Carnegie Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

REPORT

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Master of Science

TITLE Computational Study of Data Preprocessing Methods and Machine Learning Classification Algorithms for Software Bug Prediction

PRESENTED BY Zhenjie Jia

ACCEPTED BY THE DEPARTMENT OF

Chemical Engineering

N. Sahinidis
NIKOLAOS SAHINIDIS, PROFESSOR AND ADVISOR

12/5/2019
DATE

ANNE S. ROBINSON, PROFESSOR AND DEPARTMENT HEAD

12/11/2019
DATE

Computational Study of Data Preprocessing Methods and Machine Learning Classification Algorithms for Software Bug Prediction

Zhenjie Jia

Carnegie Mellon University

Master's Project Report

2019

Advisor: Prof. Nikolaos V. Sahindis,

Prof. Nikolaos Ploskas

Contents

List of Figures

List of Tables

1. Abstract

This is a joint work with Xinyi He.

Predicting bugs on software is particularly useful for software developers since checking a software manually can take a lot of effort. The datasets which record the features and issues of software are usually imbalanced and only a small amount of data indicates buggy classes. The unique feature of datasets in this field results in high accuracy and low recall rate. This work presents the data preprocessing, model selection, training process and result analysis of software bugs prediction based on D'Ambros et al. dataset (D'Ambros, 2010) and Zimmermann et al. dataset (Zimmermann, 2007). We adapted sampling methods including over sample, under sample and syntactic sample to balance the dataset and then combined these methods with six different machine learning algorithms to build a bug prediction model. Experiments show that logistic regression, random forest and neural network will dramatically increase the recall rate by 20%-50% after syntactic sampling the dataset.

Keywords: bug prediction; software engineering; machine learning; classification; data preprocessing

2. Introduction

Software has an increasing influence in the world. It is very essential to build durable software with low cost. Statistically, the effort in finding and fixing bugs in a software will consume near 80% of the budget of a software development (Tassey, 2002). Therefore, the defect prediction is very important to improve the software quality and reduce the effort to find the bugs as the size of the software is getting larger. From the 1970s', many researchers have been working on using auto-detection methods to predict bugs in the software. Nowadays, with the trend of machine learning, researchers applied many machine learning algorithms on this topic. The software defect or bug prediction can be seen as a classification problem of whether a class in the software has bugs or not. The general process of software defect prediction requires the following steps:

1) Feature extraction: to represent the status of software of different languages, we need software's feature matrix to be used as the training set.
2) Model training: using different ML methods like random forest, logistic regression, neural networks, etc. to train the model.
3) Result comparison: testing on different software or the same software's latest version.

Meanwhile, this field of research is now facing the following challenges:

1) The lack of datasets: only a few open source software have a public and clear bug tracker that can be used to generate the dataset.

2) The dataset is imbalanced: the number of data with “bugs” label in datasets is about 15%, while the rest of the data in a dataset all have the “zero bugs” label.
3) Different projects’ feature metrics are different.
4) The data in the datasets vary a lot: many features have zero values, some features’ numbers are big while some are extremely small.

The project goal is to find an improved process to predict the bugs in the software. In order to solve the dataset’s imbalance problem and improve the prediction results, this paper used over sample, under sample and syn sample methods, all combined with standardization to preprocessing the data. Then, we tried different machine learning methods on the software defect model and made a comparison of the results. Three algorithms: random forest, logistic regression and neural network are performing well. In the end, the accuracy we got in this project is near other researchers’ result on the same dataset. However, the recall rate we got is near 60% which is much higher than others’ results.

Section 2 shows other researchers work on using machine learning for bug prediction. Section 3 are simple introductions of the machine learning algorithms we used in this study. Section 4 presents our results of different algorithms combined with different data preprocessing methods on different situations including testing within the same project, testing cross the different projects and cross different versions projects. Section 5 includes our conclusions.

3. Literature review

The process of predicting bugs on software using automatic procedures starts at the beginning of the computer science history. Back to the 1970s, Akiyama (1971) first attempted to use software size-based metrics and regression models in order to predict the defects of software. However, the effect of bug prediction remained not good enough for a long time because of the restrictions on source data and learning model. Kamei & Shihab (2016) summarized that challenges on this topic back to early 2000s: i) lack of data since there weren’t much open source software and the bugs are always given at subsystem or file levels, not the function level; ii) lack of variety of independent and dependent variables. The most independent variables used are size-based metrics, especially code complexity metrics, while the most dependent variables are post-release defects.

