Optimizing bug prediction in software testing using Super Learner

MSc Research Project
Data Analytics

Prajeeth Raghunath Nair
Student ID: x17167442

School of Computing
National College of Ireland

Supervisor: Dr Vladimir Milosavljevic

Student Name:	Prajeeth Raghunath Nair
Student ID:	x17167442
Programme:	MSc Data Analytics
Year:	2018-19
Module:	MSc Research Project
Supervisor:	Dr Vladimir Milosavljevic
Submission Due Date:	12/08/2019
Project Title:	Optimizing bug prediction in software testing using Super Learner
Word Count:	7594
Page Count:	23

Signature:	
Date:	11th August 2019

Optimizing bug prediction in software testing using Super Learner

MSc Research Project
Prajeeth Raghunath Nair
x17167442

11th August 2019

Abstract

Software testing is a crucial part of every software project to ensure that the applications delivered to the end-users are defect-free and reliable. Mining of software repositories can uncover useful software metrics which can aid in the early detection of bugs through software fault prediction. This information can be utilized by software project managers to handle resource allocation and optimize the testing process effectively. This paper proposes a novel super learner classification technique for predicting bug-prone modules in the software. The base learners for the classifier comprises of Support Vector Machine (SVM), Decision Tree, Logistic Regression (LR), and the meta learner used is Extreme Gradient Boosting (XGBoost). The experiment is carried out on publicly available datasets from PROMISE repository and Eclipse bug prediction dataset. The results show that by combining the predictions of multiple base learners, the presented Super learner provides a robust and generalized performance compared to the individual classifiers for predicting bugs in software.

Keywords– Software fault prediction, defects, bugs, ensembles, machine learning, classification

1 Introduction

Software projects are generally implemented in stages adhering to the Software Development Lifecycle (SDLC). One of the crucial parts of this lifecycle is Software Testing, which has its methodologies known as the Software Testing Lifecycle (STLC). Software testing consumes a significant portion of any software projects time and effort, taking up to 35-50% of the budget for a software project (Malhotra; 2014; Kasurinen; 2010).More functionalities need to be added to a software as it matures due to technological advancements or growing market competition. The addition of these functionalities can further complicate the development process increases the risk of defects (Chen, Fang and Shang; 2016; Menzies, Milton, Turhan, Cukic, Jiang and Bener; 2010). Furthermore, it is impossible to exhaustively test a software application with limited time and resource constraints given the agile nature of development(Kim, Zimmermann, Whitehead and Zeller; 2007). The annual report for 2018 by Tricentis, a global leader in software testing, states that around 3.6 billion people were affected worldwide due to software bug related issues, which resulted in $1.1 trillion financial losses (*Tricentis*; 2018). This further indicates that the cost to fix a bug missed in the initial stages grows multifold as the software is close to its release. Thus it is necessary to detect and fix bugs as early as possible in the project lifecycle to minimize potential risks (Malhotra; 2014; Kasurinen; 2010). A possible solution to this problem is to concentrate the testing efforts on specific modules of the software that are more prone to be buggy. (Ma, Zhou, Xu, Chen and Yang; 2015).

Mining of software archives and version control systems containing code files, execution logs and bug information can provide valuable insights on the quality of software and further help in setting best practices. The project managers can use this newfound information to allocate resources and time more efficiently to better manage an agile project (Moustafa, ElNainay, Makky and Abougabal; 2018; Khoshgoftaar and Gao; 2009). The study by D'Ambros, Lanza and Robbes (2010); D'Ambros, Gall, Lanza and Pinzger (2008); Kagdi, Collard and Maletic (2007) discusses various data mining techniques to extract information from version control systems and software archives. Software fault prediction involves the ability to develop a predictive model that identifies bug-prone modules in a software application based on various software metrics and bug history. If implemented in long-term projects having sufficient historical data, it can aid in efficiently managing the software development activities. This could further help in optimizing resource utilization and reduce project expenditures (Rathore and Kumar; 2016; Xing, Guo and Lyu; 2005).

The domain of software bug prediction has intrigued many researchers in the past.It is not often necessary to predict the precise number of bugs in each module of a software application as it found that around 20% of the modules in a software contains about 80% of the bugs Boehm and California (2001). Thus the density of bugs is not evenly spread across the modules in a software. A code reviewer on average reviews around 8-20 LOC/minute (Lines of code per minute) and this effort needs to be spent individually for the entire team which can become highly labor-intensive (Kelly, Sherif and Hops; 1992). Therefore the ability to predict specific modules that are more prone to be buggy can further aid in prioritizing the efforts for code reviews, inspections and detailed testing (Yousef; 2015). This lead to further studies exploring various machine learning techniques like Fuzzy clustering using Radial Basis Function(Mahaweerawat, Sophatsathit and Lursinsap; 2002), SPRINT and CART(Khoshgoftaar and Seliya; 2002), ANN (Artificial Neural Networks) and Clustering Genetic Algorithm (Qi Wang, Bo Yu and Jie Zhu; 2004), Support Vector Machines (SVM) (Xing, Guo and Lyu; 2005) for classification in Software Bug Prediction.

Like most real-world problems, the presence of bugs in a software is not uniform across modules. There is a high imbalance in the distribution of bugs across modules in a softwareBoehm and California (2001). The number of faulty modules in a software application is considerably less than the non-faulty modules. This imbalance is usually not taken into consideration while developing a predictive model resulting in poor performance on unseen data (Öztürk; 2017). Some studies suggest the use of undersampling and oversampling techniques to handle the imbalance in data. Also, no individual classifier performs consistently across data from different projects, and hence, no single model is superior to the other in this context. The problem of software fault prediction is non-convex and since it is computationally expensive to find the global minima using a single technique, the use of ensembles can provide a feasible solution. Some of the studies by Wu, Wang, Peng, Shi and Lou (2011) and Laradji, Alshayeb and Ghouti (2015) proposed the use of ensemble technique such as Bagging, Boosting, Voting and Average Probability Ensemble for classifying bug-prone modules.

This raises the question: What techniques can be used to optimize the identification and classification of bug-prone modules in a software?

This research presents a novel approach to build a robust and generalized model for software defect prediction through feature selection techniques and handling of class imbalance in data using a Super Learner based ensemble with base learners such as Support Vector Machines(SVM), Logistic Regression(LR), Decision Trees(DT) and state-of-the-art technique XGBoost (Extreme

Gradient Boosting) as the meta-learner.

The report is further organized into the following sections: Section 2 provides information about related studies and researches in the field of software fault prediction, Section 3 discusses the methodology followed for conducting the research, Section 4 presents an overview of the modelling and design flowchart, Section 5 presents the implementation of the Super Learner model, Section 6 presents an evaluation of the developed model, Section 7 presents the conclusion.

2 Literature Review

This section contains literature from different studies performed in software fault prediction domain. Several machine learning models and techniques implemented previously are discussed, followed by the use of ensembles in classifying bug-prone modules. This follows with the introduction to the extreme gradient boosting algorithm used as the meta-model in the proposed Super Learner.

2.1 Support Vector Machines

The study by Xing, Guo and Lyu (2005) proposed the use of a novel approach based on SVM(Support Vector Machines) for software bug classification. The experiment was performed on the data mined from an application for Medical Imaging System using 11 software complexity metrics. Principal Component Analysis was used for dimensionality reduction to reduce the interrelated effect of various metrics. The study compared and contrasted two models QDA(Quadratic discriminant analysis) and SVM with and without applying PCA. It was observed that the model using SVM after applying the PCA for dimensionality reduction showed an *accuracy* of around 89% and very low Type I error. However, the use of data pertaining to a single project can be a threat to the generalized capability of the model.

A similar study by Elish and Elish (2008) proposes the use of SVM for software fault classification and compares it against eight other statistical and machine learning models. For this study, four NASA MDP(Metrics Data Program) datasets - CM1, PC1, KC1 and KC3 were used from the PROMISE repository. The Correlation Based Feature selection technique was used to select the best features for training the models. SVM was able to achieve an *accuracy* in the range of 85-94% for the 4 projects. It was also able to achieve a 100% *recall* rate for the CM1 dataset. Although the performance of SVM in terms of *accuracy* and *f-score* was a bit below the Random Forest classifier, it was competitive. Also, a *recall* rate of 100% could indicate overfitting of the model. Gondra (2008) compares the performance of ANNs(Artificial Neural Networks) and SVM on a single project data from the NASA Metrics Data Program. Sensitivity analysis is used to perform feature selection from the available 21 software metrics. The findings of the study indicate that SVM performs significantly better than ANNs with a correct classification rate of 87% as compared to 72% achieved by ANN. However, the study does not consider other evaluation parameters and results are limited to a single project.

Sunghun, Whitehead Jr. and Yi (2008) proposes the use of SVMs for identifying bug-prone modules in software. For the experiment, the software configuration management systems for 12 open source projects are mined to extract software metrics. The chi-square measure is used to perform feature selection while *accuracy*, *recall*, precision and f-score are used as the evaluation parameters. The finding indicate that SVM classifier provides acceptable performance for identifying bug-prone modules with the best *accuracy* of around 86% for the Apache project and least being 65% for the project JEdit. However, there is no comparison made on the performance of SVM to any other models for the same projects. Also, there is no discussion on the handling of class imbalance in data.

