Analytic Hierarchy Process (AHP) to rank the quality of ensemble methods. They found that AdaBoost is the best method, with the base learners of KNN, C4.5 decision tree and Naïve Bayes. Later, Khan concluded similar results [15] when used a hybrid ensemble approach to predict bugs. However, in their studies, they did not report any method to be applied in order to decrease imbalance in their data. We evaluated both AdaBoost and Bagging as an ensemble learner, with an additional resampling technique.

In a recent study, Yucalar et al. provided an exhaustive comparison of machine learning methods that are included in the popular WEKA machine learning framework [28]. They used numerous algorithms to form a baseline to which they compared the ensemble learning techniques. They found Rotation Forest (ROF), Random Forest, Logic Boost, Adaboost and Voting as the best fault predictors in terms of F-Measure and Area Under Curve (AUC). However, they did not consider the parameters of the base learners, only the number of learners used by the ensemble learners. In this study, we have tried to fine tune the parameters of the algorithms as well.

Ensemble learning techniques were also trialed in a narrowed domain of bug prediction, namely, in Aging Related Bugs prediction [24]. The study focuses on bagging, boosting, and stacking ensemble techniques which were justified to be effective in this narrowed domain as well. We stick with the general approach and evaluate ensemble learning methods on a wider range of datasets.

Most of the studies related to bug prediction; especially the ones using ensemble learning, are evaluated their approach on the PROMISE [22] and the NASA MDP dataset. However, it was shown that one should use these datasets with precautions [19]. Using only one dataset, which was constructed with one selected method, also makes the bug prediction techniques built on top of it more sensible. This generalizability threat should be eliminated by involving additional public bug datasets. The Unified Bug Dataset was previously used successfully in bug prediction [8, 10], which gives an extra opportunity to test the ensemble learning techniques on.

3 Approach

3.1 Dataset

In order to test the AdaBoost and Bagging classifiers in fault prediction, a large and representative dataset is necessary. Our choice to measure the viability of ensemble learning classifiers is the class-level part of the Unified Bug Dataset [10], which contains 60 different metrics for every 47,618 classes. The Unified Bug Dataset is an integration of 3 well-known and widely-used datasets (namely, PROMISE [22], the Bug Prediction Dataset [7] and the GitHub Bug Dataset [26]). An entry in the dataset contains 60 different numeric metrics similar to the experiments of [8] (from the simplest LOC metrics to the more complex complexity metrics) and the number of bugs that were determined for the actual class. The features which is used by the classification process is calculated by the OpenStaticAnalyzer toolset [1]. The number of bug occurrences are transferred from the original datasets, which basically means that the bug proneness of the entries come from three different approaches.

There are projects that occur multiple times in different versions in the unified dataset. Using these kinds of datasets arises the question of whether it is a problem in our approach of using machine learning techniques for fault prediction? We did not treat this as a problem, because the whole approach relies on the calculated metrics and our goal is to prove that using this kind of information is enough to successfully detect possible faults in source codes.

3.2 Preprocessing

The preprocessing of training data starts with the deletion of unnecessary class- and file-related information for every entry such as filename, parent, path, etc. Furthermore, we binarized the target labels i.e., converting the number of bugs found in a class to 0 or 1. In other words, we separated the classes into "buggy" and "not buggy" sets in order to perform binary classification on it. On the other hand, we applied one-hot encoding on the "Type" feature (class, interface or enum) in pursuit of getting better results and drop one of the newly created features to avoid the Dummy Variable trap. The Dummy Variable trap is a scenario in which the independent variables are multicollinear, or in simple terms one variable can be predicted from the others. If we want to use categorical data such as "Interface", "Class", etc., encoding to numerical data is necessary. If there is no natural ordering over our variables one-hot encoding can be applied to encode our features. One-hot encoding create a new variable for all categorical value, which is highly correlated with each other.