In recent years, with the trend of the machine learning, the software defects prediction is combined closely with many machine learning algorithms. Tantithamthavorn et al. (2016) used an off-the-shelf automated parameter optimization technique on 11 machine learning algorithms including Naive Bayes, KNN, SVM, neural network on data sets from different sources. They proved that their automated parameter optimization technique can increase the AUC 40% than the default classifiers.

Toth et al. (2016) selected 15 Java projects from GitHub to construct a public bug database. They matched the already known and fixed bugs with the corresponding source code elements (classes and files) and calculated a wide set of product metrics on these elements. After creating the desired bug database, they investigated whether the built database is usable for bug prediction. They used 13 machine learning algorithms to address this research question and finally they achieved an F-measure values between 0.7 and 0.8.

Xia et al. (2013) evaluated the effectiveness of various supervised learning algorithms to predict whether or not a bug report would be reopened. They chose seven state-of-the-art classical supervised learning algorithms in machine learning literature, including KNN, SVM, SimpleLogistic, Bayesian Network, Decision Table, CART and LWL, and three ensemble learning algorithms, including AdaBoost, Bagging and Random Forest, and evaluated their performance in predicting reopened bug reports. The experimental results showed that among the algorithms, Bagging and Decision Table (IDTM) achieved the best performance. They achieved accuracy scores of 92.91% and 92.80%, respectively, and reopened bug reports F-Measure scores of 0.735 and 0.732, respectively. These results improved the reopened bug reports F-Measure of the state-of-the-art approaches proposed by Shihab et al. by up to 23.53%.

Zhang et al. (2016) used a spectral Random Forest classifier which calculated the Laplacian matrix and perform the eigen decomposition on the matrix to normalize the matrix. They tested it in D'Ambros dataset (2010) and achieved 81% and 71% of AUC cross projects and 87% and 78% of AUC when within the projects of JDT and PDE, respectively. Couto et al. (2014) built a model that will use six alarm threshold functions to select the variations in the metrics that may have contributed to the occurrence of defects. They removed the classes with zero defects with min activation function and got 61% precision, 53% recall for JDT, 27% precision and 54% of recall for PDE.

Osman et al. (2017) revealed that tuning model hyperparameters has a statistically significant positive effect on the prediction accuracy of the models. The prediction accuracy was improved by up to 20% for KNN and by up to 10% for SVM. Kamei et al. (2010) showed that package-level predictions were not more effective than file-level predictions and the effectiveness of package-level predictions can improve if we perform the predictions at the file-level then lift it to the package-level instead of collecting all metrics at the package-level. However, the new model still did not outperform file-level predictions when considering the quality assurance efforts.

Shivaji et al. (2009) proposed a feature selection technique applicable to classification-based bug prediction. Zimmermann et al. (2007) conducted a work on the code base of the Eclipse programming environment. They extended their data with common complexity metrics and the counts of syntactic elements. Then, they built logistic regression models for the Eclipse bug data set to predict whether or not files/packages have post-release defects. In their final version of the paper, for file level with Eclipse version 3.0, the accuracy scored, recall score and precision score are about 0.869, 0.224 and 0.675 while for package level with Eclipse version 3.0, the accuracy scored, recall score and precision score are about 0.855, 0.789 and 0.892.

Table 1 shows all the popular dataset that have been used in previous research. The first column shows the number of the dataset, the second column shows the name of the dataset and the third column shows the link of the dataset.

Table 1: Bug prediction datasets

Dataset No	Dataset	Link
1	D'Ambros et al's dataset (D'Ambros, 2010)	http://bug.inf.usi.ch/index.php
2	Zimmermann's dataset (Zimmermann, 2007)	https://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/
3	NASA's dataset [18]	http://promise.site.uottawa.ca/SERepository/datasets-page.html
4	Jureczko et al's dataset [13]	http://snow.iiar.pwr.wroc.pl:8080/MetricsRepo/
5	Sunghun et al's dataset [14]	https://code.google.com/archive/p/hunkim/wikis/HandlingNoise.wiki
6	SR Mining Challenge's dataset [10]	http://2008.msrconf.org/challenge/

Table 2 shows the summary of some previous studies. The first column shows the authors of each paper, the second column shows the specific techniques used, the third column shows the dataset used and the fourth column shows the result.