2.2 Logistic Regression

El Eman and Melo (2001) performed software fault prediction on a single version of a commercial Java application using Logistic Regression. A subset of 10 features was selected that are associated with the bug proneness and are not correlated to each other. The evaluation is performed using J coefficient, AUC and ROC curve. The findings indicate that the developed Logistic Regression model using design metrics provide acceptable performance in terms of *accuracy.* However, the technique used to select the sub-features is not discussed in detail.

The experiment conducted by Denaro, Pezzè and Morasca (2003) implemented logistic regression for software fault prediction on data collected from an antenna configuration system and the projects Apache 1.3 and Apache 2.0. A combination of 9 orthogonal method-level metrics is selected using Principal component analysis to build the model. For evaluating the results of the study parameters such as R^2, overall completeness, faulty module completeness and faulty module correctness are used. It is observed that logistic regression along with cross-validation provides good performance in terms of identifying bug-prone modules in a software. Although the study has not compared the performance of Logistic Regression with any other models, it does provide a strong indication of the generalization capability of the algorithm on different projects.

The study by Olague et al. (2007) proposes the use of Logistic Regression to classify defect prone modules in software. For this data from several versions of Mozilla Rhino project is used which has a large number of object-oriented metrics. The evaluation of the model was based on accuracy on a different subset of features. The study concludes that the performance of the model provides acceptable *accuracy* in the range of 70-86% for different versions of the software. However, the study does not discuss any feature selection techniques and evaluates the model performance solely on *accuracy.* An experimental study by Cruz and Ochimizu (2009) further explored the use of Logistic Regression in software fault prediction. The data from Mylyn project of Eclipse, an e-commerce system (ECS) and a banking system (BNS) is used for the analysis with metrics such as CBO, RFC and WMC. The use of simple log transformations is done to compare complexity metrics among different projects. For evaluating the performance of the model, statistical tests of significance and goodness of fit are used. However, the study could have used more robust evaluation parameters such as *AUC, accuracy* and *recall* to establish the credibility of the results.

2.3 Decision Trees

For software fault prediction, Khoshgoftaar and Seliya (2003) proposed the use of SPRINT decision tree algorithm, which is an extension of the CART(Classification and Regression Tree) technique. The study makes use of software metrics from four historical releases of a large telecommunication system. A combination of 24 product metrics and 4 execution metrics, i.e., a total of 28 software metrics are used to build the specified model. The performance of the model was evaluated based on Type-I error, Type-II error and misclassification error. The findings of the experiment indicate that trees built using SPRINT are more complex than the tree using CART but provide better stability and robustness. However, the rationale behind choosing the metrics is not explained. A study by Güne Koru and Liu (2007) to identify and characterize bug-prone modules in a software proposed the use of tree-based models. The class-level static metrics from KOffice and Mozilla projects are used for building the predictive model. The experiment aimed at validating Pareto's Law, which states that the majority of issues are concentrated in a small proportion of modules. It was found that the tree-based models provided acceptable performance in identifying bug-prone modules which can aid in efficiently managing project resources. However, there is no discussion on evaluation parameters used for reaching the stated findings.

2.4 Ensembles

The popularity of ensemble techniques has increased in the domain of software bug prediction due to its promising potential in improving the prediction capabilities of models (Rathore and Kumar; 2017). Various studies were reviewed to understand the benefits and application of ensemble machine learning techniques.

2.4.1 Homogeneous ensembles

The failure to identify fault-prone module can be more expensive than misclassifying non-faulty modules in software defect prediction. A solution to this issue by Zheng (2010) proposes the use of three cost-sensitive boosting algorithms using AdaBoost to boost the neural network. Here, four datasets from NASA MDP program consisting of 21 software metrics have been used to train the model. It was observed that the *accuracy* of the neural networks improved using the boosting technique. The evaluation is performed using the parameters *Type I, Type II and ECM(expected cost of misclassification)*. However, Neural Networks, being a deep learning technique, requires a large amount of data to be trained. This needs to be taken into consideration while building a predictive model which may otherwise lead to a biased model.

To improve software fault classification Mauša, Grbac, Bogunović and Bašić (2015) experimented with five releases from the Eclipse JDT project using Rotation Forest, an ensemble of Decision trees. Dimensionality reduction was achieved using Principal Component Analysis (PCA) technique. The resulting model was evaluated using *accuracy, True Positive rate, False Positive rate, F-measure, Kappa statistics* and *Area under receiving operating curve(AUC)*. It was found that the Rotation forest provided better *accuracy* than the Random Forest and Logistic Regression models. However, data from a single project was used to evaluate the performance of the model.

2.4.2 Heterogeneous ensembles

The work by Khoshgoftaar and Seliya (2003) analyses the effects of ensemble methods over traditional individual classifiers such as C4.5 and Decision trees for software fault prediction. The ensemble methods such as bagging, boosting and logit-boost were used in this research and were evaluated only based on *misclassification rates* for Type I and Type II error. The study observed that the ensemble or combined models outperformed the base learners or individual classifiers. It was also found that the boosting model consisting of different base learners worked better than the bagging model. However, the evaluation of these models was based solely on a single performance parameter.

Aljamaan and Elish (2009) ffurther experimented with measuring the performance of bagging and boosting techniques compared to individual classifiers. The machine learning algorithms used in this study are Multilayer perceptron, Radial Basis Function Network, Bayesian Belief Network, SVM and Decision Trees. The model is built using nine class-level metrics from project KC1 of NASA MDP Program. It was observed that the ensemble models using bagging and boosting performed better than individual classifiers. However, classification *accuracy* is the sole parameter used to evaluate the performance of the models. Also, the literature does not discuss the use of any feature selection techniques for building the model. To tackle the challenge of Software Bug prediction, Misirli, Bener and Turhan (2011) proposed an ensemble using Naïve Bayes, Artificial Neural network(ANN), and Voting feature intervals. The models were trained using data of seven projects from the PROMISE repository. A total of 11 features were selected using the InfoGain feature selection method and the performance was evaluated using *probability of detection (pd), balance* and *precision*. The findings indicate that the classification rate of the ensemble classifier was significantly higher than individual classifiers. Similar to the previous literature, Twala (2011) developed an ensemble model using Apriori, Decision

Tree, k-nearest neighbour, Neaïve Bayes and SVM to detect bugs in software. Requirement and static code metrics of four projects from the NASA MDP program are used to develop the fault prediction models. The variable misclassification cost is used to handle class imbalance issue but has not been explained. The findings indicate that ensembles perform better than the chosen single classifiers as can be seen in previous literatures. However, the proposed model was evaluated solely based on the *misclassification rate.*

To further explore ensemble techniques in software fault prediction, Siers and Islam (2015) developed a cost-sensitive classification and voting technique using Decision trees. The data of 6 projects from NASA MDP repository are used for building the model and features are selected using GetGoodAttributes technique. SMOTE (Synthetic minority oversampling technique) is used for handling the class-imbalance by generating synthetic values of faulty classes and is one of the most prominently used techniques for effectively handling class imbalance. The time required to calibrate the proposed ensemble model is less than the time taken by traditional techniques that are used for such predictions. However, only precision and *recall* are used as evaluation metrics for the model.

The work by Elish, Aljamaan and Ahmad (2015) also studies the use of heterogeneous ensembles over homogeneous ensembles. Base learners such as Support Vector Machines, Decision Trees, Radial Bias function or RBF network and Multilayer Perceptron networks are used in the study. The models were trained using ten-fold cross-validation with 11 class-level features or software metrics. The data set used for this study is from the projects UIMS and QUES. The results of this study were similar to the ones observed by Khoshgoftaar and Seliya (2003) and reinstated the fact that heterogeneous ensembles perform better than homogeneous ensembles.

The work by Rathore and Kumar (2017) proposed the use of ensembles for predicting the number of bugs in a software. It proposes to combine the base learners in an ensemble using linear and non-linear combinations. The linear combination approach uses Linear Regression, Genetic Programming and Multilayer perceptron as base learners with a linear regression rule to combine the base learners. The non-linear combination uses Gradient Boosting Regression (GBR) is used as a meta learner. This analysis was performed on projects from the PROMISE repository and class imbalance was handled using SmoteR technique. Evaluation metrics such as *Average absolute error, Prediction at level l, Average relative error, Measure of completeness* are used. It is observed that the non-linear combination performs better than the linear combination of ensembles in terms of *accuracy.* The current study proposes the use of XGBoost to deal with the high computational cost of ensemble methods such as the one mentioned above. This research also differs from the one above by performing a classification task rather than a regression one i.e. it aims to classify the software bugs based on software metrics rather than predicting how many bugs would occur. For this, a super learner model is proposed which will be explained further in the study.