Other preprocessing possibilities for the features are the normalization – where metrics are linearly transformed into the [0,1] interval and standardization

– where features are transformed in order to get zero mean and one standard deviation. Standardized data is essential for accurate data analysis, it is easier to draw clearer conclusions about the current data when one has other data to measure it against.

3.3 Resample Techniques

In our dataset the distribution of examples across the classes is biased. From the 47,618 samples there are 38,838 labelled as "not buggy" which makes up 82% of the overall data. Performing binary classification on an unbalanced dataset is a challenging task. In order to solve this problem, we used different resample techniques, such as SMOTE [5], RUS [13] and random upsampling method which was also used in the Deep Water Framework (DWF) [11]. In the following, let us dedicate a few sentences to these algorithms.

SMOTE (Synthetic Minority Oversampling Technique) selects a random sample from the minority class, then selects its k nearest neighbors from the feature space. It chooses a random point from the k neighbors then connects to the originally selected sample with a line, and creates new samples along the line in feature space. So the new samples will be convex combinations of the original samples of the minority class.

RUS (Random Under-Sampling) uses all the samples from the minority class and random points from the majority class in order to achieve a balanced dataset.

The random upsampling strategy works by duplicating randomly selected samples from the minority class until the size of the two classes reach the size of the majority class multiplied by the given factor. In our study we applied a 50% upsampling rate, as it was shown to be the best upsampling parameter on the same dataset [11].

3.4 Learners

During our experiments we applied several learning techniques, including different ensemble learners combined with various base learner techniques. Several experiments show that the use of ensemble models can improve the performance of base learning techniques [18, 24]. Ensemble models combine the results of multiple base learner instances in order to provide a prediction for the class of the given sample. Different learner methods can be used as base learning methods as long as they provide a slightly better prediction than a random choice would. As base learners, we tried the Gaussian Naïve Bayes, along with the Decision Tree classifier, which considered to be a good choice for ensemble learning approaches [28]. We applied both Bagging and AdaBoost as ensemble learners for these base learners. The latter has shown to be one of the best choices as an ensemble learner [18].

During our experiments, we used the implementation of the learning techniques provided in the scikit learn python package[3].

[3] https://scikit-learn.org/

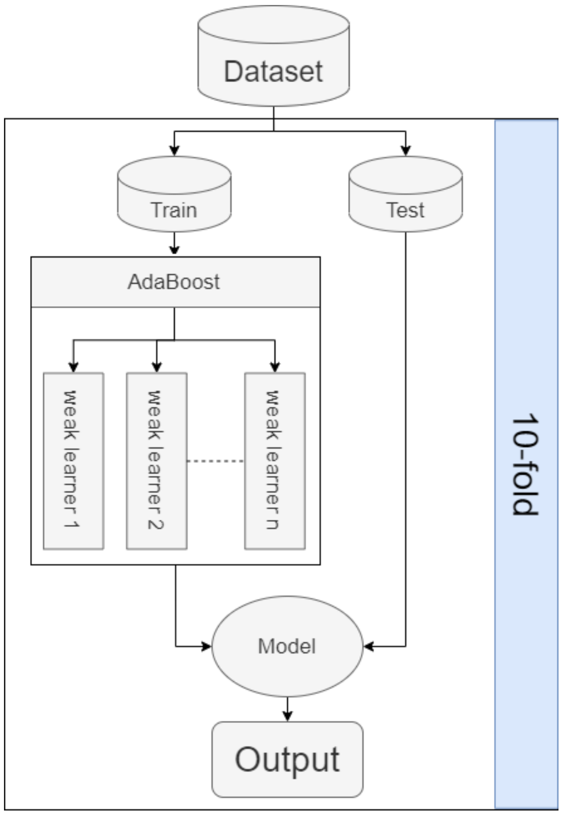

Fig. 1. Workflow of the approach

4 Evaluation

In our study, we used *10-Fold* strategy to evaluate our models. It means that we had 10 different splits of our data, which all contained separate sets for both the train and test data. Then, the given model has been evaluated on the 10 train-test datasets and the given results has been averaged. This workflow can be seen in Figure 1.