Table 2: Literature review summary

Paper	Techniques used	Dataset	Results
Tantithamthavorn et al. (2016)	An off-the-shelf automated parameter optimization technique on 11 machine learning algorithms	1, 2, 3, 4, 5	They increase 40% the AUC rate than the default classifiers
Toth et al. (2016)	13 machine learning algorithms	Java projects from GitHub	They achieved F-measure values between 0.7 and 0.8 and very high and promising bug coverage values (up to 100 %)
Xia et al. (2013)	KNN, SVM, SimpleLogistic, Bayesian Network, Decision Table, CART, LWL, AdaBoost, Bagging and Random Forest.	dataset from [15][16]	Bagging and Decision Table (IDTM) show the best performance, their accuracy scores are 92.91% and 92.80% respectively, and F-Measure scores for reopened bug reports are 0.735 and 0.732 respectively.
Zhang et al. (2016)	random forest, naive Bayes, logistic regression, decision tree, and logistic model tree with spectral clustering	1, 3, 4	The median accuracy result they got when crossing the project is 70%, and the 80% accuracy within the projects.
Couto et al. (2014)	multilayer perceptron with different activation functions like :min, max, mean, median	1	They reached an average precision ranging from 28% (EclipsePDE UI) to 58% (Eclipse JDT Core) and a median precision ranging from 31% (Eclipse PDE UI) to 58% (Eclipse JDT Core).
Osman et al. (2017)	support vector machines (SVM), and an	1	The prediction accuracy was improved by up to 20% in KNN and by up to 10% in SVM

	implementation of the k-nearest neighbors algorithm		
Kamei et al. (2010)	regression model, regression tree and random forest	6	At the file-level, random forest produce the best predictor performance compared to linear models and regression trees. Package-level predictions are less effective than file-level predictions. When they test or review 20% of all modules based on the predicted fault density, they could detect almost 74% of faults using file-level models versus 62% of faults using package-level models.
Shivaji et al. (2009)	Naive Bayes and Support Vector Machine, using a feature selection technique	Software revision history for Apache, Columba, Gaim, etc. (No open source yet)[11]	The best accuracy scored, recall score and precision score and they achieve are about 0.91, 0.67 and 0.96 with Bayes F-measure Technique.
Zimmermann et al. (2007)	Logistic regression	2	For file level with version 3.0, the accuracy scored, recall score and precision score they get are about 0.869, 0.224 and 0.675.

4. Machine Learning algorithms

We experimented with seven machine learning algorithms in this study, including Naive Bayes, Logistic Regression, K-nearest Neighbors, Decision Tree, Random Forest, and Neural Networks. In this section, we will briefly introduce these classifiers.

4.1 Naive Bayes

Naive Bayes is a simple technique for constructing classifiers: models that assign class labels to problem instances, represented as vectors of feature values, where the class labels are drawn from a finite set. There is not a single algorithm for training such classifiers, but a family of algorithms based on a common principle: all naive Bayes classifiers assume that the value of a particular feature is independent of the value of any other feature, given the class variable. For example, a

fruit may be considered to be an apple if it is red, round, and about 10 cm in diameter. A naive Bayes classifier considers each of these features to contribute independently to the probability that this fruit is an apple, regardless of any possible correlations between the color, roundness, and diameter features.

For some types of probability models, naive Bayes classifiers can be trained very efficiently in a supervised learning setting. In many practical applications, parameter estimation for naive Bayes models uses the method of maximum likelihood; in other words, one can work with the naive Bayes model without accepting Bayesian probability or using any Bayesian methods. Figure 1 and Equations (1) - (4) show the derivation process of Bayes' theorem.