2.5 Introduction to Extreme Gradient Boosting (XGBoost)

Extreme Gradient Boosting(XGBoost) was introduced by (Chen and Ma; 2015) that is more scalable and efficient in terms of performance and accuracy than its predecessor the Gradient Boosting Machines(GBMs). XGBoost is also known as a regularized boosting model. This algorithm is optimized by implementing novel approaches to handle missing data by using techniques such as sparsity-awareness. In the case of weighted datasets, it applies an optimal split for points using a distributed weighted Quantile Sketch algorithm. The computational speed of the XGBoost algorithm is very high and it also makes optimal use of memory Dhaliwal et al. (2018).

XGBoost comes with the following improvements over gradient boosting trees(Chen and Ma; 2015):

- Regularization: It prevents overfitting by using Lasso(L1) and Ridge(L2) regression to make the model more generalized.

- Parallel processing: To increase the execution speed, XGBoost can make use of multiple CPU cores for parallel processing.
- Tree pruning: Unlike the traditional GBM which does a greedy search, XGBoost performs splits to the maximum depth then performs backward pruning of the tree to remove splits which does not provide any positive gain.
- Built-in Cross validation: By allowing a cross-validation at each iteration, it becomes easy to find the exact number of boosting iterations to fit the optimal model.

2.6 XGBoost used in different domains

For this research, the meta-model proposed for building the Super Learner is Extreme Gradient Boosting(XGBoost) since it is a state-of-the-art boosting algorithm not yet studied in the domain of software fault prediction. The following literature discusses the use of XGBoost in other domains similar to software fault prediction.

Ahmadi, Ulyanov, Semenov, Trofimov and Giacinto (2016) implemented a predictive model using XGBoost to classify malware based on data from Microsoft. Here features are grouped based on the varied behavior of malware. The proposed model was able to achieve high *accuracy* of about 98%. A similar study by Zhang, Huang, Ma, Yang and Jiang (2016) proposes an ensemble with XGBoost and ExtraTreeClassifier as the base learners. Here, XGBoost is also used as the meta-model to classify malware threats as the existing techniques do not provide acceptable accuracy and efficiency. Similar to software metrics used in software fault prediction, this research aims to use multiple categories of static features to classify malware families.On experimenting with data from Microsoft, the resulting model was able to achieve an *accuracy* of around 99%. The results show promising capabilities of XGBoost in the classification of fault-prone modules in a software. This literature further supports the idea of using XGBoost as the meta-learner in the current work.

Dhaliwal, Nahid and Abbas (2018) implemented XGBoost to classify malicious packets in network and build a relaible intrusion detection system. This involves deploying the XGBoost on the network socket layer (NSL-KDD) dataset to evaluate its performance. The goal is to understand data integrity, identify malicious packets in the network and discard them to ensure the safety of the network. Hyperparameter tuning is performed to get the optimal values for training the XGBoost model. The proposed model was able to achieve high *recall* rate, followed by *accuracy* and precision of close to 98%.

A similar study by Chen, Jiang, Cheng, Gu, Liu and Peng (2018) explores the use of XGBoost to identify the DDoS(Distributed Denial of service) in software-defined cloud-based network. It is necessary that the developed model is quick to classify incoming DDoS as it can paralyze the entire network. The proposed model is trained using the flow packet data from the TCP dump. It is observed that the XGBoost model is able to achieve high *accuracy* and low false-positive rates. Also, the model is more scalable and significantly faster than other existing techniques.

3 Methodology

This study aims to develop a model that can be applied to software projects and can aid with the early detection of bugs. The above literature review provides a clear idea of the existing approaches used for software fault prediction. The Knowledge Discovery in Databases (KDD) methodology is used for this research as proposed by Fayyad et al. (1996) and generally used in software fault prediction literature (Khoshgoftaar, Allen, Jones and Hudepohl; 1999; Mertik, Lenic, Stiglic and Kokol; 2006; Gayathri and Sudha; 2014).

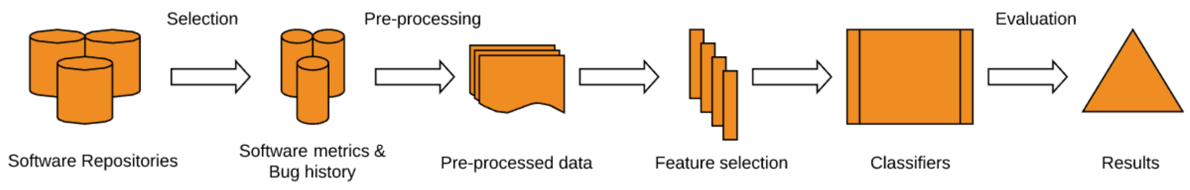


Figure 1: Stages in the KDD process

The 5 stages of the KDD approach is shown in Figure 1 [1]:

- The *Selection* stage involves selecting relevant dataset for analysis.
- The *Pre-processing* stage includes steps such as handling missing values, modifying column names, creating new columns based on available data, encoding factor levels and refining the data.
- The *Feature selection* stage involves techniques to choose only relevant features for building the model.
- In the *Implementation* stage, the proposed machine learning model is developed.
- Finally, the *Evaluation* stage consists of measuring the performance of the models based on different metrics.

3.1 Data Selection

For this study, four different datasets are obtained from two sources - PROMISE[2] and Eclipse[3] bug prediction dataset. The PROMISE repository is used to obtain data for two projects, i.e., ANT v1.7 and PROP v1.0. The other two datasets are obtained from Eclipse software repository created by Zimmermann, Premraj and Zeller (2007). Each record in the dataset represents a class file or module. The features in the datasets are software metrics which are extracted from the version control systems, log files and bug history of the software.

The ANT v1.7 has 745 records and contains 22 features that are software metrics such as wmc(weighted method count), dit(depth of inheritance tree), rfc(response for class), lcom (lack of cohesion in methods),loc (line of code), cbo(Coupling between objects) and Number of bugs etc. The project PROP v1.0 contains 18,472 records with 22 software metrics similar to the Ant project. The project Eclipse v2.0 has 6,729 records and 202 software metrics such as ACD(Number of anonymous type declarations), FOUT (Number of method calls (fan out)), MLOC (Method lines of code), NBD (Nested block depth), PAR(Number of parameters), VG (McCabe cyclomatic complexity), Number of bugs. The dataset for project Eclipse 3.0 contains 10,593 records and has features similar to Eclipse v2.0. The data from projects used in this study along with the software metrics are summarised in Figure 2 below.

[1]https://www.lucidchart.com/
[2]https://github.com/klainfo/DefectData/tree/master/inst/extdata/terapromise/ck
[3]https://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/

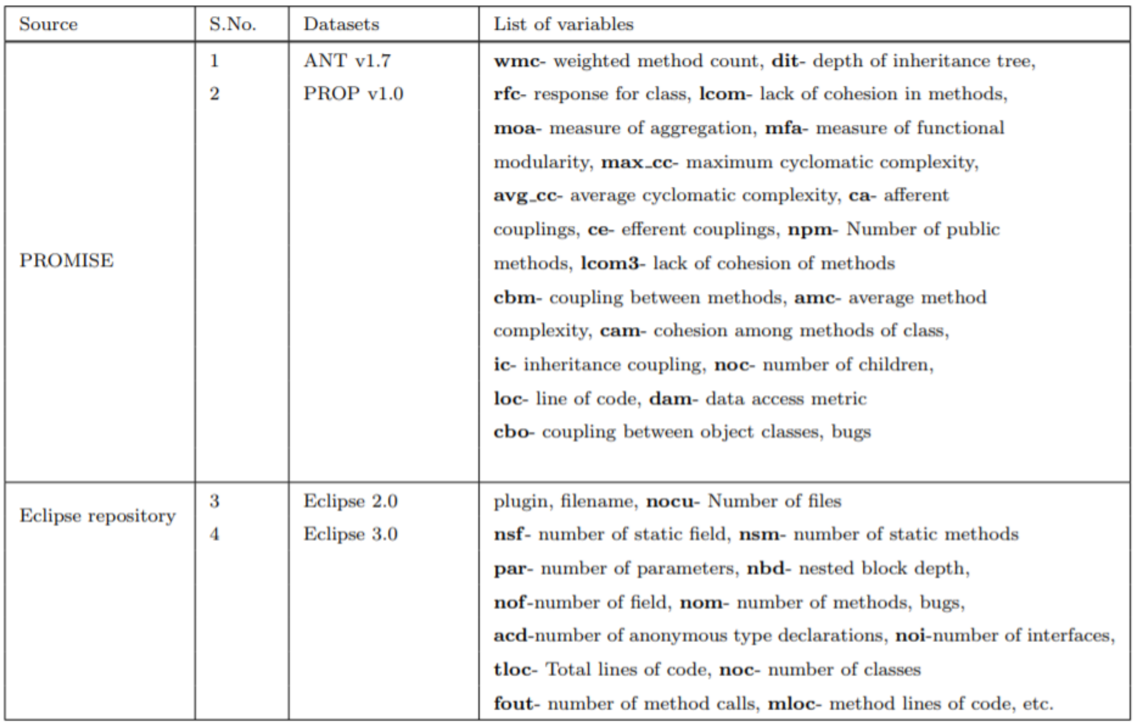
Project	Bug present	Bug not present	Proportion of buggy modules	Total number of modules
Eclipse v2.0	975	5754	14.4%	6729
Eclipse v3.0	1568	9025	14.8%	10593
Ant v1.7	579	166	22.2%	745
Prop v1	2738	15733	14.8%	18471

Table 1: Dataset summary

3.2 Preprocessing

The datasets available from the PROMISE and Eclipse software repository had complete cases of all records; therefore, missing values did not have to be handled separately. However, other pre-processing tasks were carried out on the dataset as below.