During the evaluation to measure our models' performance, we mainly focused on F-measure, although we calculated Accuracy, Precision, and Recall as well. All of these metrics are calculated from the True Positive, False Positive, True Negative, and False Negative values of the confusion matrix with the following formulas.

True Positive: The number of correctly labeled "bugged" samples.
False Positive: The number of incorrectly labeled "not bugged" samples.
True Negative: The number of correctly labeled "not bugged" samples.
False Negative: The number of incorrectly labeled "bugged" samples.

Table 1. F-measure values for different preprocessing techniques

Ensemble learner	Base learner	None	Normalize	Standardize	Normalize + Standardize
AdaBoost	DecisionTree	47.38%	45.13%	47.39%	47.39%
AdaBoost	GaussianNB	30.62%	36.44%	31.08%	31.08%
Bagging	DecisionTree	28.51%	28.51%	28.51%	26.05%
Bagging	GaussianNB	35.02%	35.02%	35.67%	37.17%

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

Precision:
$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Recall:
$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

F-measure:
$$\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F-measures values measured with different ensemble learner, base learner combinations, and with different preprocessing techniques can be seen on Table 1. To keep the experiment as simple as possible, we did not use any preprocessing technique (normalization and standardization) later on our dataset (the achieved results were similar when no normalization and/or standardization was involved).

4.1 RQ1: Does AdaBoost performs better than other classifier methods for bug prediction?

As previous studies also showed that AdaBoost can be a superior choice of ensemble learning techniques, we mainly focus on this algorithm. In order to measure the performance of AdaBoost, we compared AdaBoost with another widely-used classifier, namely the Bagging classifier. We selected Bagging instead of other classifiers, because Bagging is an ensemble learner as well, which gives us a good comparison base against the AdaBoost classifier. As mentioned in Section 3.4, we tried two different weak learners i.e. the Decision Tree and Naïve Bayes classifiers. Our measurements are shown in Table 2.

The table shows the ensemble learners in the first column before specifying the base learner in the second column. We have grouped the entries based on these two and separated the groups with a horizontal line. In each group, we highlighted the best configuration with bold typeface. Best algorithms were marked based on the F-Measure scores.

As the table shows, the configuration for the best result was AdaBoost with Decision Tree after some hyperparameter tuning. We could achieve an F-Measure of 54.61% with a high accuracy (81.96%) with the configuration of 300 estimators and a learning rate of 0.05. High accuracy in itself is not enough since the dataset is biased, however the precision is above 50% which means that half of entries marked as buggy were in fact buggy. Finally, we could identify 58.9% of the total buggy entries as the recall value suggests.

Answering RQ1: As we can see the AdaBoost performs better with the Decision Tree base learner than the Bagging. On the other hand, Bagging seems to provide slightly better results when used with Naïve Bayes and not with Decision Tree. Our measurements prove that with the best hyperparameters AdaBoost classifier with Decision Tree can outperform other classifiers on the Unified Bug Dataset [11].

4.2 RQ2: Is there any resample technique which performs consistently better than others?

Since our dataset was imbalanced as mentioned in Section 3.3, some kind of resampling is necessary. SMOTE, RUS, and random upsampling techniques, which were briefly explained, can enhance the reliability and efficiency of the learning algorithms. Table 3 shows the F-Measure results of the ensemble learning algorithms with respect of the resampling methods. As it can be depicted, random