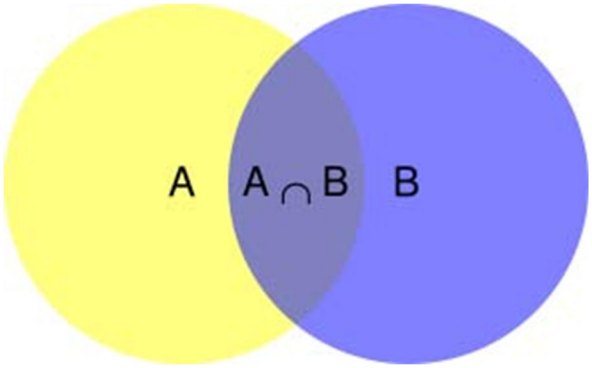

Figure 1: Conditional probability

In the case of an event B, the probability of occurrence of the event A is represented by $P(A|B)$

$$P(A|B) = P(A \cap B)/P(B) \quad (1)$$

$$P(A \cap B) = P(A|B)P(B) \quad (2)$$

In the same way, we can get:

$$P(A \cap B) = P(B|A)P(A) \quad (3)$$

Thus, we can get the foundation of the Naive Bayes Classifier (Bayes' theorem):

$$P(A|B) = P(B|A)P(A)/P(B) \quad (4)$$

4.2 Logistic regression

Logistic regression is used to model the probability of a certain class or event existing such as pass/fail, win/lose, alive/dead or healthy/sick. This can be extended to model several classes of events such as determining whether an image contains a cat, dog, lion, etc... Each object being detected in the image would be assigned a probability between 0 and 1 and the sum adding to one.

Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable, although many more complex extensions exist. In regression analysis, logistic regression is estimating the parameters of a logistic model. Mathematically, a binary logistic model has a dependent variable with two possible values, such as pass/fail which is

represented by an indicator variable, where the two values are labeled "0" and "1". In the logistic model, the log-odds for the value labeled "1" is a linear combination of one or more independent variables; the independent variables can each be a binary variable or a continuous variable. The corresponding probability of the value labeled "1" can vary between 0 and 1, hence the labeling; the function that converts log-odds to probability is the logistic function, hence the name. The unit of measurement for the log-odds scale is called a logit, from logistic unit, hence the alternative names. Analogous models with a different sigmoid function instead of the logistic function can also be used; the defining characteristic of the logistic model is that increasing one of the independent variables multiplicatively scales the odds of the given outcome at a constant rate, with each independent variable having its own parameter; for a binary dependent variable this generalizes the odds ratio.

The binary logistic regression model has extensions to more than two levels of the dependent variable: categorical outputs with more than two values are modeled by multinomial logistic regression, and if multiple categories are ordered, by ordinal logistic regression, for example the proportional odds ordinal logistic model. The model itself simply models probability of output in terms of input, and does not perform statistical classification, though it can be used to make a classifier, for instance by choosing a cutoff value and classifying inputs with probability greater than the cutoff as one class, below the cutoff as the other; this is a common way to make a binary classifier.

The function used is the Sigmoid Function, which is also called the Logistic Function and is shown in Equations (5) and (6). θ^T is the parameter, x is the input data, $h_\theta(x)$ gives us the probability that the output is 1 and $g(z)$ is the sigmoid function, where z always represents $\theta^T x$.

$$h_\theta(x) = g(\theta^T x) \qquad (5)$$

$$g(z) = \frac{1}{1 + e^{-z}} \qquad (6)$$

The cost function for logistic regression is shown in Equations (7), (8) and (9).

$$Cost(h_\theta(x), y) = -log(h_\theta(x)) \quad if\ y = 1 \qquad (7)$$

$$Cost(h_\theta(x), y) = -log(1 - h_\theta(x)) \quad if\ y = 0 \qquad (8)$$

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) \qquad (9)$$

Figure 2 shows an example of the Logistic Regression. The blue dots mean the positive samples, the red dots mean the negative samples and the green line represents the decision boundary.