- The redundant columns are dropped as they do not significantly contribute to the overall prediction. For example, the *version* column with value '1' for all the records.
- The records that have LOC(Line of code) feature as 0 is dropped, as a module or class file without a single line of code cannot contain any bug.
- Since the study aims to develop a classifier to identify bug-prone modules, the dependent variable *Number of bugs* is converted into factor and renamed as *Has bugs?* with two levels: Yes - if the count of bugs is 1 or more and No - if the count of bugs is 0.
- Further, to fit the meta learner, XGBoost, dummies are created using one-hot encoding technique through *createDummyFeatures* method of *mlr* package in R.



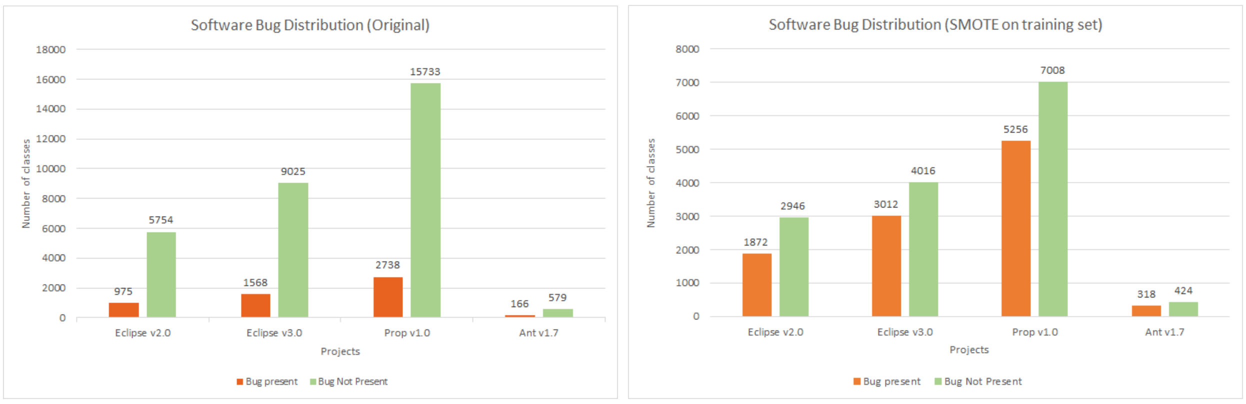
Source	S.No.	Datasets	List of variables
PROMISE	1 2	ANT v1.7 PROP v1.0	**wmc**- weighted method count, **dit**- depth of inheritance tree, **rfc**- response for class, **lcom**- lack of cohesion in methods, **moa**- measure of aggregation, **mfa**- measure of functional modularity, **max_cc**- maximum cyclomatic complexity, **avg_cc**- average cyclomatic complexity, **ca**- afferent couplings, **ce**- efferent couplings, **npm**- Number of public methods, **lcom3**- lack of cohesion of methods **cbm**- coupling between methods, **amc**- average method complexity, **cam**- cohesion among methods of class, **ic**- inheritance coupling, **noc**- number of children, **loc**- line of code, **dam**- data access metric **cbo**- coupling between object classes, bugs
Eclipse repository	3 4	Eclipse 2.0 Eclipse 3.0	plugin, filename, **nocu**- Number of files **nsf**- number of static field, **nsm**- number of static methods **par**- number of parameters, **nbd**- nested block depth, **nof**-number of field, **nom**- number of methods, bugs, **acd**-number of anonymous type declarations, **noi**-number of interfaces, **tloc**- Total lines of code, **noc**- number of classes **fout**- number of method calls, **mloc**- method lines of code, etc.

Figure 2: Details of the dataset and software metrics

3.3 Feature Selection

The features used to build a predictive model largely influences its overall performance. Thus it becomes necessary to carefully select features while training the model to create more generalized and simple models having low variance. Since the data from projects under consideration have high dimensionality, it is imperative to choose features that do not suffer from multicollinearity (D'Ambros et al.; 2010). A filter-based feature selection technique, Correlation Based feature selection (CFS) is used as it provides a subset of features that are highly correlated with the dependent variable and have no correlations amongst them(Wang, Khoshgoftaar and Napolitano; 2014; Malhotra, Bahl, Sehgal and Priya; 2017).

Post feature selection, since the data has high class imbalance, the technique *SMOTE*(Synthetic Minority Oversampling Technique) is applied to the training set (Laradji, Alshayeb and Ghouti; 2015). SMOTE makes use of the k-nearest neighbor technique where the nearest neighbors of a minority class are identified and synthetic values are generated using them, in this case, the bug-prone modules. It is essential to handle the class imbalance as the machine learning algorithms need to learn the characteristics of both bug-prone and non-buggy modules to classify an unseen record correctly. If the instances of bug-prone modules are less, the model would be biased in classifying the modules correctly and would identify a module as non-buggy majority of the time. The Figure 3 shows the distribution of the buggy and non-buggy modules in the original datasets and the bug distribution in the training set post applying *SMOTE*.



(a) Bug Distribution in the original dataset

(b) Training set after applying *SMOTE*

Figure 3: Software Bug Distribution

3.4 Models

The proposed Super Learner (Polley and van der Laan; 2010) is an ensemble of two layers. The base layer consists of three machine learning algorithms widely used in Software Fault Prediction domain, namely: Logistic Regression(LR), Support Vector Machines(SVM) and Decision Tree(DT). The meta-layer consists of the state-of-the-art algorithm Extreme Gradient Boosting(XGBoost) to learn from the predictions made by the base learners and provide a more robust, generalized and optimal model.

3.5 Metrics for evaluation

There is no established standard benchmark to evaluate the performance of a software bug prediction model Arora et al. (2015). Hence, for assessing the results, similar metrics used in the literature are recorded. The performance of each base learner is measured individually, and then the performance of the super learner is measured. The following parameters are used for evaluation, namely: *AUC*, *accuracy* and *recall*. This study does not consider parameters like F-score, Precision due to its unstable nature as Software fault prediction deals with highly imbalanced classes Menzies et al. (2007); Malhotra (2015).

***AUC-ROC* curve:** ROC stands for receiver operating characteristic and is a probability curve, which graphically represents the performance of a binary classifier. *AUC* stands for Area Under ROC curve and defines the capability of a model to distinguish between classes. The higher the value of *AUC*, i.e., closer to 1, the better the model is at identifying the correct class labels. In this case, higher the *AUC*, better is the model at identifying bug-prone modules from the non-buggy modules. ROC provides a robust analysis of the classifier performance in the presence of imbalanced class distributions (Kaur and Kaur; 2018). In a ROC curve, the TPR (True positive rate) on the y-axis is plotted against the FPR (False positive rate) on the x-axis. The terms used in ROC is defined below:

$$TPR(True\ Positive\ Rate) = \frac{True\ positive}{True\ positive + False\ negative}$$

$$Specificity = \frac{True\ negative}{True\ negative + False\ positive}$$

$$\begin{aligned} FPR(False\ Positive\ Rate) &= 1 - Specificity \\ &= \frac{False\ positive}{True\ negative + False\ positive} \end{aligned}$$

***Accuracy*:** It is defined as the number of correct predictions made by a classifier over the total number of predictions. It is represented as:

$$Accuracy = \frac{True\ positive + True\ negative}{True\ positive + True\ negative + False\ positive + False\ negative}$$