Table 2. Evaluation of ensemble learners

Ensemble learner	Base learner	Number of estimators	Learning rate	F-measure	Accuracy	Precision	Recall
AdaBoost	DecisionTree	100	0.05	54.13%	81.06%	48.91%	60.64%
AdaBoost	DecisionTree	200	0.05	54.61%	81.56%	50.00%	60.17%
AdaBoost	**DecisionTree**	**300**	**0.05**	**54.61%**	**81.96%**	**50.92%**	**58.90%**
AdaBoost	DecisionTree	400	0.05	54.12%	81.89%	50.75%	57.98%
AdaBoost	DecisionTree	500	0.04	53.93%	81.83%	50.62%	57.72%
AdaBoost	**GaussianNB**	**100**	**0.04**	**37.23%**	**77.41%**	**38.38%**	**36.52%**
AdaBoost	GaussianNB	200	0.04	36.10%	75.78%	35.32%	37.23%
AdaBoost	GaussianNB	300	0.04	35.34%	74.36%	33.18%	38.09%
AdaBoost	GaussianNB	400	0.04	34.94%	73.45%	32.04%	38.76%
AdaBoost	GaussianNB	500	0.04	34.58%	72.78%	31.21%	39.09%
Bagging	**DecisionTree**	**100**		**50.54%**	**79.21%**	**45.00%**	**57.67%**
Bagging	DecisionTree	200		50.48%	79.20%	44.98%	57.53%
Bagging	DecisionTree	300		50.48%	79.21%	45.00%	57.49%
Bagging	DecisionTree	400		50.48%	79.22%	45.01%	57.49%
Bagging	DecisionTree	500		50.51%	79.22%	45.02%	57.55%
Bagging	GaussianNB	100		35.14%	80.50%	45.39%	28.69%
Bagging	**GaussianNB**	**200**		**35.16%**	**80.49%**	**45.38%**	**28.71%**
Bagging	GaussianNB	300		35.09%	80.48%	45.33%	28.64%
Bagging	GaussianNB	400		35.12%	80.49%	45.34%	28.68%
Bagging	GaussianNB	500		35.10%	80.48%	45.32%	28.67%

upsampling (*RandUp.*) achieved the best results regarding the F-measure values in all cases (marked as bold in the table).

Besides the results of F-Measure, Table 4, 5 and 6 show the Accuracy, Precision, and Recall results, respectively. In case of Accuracy and Precision, none of the resampling techniques could improve upon the original results. However, in case of recall, RUS could improve a vast amount when used with Decision Tree as a base learner.

During our experiments, we primarily focus on the F-Measure metric, so we used random upsampling as our resampler in our workflow later on. These values are acquired with fixed hyperparameters because of consistency. The hyperparameters for AdaBoost were *300* for *Number of estimators*, *0.05* for the *Learning rate*, and the hyperparameters for Bagging were *100* for *Number of estimators*, *1.00* for *Max features*, *1.00* for *Max samples* as well. The Decision Tree's fixed hyperparameters were *6* for *Max depth*, *22* for *Min sample leaf* along with *gini* as *Criterion*. The rest of the parameters for learning was the default values provided by the implementation of scikit learn.

Answering RQ2: Our experiments show the three resampling techniques can get better scores regarding specific metrics. The question itself can depend on the context, but in our case, where F-measure was the primary evaluation metric, we can clearly state that the random upsampling is the best choice.

4.3 RQ3: Which is the best weak learning algorithm and which parameter configuration is the most powerful?

As previously mentioned, Table 2 shows AdaBoost and Bagging classifiers work better with Decision Tree than the Gaussian Naïve Bayes classifier. The best parameters for Decision Tree classifier depends on the ensembler learner too. The best hyperparameters for the AdaBoost Classifier with different ensemblers shown in Table 7 and Table 8.

Table 3. F-measure values of different resampling techniques

Ensemble learner	Base learner	None	RandUp.	RUS	SMOTE
AdaBoost	DecisionTree	47.38%	**53.91%**	52.83%	48.01%
AdaBoost	GaussianNB	30.62%	**32.39%**	32.31%	33.21%
Bagging	DecisionTree	28.51%	**50.51%**	49.16%	47.49%
Bagging	GaussianNB	35.02%	**35.10%**	35.53%	36.14%