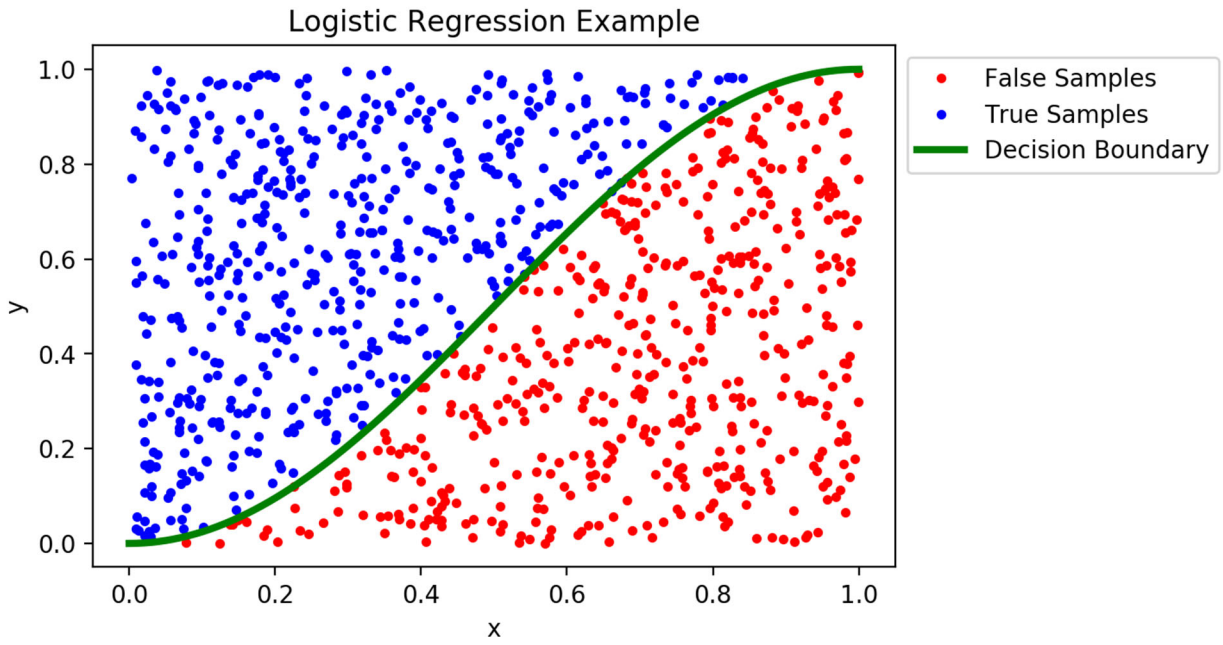

Figure 2: Logistic Regression

4.3 K-Nearest Neighbors

In pattern recognition, the K-Nearest Neighbors Algorithm (k-NN) is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. The output depends on whether k-NN is used for classification or regression. In k-NN classification, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbor.

KNN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until classification. Both for classification and regression, a useful technique can be to assign weights to the contributions of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones. For example, a common weighting scheme consists in giving each neighbor a weight of 1/d, where d is the distance to the neighbor. Figure 3 is an example of the classification result of the KNN classifiers. According to the distance from three different data points, this space is divided into 3 parts.

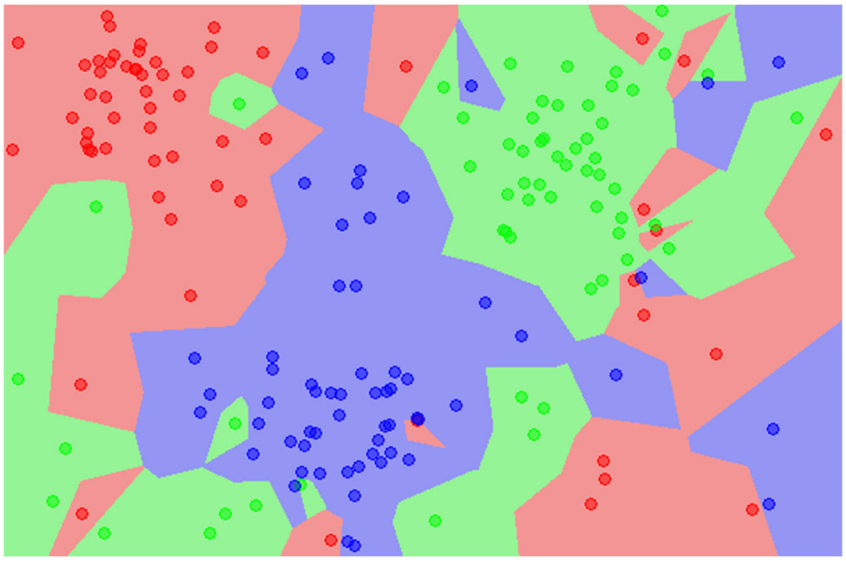

Figure 3: KNN (Karnin and Sadoughi, 2018)

4.4 Decision Tree

A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represent classification rules. In decision analysis, a decision tree and the closely related influence diagram are used as a visual and analytical decision support tool, where the expected values (or expected utility) of competing alternatives are calculated.