***Recall*:** It is defined as the ability of a classifier to identify all instances of interest correctly. In this case, accurately identifying all bug-prone modules. It is represented as:

$$Recall = \frac{True\ positive}{True\ positive + False\ negative}$$

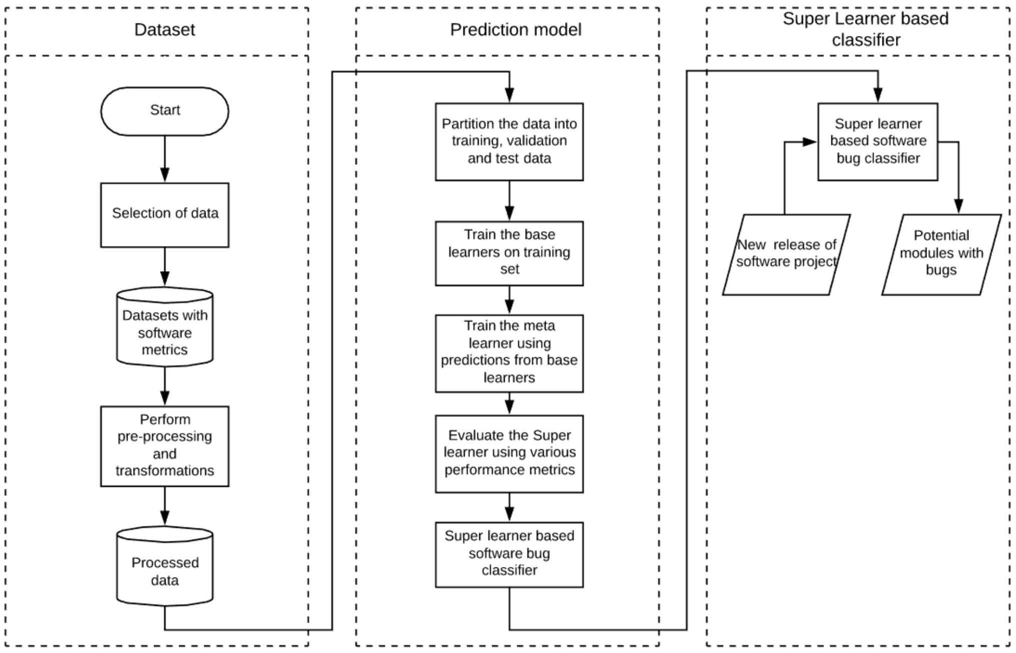
4 Design Specification

This section shows a brief overview of the model design and system development followed in this research.

4.1 Architectural Design

Figure 4 presents an overview of the proposed Super Learner for software fault classification. It contains the following components:

- Dataset: In this stage, datasets containing software metrics and bug history related to the software application are collected.
- Prediction model: Consists of training and evaluating the Super Learner for classifying bug-prone modules
- Super Learner based classifier: The developed model can then be used to classify bug-prone modules in the new release of a software project.



Dataset
Prediction model
Super Learner based classifier
Start
Selection of data
Datasets with software metrics
Perform pre-processing and transformations
Processed data
Partition the data into training, validation and test data
Train the base learners on training set
Train the meta learner using predictions from base learners
Evaluate the Super learner using various performance metrics
Super learner based software bug classifier
Super learner based software bug classifier
New release of software project
Potential modules with bugs

Figure 4: Overview of system development flowchart

4.2 Components of the Super Learner

As seen in Figure 5, the Super Learner classifier consists of two layers - Base learners and Meta learner. The rationale behind selecting these models is that previous literature as per Section 3 shows that the chosen models have provided significant improvements in software bug classification. Also, the performance of the base classifiers are diverse as per the initial visual analysis of their results using box plots. A brief overview of the meta-learner is provided in Section 2. The outline of base learners used in the classifier is discussed below.

4.2.1 Logistic Regression

Logistic Regression is used for classification tasks when the outcome of the response or target variable is binary or dichotomous e.g., 0,1 or "Yes", "No". This model, checks for the probability of occurrence of the response variables (Denaro et al.; 2003). Mathematically, the model is represented using a sigmoid function as

$$f(x) = 1/(1 + e^{-x})$$
$$where\ f(x) = 1,\ x \to \infty,$$
$$f(x) = 0,\ x \to -\infty$$

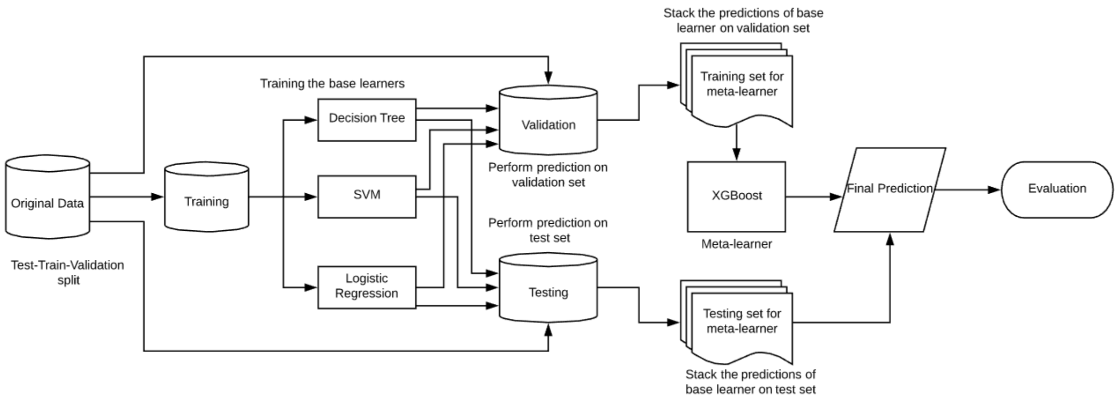

Stack the predictions of base learner on validation set
Training the base learners
Decision Tree
SVM
Logistic Regression
Original Data
Training
Test-Train-Validation split
Validation
Perform prediction on validation set
Perform prediction on test set
Testing
Training set for meta-learner
XGBoost
Meta-learner
Final Prediction
Evaluation
Testing set for meta-learner
Stack the predictions of base learner on test set

Figure 5: Super Learner based model for bug classification

Here, let x = $\theta_0 + \theta_1$x where θ_0 and θ_1 are the parameters of model estimation. In this study, $\theta_0 = 0$ and $\theta_1 = 1$ for the two classes and x is the number of bugs. If the number of bugs is greater than 1, the class is assigned to it is 1 and 0 if less than 1.

4.2.2 Decision Trees

A Decision Tree model was used as a second base learner to train the data. Decision trees can be used for both classification and regression analysis. In this study, classification is carried out using decision tree models. A decision tree classifies data starting from a root node which splits into ”Yes” and ”No” based on the condition (e.g., does this module have bugs? the result of which is a binary response). This node is further split to find the most optimum split. The Gini impurity is checked at each node to decide if a further split is required. For dataset *T*, containing records from *N* classes, *Gini(T)* can be represented as,

$$Gini(T) = 1 - \sum_{i=1}^{N} {p^2}_{\text{i}}$$

where, p_i is the corresponding frequency of class *i* in *T*. This method of splitting from a root node to the base node is called growing trees. Since decision trees try to find optimum splits based on Gini index, they tend to overfit. Therefore, trees need to be pruned, and hyperparameters need to be tuned to avoid overfitting. Here, random search was used for hyperparameter optimization followed by a grid search to further narrow down the search space. However, other advantages of using decision trees are that they perform implicit feature selection and are non-parametric. Therefore, they can handle all types of data i.e., both numerical and categorical and does not make assumptions on the distribution of data. Non-linear relationships between the variables do not affect the performance of decision trees.

4.2.3 Support Vector Machine

The third base learner used for this research is Support Vector Machines(SVM). SVM is used for classification when the data is n-dimensional. It tries to find a hyperplane that separates the classes instead of a linear combination of points when the data is highly dimensional. Hyperplanes are decision boundaries and points are assigned classes based on what side of the plane they are. A hyperplane is created by using support vectors. The support vectors are along the boundary of the plane and the aim is to maximize the distance between these support vectors and the separation plane. If the number of dimensions is 2, then the plane is a straight line that divides the two classes and a 3-dimensional plane if there are more than two classes. SVMs are more robust and powerful than Logistic Regression models and are capable of handling large dimensions of data. Some times data is not linearly separable in 2D space, but when viewed from a higher or different dimension, it can appear to be linearly separable. A kernel trick or approach can be used in such a case. SVM supports many different types of kernels, namely: Gaussian, Polynomial, Sigmoid, etc. In this research, SVM with radial bias kernel function is used. Mathematically, RBF kernel is represented as

$$\mathrm{K}(\mathrm{x_i},\mathrm{x_j}) = e^{-\gamma(x_\mathrm{i} - x_\mathrm{j})^2}$$

where γ is the tuning parameter. This parameter depends on the euclidean distance between the support vectors chosen. If the value of γ is high, the value of the kernel will be very small from the equation above. This will result in the boundary plane overfitting the data. Therefore, the tuning parameter needs to be selected carefully.

5 Implementation

In this section, the implementation of the proposed Super Learner model for software bug prediction is discussed in detail. This study is performed on RStudio using the language R as it provides all the necessary packages and libraries to build and train the model. The collected dataset for four different projects from PROMISE and Eclipse software bug repositories are pre-processed and refined before building the models. All the necessary packages and library are installed in RStudio to carry out this study. The package *dplyr* is used to perform pre-processing of the data i.e., to filter out irrelevant records and features as discussed in the pre-processing steps in Section 3. The package *caret* is predominantly used to train all the base learners, namely: Support Vector Machine(SVM), Logistic Regression(LR) and Decision Trees(DT) as it supports training on a wide range of machine learning algorithms.The *caret* package also provides methods to tune the hyperparameters along with measuring the performance of the classifiers. The function *CreateDataPartition* of *caret* package is also used to create random stratified partitioning of the dataset into training, validation and testing in the ratio 60:20:20. The library *Biocomb* is used to implement the *correlation-based feature selection* technique to select relevant features from the training set to train the base models. A total of 10 features were selected for the projects Ant v1.7 and Prop v1.0, and 22 features were selected for the projects Eclipse v2.0 and v3.0.