Table 4. Accuracy values of different resampling techniques

Ensemble learner	Base learner	None	RandUp.	RUS	SMOTE
AdaBoost	DecisionTree	**84.76%**	81.93%	75.27%	84.59%
AdaBoost	GaussianNB	**73.69%**	68.57%	69.15%	63.89%
Bagging	DecisionTree	**83.29%**	79.22%	71.32%	75.72%
Bagging	GaussianNB	80.47%	**80.48%**	80.34%	80.34%

Table 5. Precision values of different resampling techniques

Ensemble learner	Base learner	None	RandUp.	RUS	SMOTE
AdaBoost	DecisionTree	**65.12%**	50.86%	40.75%	63.55%
AdaBoost	GaussianNB	**29.83%**	26.97%	27.23%	25.26%
Bagging	DecisionTree	**67.57%**	45.02%	36.55%	39.50%
Bagging	GaussianNB	45.25%	**45.32%**	44.90%	44.99%

Table 6. Recall values of different resampling techniques

Ensemble learner	Base learner	None	RandUp.	RUS	SMOTE
AdaBoost	DecisionTree	37.26%	57.36%	**75.14%**	38.61%
AdaBoost	GaussianNB	31.49%	40.79%	40.00%	**48.68%**
Bagging	DecisionTree	18.08%	57.55%	**75.18%**	59.56%
Bagging	GaussianNB	28.58%	28.67%	29.41%	**30.21%**

Only 100 estimators works best for Bagging which is positive, however, the achieved results were also lower. Optimal estimator number is 300 in case of Decision Tree which could be argued whether causes a slower runtime or not. Although we did not have the exact measurements for runtimes, we can state that runtime was never an issue.

In case of the other common parameters, Adaboost and Bagging are on consensus. Both ensemble learners performed the best with a decision tree having the max depth of 6 and min sample leaf as 22. Gini is superior over entropy in both cases for the Criterion parameter.

Answering RQ3: Based on our measurements, the best hyperparameters for the AdaBoost classifier and the Decision Tree were *300* for *Number of estimators*, *0.05* for the *Learning rate*, *6* for *Max depth*, *22* for *Min sample leaf* along with *gini* as *Criterion*, and the best hyperparameters for Bagging and Decision Tree were *100* for *Number of estimators*, *1.00* for *Max features*, *1.00* for *Max samples*, *6* for *Max depth*, *22* for *Min sample leaf* along with *gini* again as *Criterion.*

5 Threats to Validity

In our work we tried to be as objective as possible. However, there could be some factors that could make us produce invalid results. One factor could be the quality of the dataset we used. Since we used an already published dataset, we had no influence over the quality of it. The Unified Bug Dataset has numerous

Table 7. Best hyperparameters for AdaBoost and Decision Tree

Number of estimators	Learning rate	Max depth	Min sample leaf	Criterion
300	0.05	6	22	gini

Table 8. Best hyperparameters for Bagging and Decision Tree

Number of estimators	Max features	Max samples	Max depth	Min sample leaf	Criterion
100	1	1	6	22	gini

reviews [2, 6, 11, 9], so it makes us believe that the dataset is a reliable source for software fault detection studies.

Another factor could be that we choose to fix the parameters of the preprocessing techniques, resample techniques and we had a limited search space for our experiments due to our limited resources. During our study we used 1,337 as a fixed seed for every execution in order to make our results reproducible. To measure how changes of these values would have made an impact on our results was out of the scope of this study, since it would have blown the search space.

6 Conclusion and Future Work

In this paper, we presented a detailed approach on how to apply AdaBoost classifier with different base learners in order to predict software faults from static source code metrics alone. We focused our study on revealing the capabilities of AdaBoost classifier in bug detection. We investigated the optimal parameter setup both for the ensemble learner and the base learner. We also tested whether different preprocessing steps would enhance the effectiveness of the ensemble learning method regarding F-Measure, Accuracy, Precision, and Recall metrics as well. Our study used a more recent public bug dataset, the Unified Bug Dataset as its input in order to check the generalizability of the ensemble learners.