A decision tree consists of three types of nodes:

1) Decision nodes – typically represented by squares
2) Chance nodes – typically represented by circles
3) End nodes – typically represented by triangles

Decision trees are commonly used in operations research and operations management. If, in practice, decisions have to be taken online with no recall under incomplete knowledge, a decision tree should be paralleled by a probability model as a best choice model or online selection model algorithm. Another use of decision trees is as a descriptive means for calculating conditional probabilities. Figure 4 shows the basic structure of the decision tree. Each circle represents a node. The data will be classified through majority vote or Information entropy.

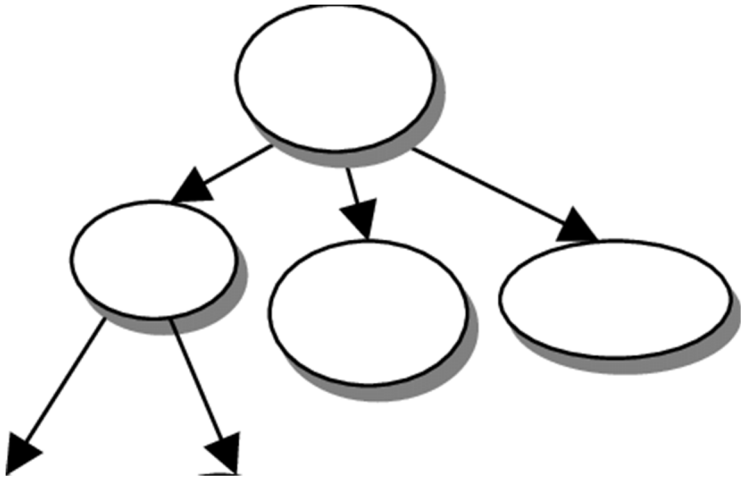

Figure 4: Decision Tree

4.5 Random forest

Random forest or random decision forest is an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to them training set. The first algorithm for random decision forests was created by Tin Kam Ho using the random subspace method, which, in Ho's formulation, is a way to implement the "stochastic discrimination" approach to classification proposed by Eugene Kleinberg. An extension of the algorithm was developed by Leo Breiman and Adele Cutler, who registered "Random Forest" as a trademark. The extension combines Breiman's "bagging" idea and random selection of features, introduced first by Ho and later independently by Amit and Geman in order to construct a collection of decision trees with controlled variance.

Figure 5 shows the main structure of Random forest. The random forest is actually a special bagging method that uses the decision tree as a model in bagging. First, the bootstrap method is used to generate different training sets. Then, for each training set, a decision tree is constructed. When the nodes find features to split, not all features can be found to maximize the indicators (such as information gain). Instead, a part of the features is randomly extracted from the features, and an optimal solution is found among the extracted features, applied to the nodes, and split. The method of random forest is based on bagging, which is the idea of integration. In fact, it is equivalent to sampling samples and features, so over-fitting can be avoided to some extent. The method of the prediction phase is the bagging strategy, majority voting.