Since there is a high class imbalance in the datasets, library *DMwR* is used to apply SMOTE(Synthetic Minority Oversampling Technique) on the training sets to generate synthetic data for minority classes, in this case bug-prone modules. The meta-model XGBoost for the super learner is implemented using package *mlr*, which is similar to the package *caret.* The advantage of using *mlr* is that it has customized options to tune hyperparameter for XGBoost. Also, XGBoost performs well with sparse data, and so for the encoding of the input data to XGBoost, the function *createDummyFeatures* from the MLR package is used.

The package *ROCR* is used to plot the ROC(Receiver Operating Characteristic) curve for visually evaluating the performance of the models. Also, MS Excel is used to record the results from the experiment. To further generalize the results, each experiment is conducted three times, and the mean of the outcomes is considered as the final performance of the models.

To build the Super Learner, stacked generalization technique put forward by Breiman (1996) is used as seen in Figure 5. The following steps are implemented:

1. The base models SVM, Logistic Regression and Decision tree are tuned and trained on the training set.

2. Using the fitted base models, predictions are performed on the validation and test sets.

3. The predictions made on the validation set by each base model is stacked along with the actual value, which becomes the training set for the meta-model XGBoost.

4. Similarly, the predictions on the test set by each base model is stacked to form the testing set for the meta-model.

5. The meta-model is then trained using the stacked data from Step 3.

6. Finally, the testing set created in Step 4 is used to perform the final prediction of the Super Learner.

The performance of individual base-learners and the Super Learner is evaluated for each project. A baseline model is first created without handling the class imbalance in the training sets. Logistic Regression is implemented using the *Caret* package in R. The family is set to *binomial* as the target or response variable is binary in nature. The Decision tree model is also trained using the *Caret* package's *rpart* method, and its performance is tuned using the *cp*(Complexity) parameter. The final base learner, SVM, is implemented using the *svmRadial* method of the *Caret* package. The training data is first scaled and centered using the method *preProc* before fitting the model. The hyperparameter *Sigma* (σ) and *c*(cost of misclassification) are tuned to improve the performance of the model further. For implementing the meta-model XGBoost, *MLR* library is used. Since the predictions from the base learners are fed as input to the XGBoost model, it is converted into sparse data using *createDummyFeatures* method. The parameters *min child weight, max depth, colsamplebytree, subsample, eta* are tuned to get the optimal model.

The proposed model also deals with the handling of class imbalance in software fault prediction problem. For this, the technique *SMOTE* is applied using the *DMwR* package to the training set. This creates synthetic values for buggy modules and improves the representation of minority classes in the data. The above-mentioned steps are repeated after handling the class imbalance in the training sets to build the new models. A 10-fold cross validation technique is also applied to reduce the risk of biasing or over fitting of the data. These predictions are stacked and fed as input to the XGBoost model and the final predictions on the test set are recorded. The iteration with the balanced set is also performed three times to generalize the results. The results of the baseline and proposed model are then compared using different evaluation metrics as discussed in the next section.

6 Evaluation

6.1 Performance of classifiers on unbalanced data

A baseline model for the individual classifiers and the Super learner is built using unbalanced data. It can be observed from Table 2 that the performance of individual classifiers varies on different projects. For project Eclipse v2.0, the SVM classifier outperforms Logistic Regression and Decision Tree with an AUC score of 0.83 and an *accuracy* of 89%. Although, the *recall* value for the Decision Tree classifier is 47% slightly better than SVM. Similarly, for project Eclipse v3.0 and Prop v1.0, the performance of the SVM classifier is better with an AUC score of 0.83 and 0.79, respectively. It can be seen that the *recall* value of Decision Tree was better in both the projects with 35% and 26% respectively. The *recall* score of SVM for Prop v1.0 is quite low, with a rate of 9%. For project Ant v1.7, the performance of Logistic Regression is better than the other base learners with an AUC score of 0.75 and an *accuracy* of 81%.

The performance of the Super Learner for all the projects is in the range of 0.7 to 0.85 for AUC and 80-90% in terms of *accuracy*. Thus the performance of the Super Learner is better or at least competitive when compared to the individual classifiers except for the poor *recall* rate of 7% for the project Prop v1.0.

Project	Models	Performance measures		
		AUC	*accuracy*	*recall*
Eclipse v2.0	Logistic Regression	0.78	88%	31%
	SVM	0.83	89%	32%
	Decision Tree	0.75	87%	47%
	Super Learner	**0.8**	**89%**	**34%**
Eclipse v3.0	Logistic Regression	0.78	87%	24%
	SVM	0.83	85%	23%
	Decision Tree	0.76	87%	35%
	Super Learner	**0.75**	**85%**	**36%**
Prop v1.0	Logistic Regression	0.67	84%	26%
	SVM	0.79	85%	9%
	Decision Tree	0.67	84%	26%
	Super Learner	**0.77**	**86%**	**7%**
Ant v1.7	Logistic Regression	0.75	81%	37%
	SVM	0.7	81%	40%
	Decision Tree	0.69	79%	45%
	Super Learner	**0.73**	**81%**	**37%**

Table 2: Evaluation of classifiers - Unbalanced data (Baseline)

6.2 Performance of classifiers on balanced data

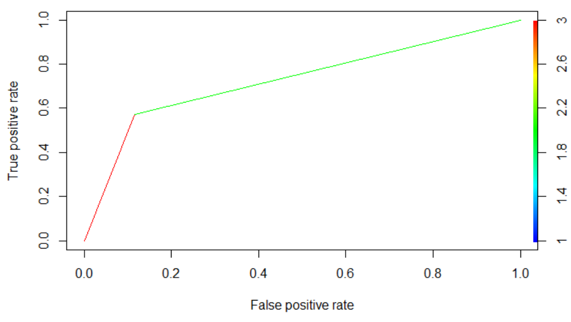
Now, the same experiment is repeated on a balanced training set for all the projects to evaluate the proposed model. Similar to the previous results it can be observed from Table 3 that the performance of individual classifiers is different for each of the projects under consideration. For the project Eclipse v2.0, the SVM classifier outperforms Logistic Regression and Decision Tree with an *AUC* score of 0.69, *accuracy* of 81% and a *recall* rate of 70%. Similarly, for project Eclipse v3.0 and Ant v1.7, the performance of the Logistic Regression classifier is better with an *AUC* score of 0.68 and 0.71, respectively. Although it can be seen that the *recall* rate of SVM is better for Eclipse v3.0 at 78%.For project Prop v1.0, the Decision Tree classifier performs the best compared to other base classifiers, with an *AUC* score of 0.63, *accuracy* of 80% and *recall* rate of 52%.

The *AUC* value of the Super Learner for all the projects is in the range of 0.7 to 0.75 better than any of the individual base classifiers. The *accuracy* achieved by the Super Learner is also the highest for all the projects with a range of 80-88%. The *recall* rate for the Super Learner is competitive when compared to the individual base classifiers.

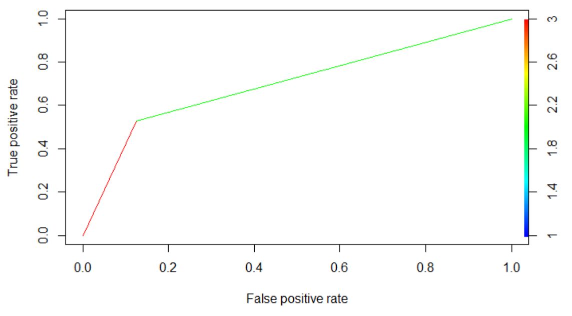
To visually evaluate the performance of the base learners and the combined Super Learner model, the *ROC* curve is plotted as seen in Figure 6. The performance of a classifier is acceptable if the curve is closer to the point(0,1) on the Y-axis. However, it can be difficult to differentiate between the performance of multiple classifiers using ROC alone, hence the *AUC* score is calculated to further interpret the same as can be seen in Table 2 and Table 3.