We concluded that AdaBoost with a proper resampling technique can be an appropriate method for software fault prediction based on static source code metrics, especially combined with Decision Tree classifier. We could achieve 54.61% F-Measure, 81.96% Accuracy, 50.92% Precision, and 58.90% Recall with a proper parameterization.

Applying different resampling technique is highly context-sensitive. When F-Measure is the main focus of ours, we should employ random upsampling, while Recall is the most important metric, one should apply RUS with a decision tree.

The advantage of applying Decision Trees over Naïve Bayesian methods as base learners is clearly visible. We got the best end results by setting the number of estimators to 300, the learning rate to 0.05, max depth to 6, min sample leaf to 22 and the criterion to gini in case of AdaBoost with Decision Tree. In case of Bagging, 100 estimators gave the optimal results with max features and max samples both set to 1. Additional decision tree parameters (max depth, min sample leaf, criterion) should be unchanged in case of Bagging.

Out future plans include trying different ensemble learner methods combined with other base learner techniques. We would also like to try other preprocessing techniques as well in our future work. We also need to test the effectiveness of ensemble learning techniques on file and especially on method level to see if the conclusions hold.

Overall, we consider our findings a successful step towards understanding the AdaBoost classifier and the role it can play in software fault prediction.

Acknowledgement

This research was supported by the project ”Integrated program for training new generation of scientists in the fields of computer science”, no EFOP-3.6.3-VEKOP-16-2017-00002. The project has been supported by the European Union and co-funded by the European Social Fund.

The research was partly supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (MILAB),and by grant NKFIH-1279-2/2020 of the Ministry for Innovation and Technology, Hungary, and by grant 2018-1.2.1-NKP-2018-00004 “Security Enhancing Technologies for the IoT” funded by the Hungarian National Research, Development and Innovation Office.

References

1. OpenStaticAnalyzer static code analyzer. (2021), https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer
2. Bejjanki, K.K., Gyani, J., Gugulothu, N.: Class imbalance reduction (cir): A novel approach to software defect prediction in the presence of class imbalance. Symmetry **12**(3) (2020). https://doi.org/10.3390/sym12030407, https://www.mdpi.com/2073-8994/12/3/407
3. Catal, C.: Software fault prediction: A literature review and current trends. Expert systems with applications **38**(4), 4626–4636 (2011)
4. Chaturvedi, K., Bedi, P., Misra, S., Singh, V.: An empirical validation of the complexity of code changes and bugs in predicting the release time of open source software. In: 2013 IEEE 16th International Conference on Computational Science and Engineering. pp. 1201–1206 (2013). https://doi.org/10.1109/CSE.2013.201
5. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research **16**, 321–357 (2002)
6. Compton, R., Frank, E., Patros, P., Koay, A.: Embedding java classes with code2vec: Improvements from variable obfuscation. In: Proceedings of the 17th International Conference on Mining Software Repositories. p. 243–253. MSR ’20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3379597.3387445, https://doi.org/10.1145/3379597.3387445
7. D’Ambros, M., Lanza, M., Robbes, R.: An extensive comparison of bug prediction approaches. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). pp. 31–41 (2010). https://doi.org/10.1109/MSR.2010.5463279
8. Ferenc, R., Bán, D., Grósz, T., Gyimóthy, T.: Deep learning in static, metric-based bug prediction. Array **6**, 100021 (Jul 2020). https://doi.org/10.1016/j.array.2020.100021,