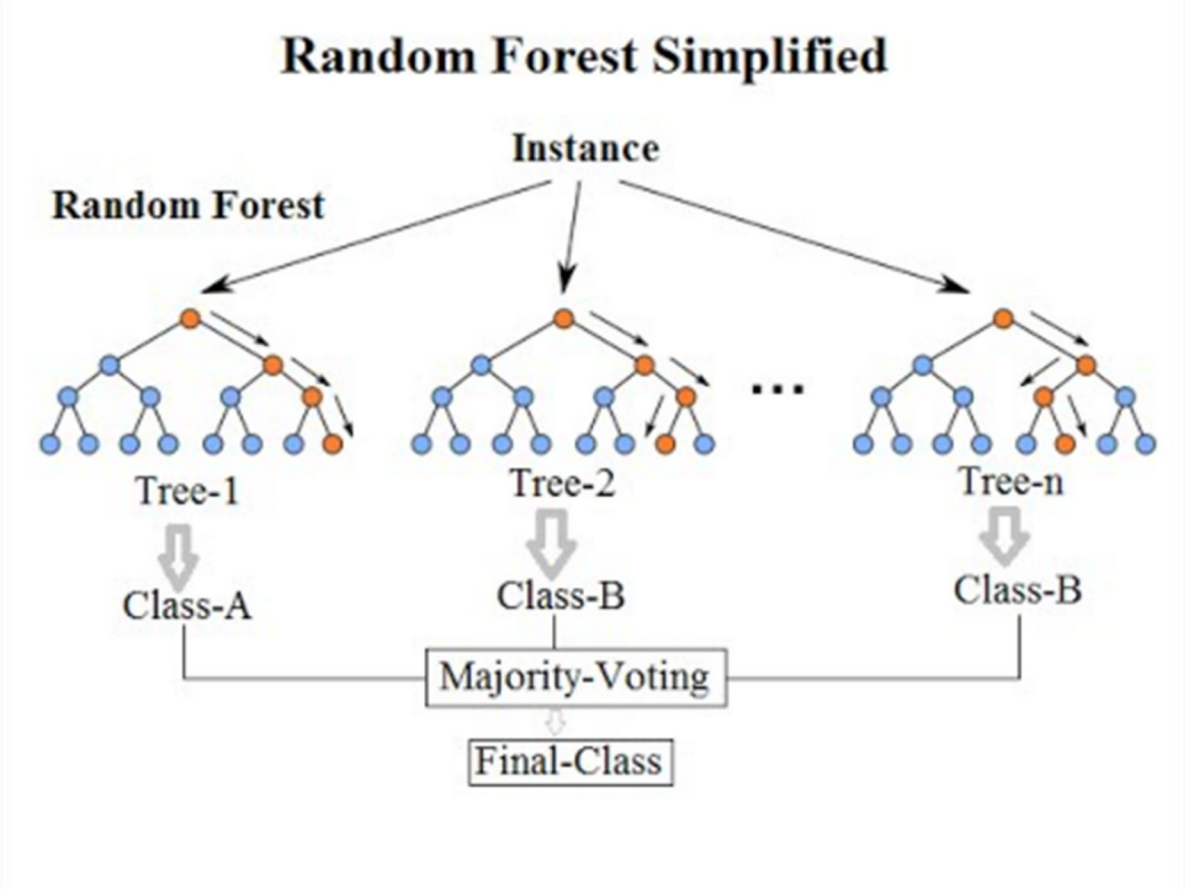

Figure 5: Random Forest (Koehrsen, 2017)

4.6 Multilayer perceptron (MLP)

Multilayer perceptron (MLP) is a class of feedforward artificial neural network. An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable. Multilayer perceptron is sometimes colloquially referred to as "vanilla" neural networks, especially when they have a single hidden layer. Figure 6 shows the main structure of Multilayer perceptron, which consists of input layer, hidden layers and output layer.