Project	Models	Performance measures		
		AUC	*accuracy*	*recall*
Eclipse v2.0	Logistic Regression	0.68	82%	65%
	SVM	0.69	81%	70%
	Decision Tree	0.66	82%	69%
	Super Learner	**0.75**	**88%**	**62%**
Eclipse v3.0	Logistic Regression	0.68	82%	53%
	SVM	0.59	63%	78%
	Decision Tree	0.65	81%	44%
	Super Learner	**0.73**	**86%**	**56%**
Prop v1.0	Logistic Regression	0.58	77%	37%
	SVM	0.62	77%	50%
	Decision Tree	0.63	80%	52%
	Super Learner	**0.7**	**84%**	**42%**
Ant v1.7	Logistic Regression	0.71	80%	59%
	SVM	0.69	78%	58%
	Decision Tree	0.69	78%	53%
	Super Learner	**0.67**	**81%**	**53%**

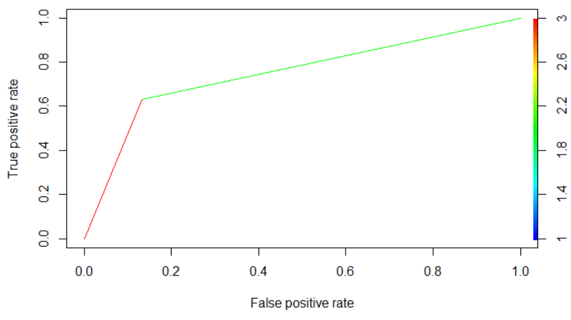
Table 3: Evaluation of classifiers - Balanced data



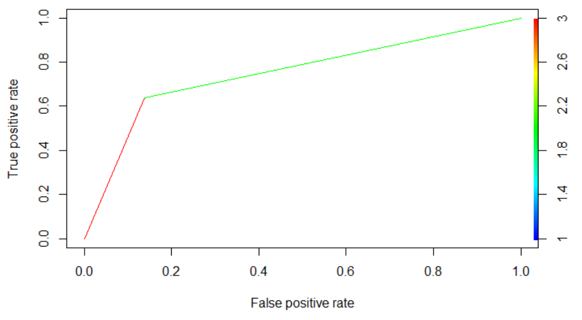
(a) Logistic Regression



(b) Support Vector Machine



(c) Decision Tree



(d) Super Learner

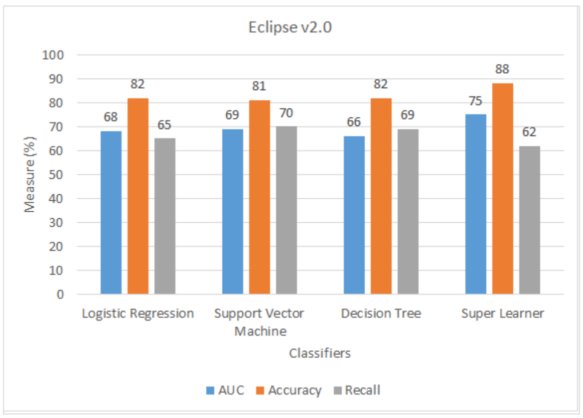
Figure 6: ROC Curve of the Base Learners and Super Learners for Project Ant v1.7

6.3 Discussion

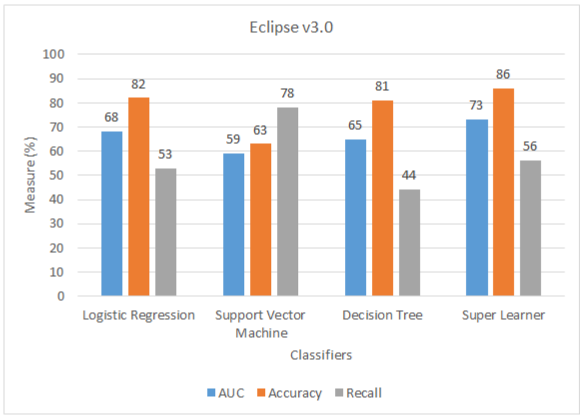
On observing the results in Table 2 and Table 3 it can be concluded that there is a significant improvement in the *recall* value of all classifiers after handling the class imbalance in the training set. Thus it can be affirmed that resampling the minority class, i.e., the bug-prone modules using SMOTE during the training phase can significantly improve the *recall* performance of the classifiers.

It is also evident from Table 3 that the performance of the proposed Super Learner exceeds that of the individual base learners. The AUC score and *accuracy* of the proposed Super Learner using XGBoost is higher than the individual classifiers such as SVM, LR and DT. The *recall* value for the Super Learner is better than DT and LR classifier for Eclipse v3.0. Similarly, the recall value for the Super Learner is better than LR classifier for Prop v1.0 at 42%.

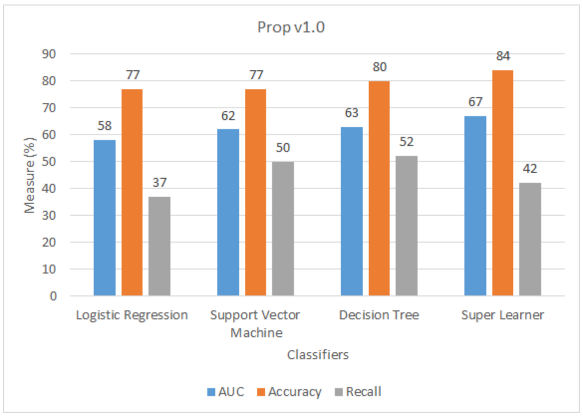
It can also be observed that the base classifiers provided varied predictions, and none of them performed consistently across different projects. However, the proposed Super learner classifier provided better predictions in terms of both AUC and *accuracy* and competitive *recall* value compared to the base classifiers across the projects under consideration. The Figure 7 shows the comparison of individual classifiers and the proposed Super Learner for performance parameters such as AUC, *accuracy* and *recall*.



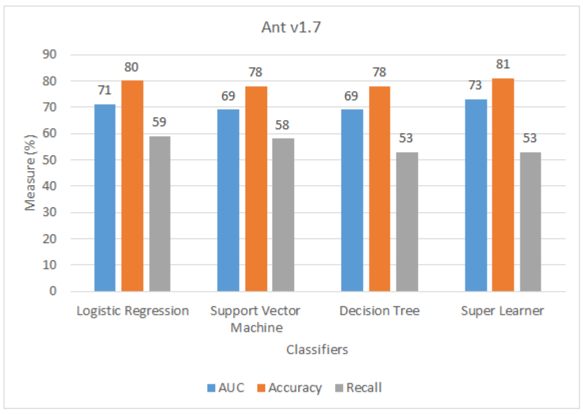
(a) Eclipse v2.0



(b) Eclipse v3.0



(c) Prop v1.0



(d) Ant v1.7

Figure 7: Results: Performance comparison of the models across projects

7 Conclusion

This research has presented the implementation of Super Learner based classifier for Software Bug prediction using a combination of four different machine learning techniques such as Logistic Regression, Support Vector Machines, Decision Tree and XGBoost. The experiments were performed on four different project datasets.

The findings suggest the use of class imbalance techniques such as SMOTE significantly improves the *recall* performance of the machine learning models. It is also observed that the performance of individual base learners was not consistent across all the projects under consideration. A single classifier may produce good predictions for one project and perform poorly for the other. Therefore, using the presented Super Learner can help generalize the models so that the performance is consistent across software projects as seen from the results in this research. Thus the presented Super Learner combines the prediction capabilities of multiple base learners and provides more robust and generalized performance for software bug prediction. The presented approach is a valuable addition to the existing works in the domain of Software bug prediction as it can be further adopted for different projects having similar metrics with budget and time constraints, and where end-to-end testing is difficult.

7.1 Future Work

Although the current model provides a good fit and has promising scores for all evaluation metrics, the Super Learner may need to be evaluated further for larger data sets. Currently, the Super Learner consists of only two layers, therefore, in future more layers can be added to further optimize the model. In addition, the presented model can also be analyzed for cross-project fault prediction as it can be interesting to find its potential use for new projects when no historical data is available. Also, different combinations for the base learners can be further explored to improve the prediction capabilities.

Acknowledgement

I would like to express my gratitude to my supervisor Dr Vladimir Milosavljevic for his constant feedback and valuable advice throughout the implementation of this research project.

References

Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M. and Giacinto, G. (2016). Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification, *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy - CODASPY '16* pp. 183–194.

Aljamaan, H. I. and Elish, M. O. (2009). An empirical study of bagging and boosting ensembles for identifying faulty classes in object-oriented software, *2009 IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2009 - Proceedings* pp. 187–194.

Arora, I., Tetarwal, V. and Saha, A. (2015). Open issues in software defect prediction, *Procedia Computer Science* **46**(Icict 2014): 906–912.

Boehm, B. and California, S. (2001). Software Defect Reduction Top 10 List, pp. 135–137.

Breiman, L. (1996). Stacked regressions, *Machine Learning* **24**(1): 49–64.

Chen, L. I. N., Fang, B. I. N. and Shang, Z. (2016). SOFTWARE FAULT PREDICTION BASED ON ONE-CLASS SVM, *2016 International Conference on Machine Learning and Cybernetics (ICMLC)* **2**: 1003–1008.

Chen, M. b. and Ma, Y. c. (2015). An empirical study on predicting defect numbers, *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE* **2015-Janua**: 397–402.

Chen, Z., Jiang, F., Cheng, Y., Gu, X., Liu, W. and Peng, J. (2018). XGBoost Classifier for DDoS Attack Detection and Analysis in SDN-Based Cloud, *Proceedings - 2018 IEEE International Conference on Big Data and Smart Computing, BigComp 2018* pp. 251–256.

Cruz, A. E. C. and Ochimizu, K. (2009). Towards logistic regression models for predicting fault-prone code across software projects, *2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009* pp. 460–463.