http://www.sciencedirect.com/science/article/pii/S2590005620300060, open Access
9. Ferenc, R., Siket, I., Hegedűs, P., Rajkó, R.: Employing Partial Least Squares Regression with Discriminant Analysis for Bug Prediction. arXiv e-prints arXiv:2011.01214 (Nov 2020)
10. Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., Gyimóthy, T.: A public unified bug dataset for java and its assessment regarding metrics and bug prediction. Software Quality Journal **28**, 1447–1506 (2020). https://doi.org/10.1007/s11219-020-09515-0, https://doi.org/10.1007/s11219-020-09515-0, open Access
11. Ferenc, R., Viszkok, T., Aladics, T., Jász, J., Hegedűs, P.: Deep-water framework: The swiss army knife of humans working with machine learning models. SoftwareX **12**, 100551 (2020). https://doi.org/https://doi.org/10.1016/j.softx.2020.100551, https://www.sciencedirect.com/science/article/pii/S2352711019303772
12. Gao, Y., Yang, C.: Software defect prediction based on adaboost algorithm under imbalance distribution. In: 2016 4th International Conference on Sensors, Mechatronics and Automation (ICSMA 2016). Atlantis Press (2016)
13. Hasanin, T., Khoshgoftaar, T.: The effects of random undersampling with simulated class imbalance for big data. In: 2018 IEEE International Conference on Information Reuse and Integration (IRI). pp. 70–79. IEEE (2018)
14. Jiang, Y., Cukic, B., Ma, Y.: Techniques for evaluating fault prediction models. Empirical Software Engineering **13**(5), 561–595 (2008)
15. Khan, M.Z.: Hybrid ensemble learning technique for software defect prediction. International Journal of Modern Education & Computer Science **12**(1) (2020)
16. Kumari, M., Misra, A., Misra, S., Fernandez Sanz, L., Damasevicius, R., Singh, V.: Quantitative quality evaluation of software products by considering summary and comments entropy of a reported bug. Entropy **21**(1) (2019). https://doi.org/10.3390/e21010091, https://www.mdpi.com/1099-4300/21/1/91
17. Nevendra, M., Singh, P.: Software bug count prediction via adaboost.r-et. In: 2019 IEEE 9th International Conference on Advanced Computing (IACC). pp. 7–12 (2019). https://doi.org/10.1109/IACC48062.2019.8971588
18. Peng, Y., Kou, G., Wang, G., Wu, W., Shi, Y.: Ensemble of software defect predictors: an ahp-based evaluation method. International Journal of Information Technology & Decision Making **10**(01), 187–206 (2011)
19. Petrić, J., Bowes, D., Hall, T., Christianson, B., Baddoo, N.: The jinx on the NASA software defect data sets. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering. pp. 1–5 (2016)
20. Polikar, R.: Ensemble learning. In: Ensemble machine learning, pp. 1–34. Springer (2012)
21. Ren, J., Qin, K., Ma, Y., Luo, G.: On software defect prediction using machine learning. Journal of Applied Mathematics **2014** (2014)
22. Sayyad Shirabad, J., Menzies, T.: The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada (2005), http://promise.site.uottawa.ca/SERepository
23. Schapire, R.E.: Explaining adaboost. In: Empirical inference, pp. 37–52. Springer (2013)
24. Sharma, S., Kumar, S.: Analysis of ensemble models for aging related bug prediction in software systems. In: ICSOFT. pp. 290–297 (2018)
25. Singh, V.B., Misra, S., Sharma, M.: Bug severity assessment in cross project context and identifying training candidates. Journal of Information & Knowledge Management **16**(01), 1750005 (2017). https://doi.org/10.1142/S0219649217500058, https://doi.org/10.1142/S0219649217500058

26. Tóth, Z., Gyimesi, P., Ferenc, R.: A public bug database of github projects and its application in bug prediction. In: Gervasi, O., Murgante, B., Misra, S., Rocha, A.M.A., Torre, C.M., Taniar, D., Apduhan, B.O., Stankova, E., Wang, S. (eds.) Computational Science and Its Applications – ICCSA 2016. pp. 625–638. Springer International Publishing, Cham (2016)
27. Wang, S., Yao, X.: Using class imbalance learning for software defect prediction. IEEE Transactions on Reliability **62**(2), 434–443 (2013)
28. Yucalar, F., Ozcift, A., Borandag, E., Kilinc, D.: Multiple-classifiers in software quality engineering: Combining predictors to improve software fault prediction ability. Engineering Science and Technology, an International Journal **23**(4), 938–950 (2020)