D'Ambros, M., Gall, H., Lanza, M. and Pinzger, M. (2008). Analysing software repositories to understand software evolution, *Software Evolution*.

D'Ambros, M., Lanza, M. and Robbes, R. (2010). An extensive comparison of bug prediction approaches, *Proceedings - International Conference on Software Engineering* pp. 31–41.

Denaro, G., Pezzè, M. and Morasca, S. (2003). Towards Industrially Relevant Fault-Proneness Models, *International Journal of Software Engineering and Knowledge Engineering* **13**(04): 395–417.

Dhaliwal, S. S., Nahid, A. A. and Abbas, R. (2018). Effective intrusion detection system using XGBoost, *Information (Switzerland)* **9**(7).

El Eman, K. and Melo, W. (2001). The Prediction of Faulty Classes Using Object-oriented, **56**(November).

Elish, K. O. and Elish, M. O. (2008). Predicting defect-prone software modules using support vector machines, *Journal of Systems and Software* **81**(5): 649–660.

Elish, M. O., Aljamaan, H. and Ahmad, I. (2015). Three empirical studies on predicting software maintainability using ensemble methods, *Soft Computing* **19**(9): 2511–2524.

Fayyad, U., Piatetsky-Shapiro, G. and Smyth, P. (1996). The kdd process for extracting useful knowledge from volumes of data, *Commun. ACM* **39**(11): 27–34.

Gayathri, M. and Sudha, A. (2014). Software Defect Prediction System using Multilayer Perceptron Neural Network with Data Mining, *International Journal of Recent Technology and Engineering* (32): 2277–3878.

Gondra, I. (2008). Applying machine learning to software fault-proneness prediction, *Journal of Systems and Software* **81**(2): 186–195.

Güne Koru, A. and Liu, H. (2007). Identifying and characterizing change-prone classes in two large-scale open-source products, *Journal of Systems and Software* **80**(1): 63–73.

Kagdi, H., Collard, M. L. and Maletic, J. I. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution.

Kasurinen, J. (2010). Elaborating software test processes and strategies, *ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation* pp. 355–358.

Kaur, A. and Kaur, I. (2018). An empirical evaluation of classification algorithms for fault prediction in open source projects, *Journal of King Saud University - Computer and Information Sciences* **30**(1): 2–17.

Kelly, J. C., Sherif, J. S. and Hops, J. (1992). An analysis of defect densities found during software inspections, *Journal of Systems and Software* **17**(2): 111 – 117.

Khoshgoftaar, T. M., Allen, E. B., Jones, W. D. and Hudepohl, J. P. (1999). Data mining for predictors of software quality, *International Journal of Software Engineering and Knowledge Engineering* **9**: 547–563.

Khoshgoftaar, T. M. and Gao, K. (2009). Feature selection with imbalanced data for software defect prediction, *8th International Conference on Machine Learning and Applications, ICMLA 2009* pp. 235–240.

Khoshgoftaar, T. M. and Seliya, N. (2002). Software quality classification modeling using the sprint decision tree algorithm, *14th IEEE International Conference on Tools with Artificial Intelligence, 2002. (ICTAI 2002). Proceedings.*, pp. 365–374.

Khoshgoftaar, T. M. and Seliya, N. (2003). Software Quality Classification Modeling Using the SPRINT Decision Tree Algorithm, *International Journal on Artificial Intelligence Tools* **12**(03): 207–225.

Kim, S., Zimmermann, T., Whitehead, E. J. and Zeller, A. (2007). Predicting faults from cached history, *Proceedings - International Conference on Software Engineering* pp. 489–498.

Laradji, I. H., Alshayeb, M. and Ghouti, L. (2015). Software defect prediction using ensemble learning on selected features, *Information and Software Technology* **58**: 388–402.

Ma, W., Zhou, Y., Xu, B., Chen, L. and Yang, Y. (2015). Empirical analysis of network measures for effort-aware fault-proneness prediction, *Information and Software Technology* **69**: 50–70.

Mahaweerawat, A., Sophatsathit, P. and Lursinsap, C. (2002). Software fault prediction using fuzzy clustering and radial-basis function network.

Malhotra, R. (2014). Comparative analysis of statistical and machine learning methods for predicting faulty modules, *Applied Soft Computing Journal* **21**: 286–297.

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction, *Applied Soft Computing* **27**: 504–518.

Malhotra, R., Bahl, L., Sehgal, S. and Priya, P. (2017). Empirical comparison of machine learning algorithms for bug prediction in open source software, pp. 40–45.

Mauša, G., Grbac, T. G., Bogunović, N. and Bašić, B. D. (2015). Rotation forest in software defect prediction, *CEUR Workshop Proceedings* **1375**(Sqamia): 35–43.

Menzies, T., Dekhtyar, A., Distefano, J. and Greenwald, J. (2007). Problems with precision: A response to "Comments on 'data mining static code attributes to learn defect predictors'", *IEEE Transactions on Software Engineering* **33**(9): 637–640.

Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y. and Bener, A. (2010). Defect prediction from static code features: Current results, limitations, new approaches, *Automated Software Engineering* **17**(4): 375–407.

Mertik, M., Lenic, M., Stiglic, G. and Kokol, P. (2006). Estimating Software Quality with Advanced Data Mining Techniques, **00**(c): 19–19.

Misirli, A. T., Bener, A. B. and Turhan, B. (2011). An industrial case study of classifier ensembles for locating software defects, *Software Quality Journal* **19**(3): 515–536.

Moustafa, S., ElNainay, M. Y., Makky, N. E. and Abougabal, M. S. (2018). Software bug prediction using weighted majority voting techniques, *Alexandria Engineering Journal* **57**(4): 2763–2774.

Olague, H. M., Etzkorn, L. H., Gholston, S. and Quattlebaum, S. (2007). Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly Iterative or agile software development processes, *IEEE Transactions on Software Engineering* **33**(6): 402–419.

Öztürk, M. M. (2017). Which type of metrics are useful to deal with class imbalance in software defect prediction?, *Information and Software Technology* **92**: 17–29.

Polley, E. C. and van der Laan, M. J. (2010). Super Learner in Prediction, *U.C. Berkeley Division of Biostatistics Working Paper 266* pp. 1–19.

Qi Wang, Bo Yu and Jie Zhu (2004). Extract rules from software quality prediction model based on neural network, *16th IEEE International Conference on Tools with Artificial Intelligence*, pp. 191–195.

Rathore, S. S. and Kumar, S. (2016). An empirical study of some software fault prediction techniques for the number of faults prediction, *Soft Computing* **21**(24): 7417–7434.

Rathore, S. S. and Kumar, S. (2017). Towards an ensemble based system for predicting the number of software faults, *Expert Systems with Applications* **82**: 357–382.

Siers, M. J. and Islam, M. Z. (2015). Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem, *Information Systems* **51**(May 2018): 62–71.

Sunghun, K., Whitehead Jr., E. J. and Yi, Z. (2008). Classifying software changes: clean or buggy?, *IEEE Transactions on Software Engineering* **34**(2): 181–196.

Tricentis (2018).
URL: *https://www.tricentis.com/news/tricentis-software-fail-watch-finds-3-6-billion-people-affected-and-1-7-trillion-revenue-lost-by-software-failures-last-year/*

Twala, B. (2011). Predicting software faults in large space systems using machine learning techniques, *Defence Science Journal* **61**(4): 306–316.

Wang, H., Khoshgoftaar, T. M. and Napolitano, A. (2014). Stability of filter- and wrapper-based software metric selection techniques, *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration, IEEE IRI 2014* pp. 309–314.

Wu, W., Wang, G., Peng, Y., Shi, Y. and Lou, G. (2011). Ensemble of Software Defect Predictors: an Ahp-Based Evaluation Method, *International Journal of Information Technology & Decision Making* **10**(01): 187–206.

Xing, F., Guo, P. and Lyu, M. R. (2005). A novel method for early software quality prediction based on support vector machine, *Proceedings - International Symposium on Software Reliability Engineering, ISSRE* **2005**: 213–222.

Yousef, A. H. (2015). Extracting software static defect models using data mining, *Ain Shams Engineering Journal* **6**(1): 133–144.

Zhang, Y., Huang, Q., Ma, X., Yang, Z. and Jiang, J. (2016). Using multi-features and ensemble learning method for imbalanced Malware classification, *Proceedings - 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 10th IEEE International Conference on Big Data Science and Engineering and 14th IEEE International Symposium on Parallel and Distributed Processing with Applications, IEEE TrustCom/BigDataSE/ISPA 2016* pp. 965–973.

Zheng, J. (2010). Cost-sensitive boosting neural networks for software defect prediction, *Expert Systems with Applications* **37**(6): 4537–4543.

Zimmermann, T., Premraj, R. and Zeller, A. (2007). Predicting defects for eclipse, *Proceedings - ICSE 2007 Workshops: Third International Workshop on Predictor Models in Software Engineering, PROMISE'07* .

dirtytalk