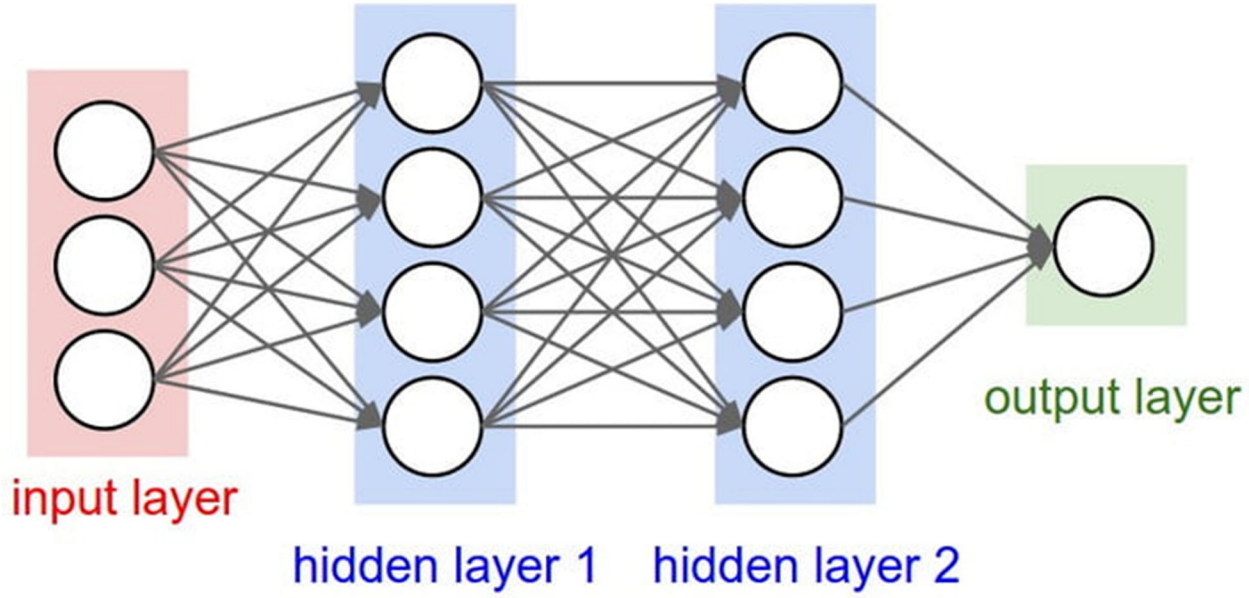

Figure 6: Multilayer perceptron (Venelin, 2017)

5. Experiments

5.1 Dataset

We used the D'Ambros's data set (D'Ambros, 2010) and Zimmermann's data set (Zimmermann, 2007) to test the model on different software projects and the last dataset to test the model on different versions of the same software.

D'Ambros's data set (D'Ambros, 2010)

The D'Ambros's dataset contains matrices of five different software, which are: Eclipse JDT Core (https://github.com/eclipse/eclipse.jdt.core), Eclipse PDE UI (https://github.com/eclipse/eclipse.pde.ui), Equinox Framework (https://github.com/eclipse/rt.equinox.framework), Lucene (https://github.com/apache/lucene-solr) and Mylyn (https://github.com/eclipse/egit-github). For each system the dataset includes the following pieces of information (D'Ambros, 2010):

- Biweekly versions of the systems parsed (with the inFusion tool) into object-oriented models, provided as mse files
- Historical information extracted from the cvs change log, including reconstructed transaction and links from transactions to model classes
- Value of 15 metrics computed from cvs change log data, for each class of the systems

- Values of 17 source code metrics (CK + 11 object-oriented metrics), for each version of each class
- Categorized (with severity and priority) post-release defect counts for each class.

We combined all the matrices provided on the website to form a matrix for each software that has 40 attributes and one target (buggy or not). Figure 7 shows the distribution of the data, including the number of data for each software and the percentage of the buggy classes and the non-buggy classes for each software. We can see that the largest percentage of buggy data is 39.81% and the smallest percentage of buggy data is only 9.26%. Three out of five software have lower than 30% buggy data.

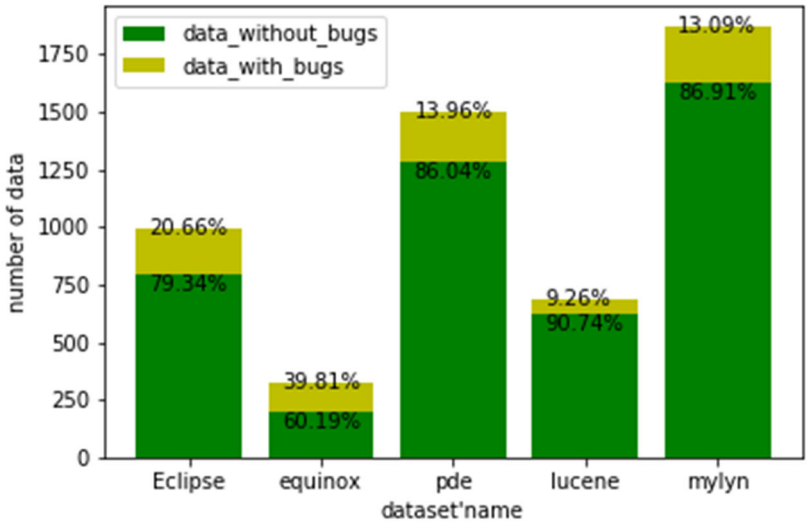

Figure 7: Distribution of D'Ambros's dataset (D'Ambros, 2010)

Zimmermann's data set (Zimmermann, 2007)

Zimmermann's data set contains three different versions of Eclipse software at file level. These datasets contain 175 attributes. One part of these attributes come from several complexity metrics like the average, maximum and sum of classes or interfaces in the project, while the other part of the attributes come from structure of abstract syntax tree of the files in the project. Figure 8 shows the distribution of the data, including the number of data for each version of the eclipse and the percentage of the buggy classes and the non-buggy classes for each version. From Figure 8, it is clear that buggy data in the three different versions of the dataset are all below 15%.

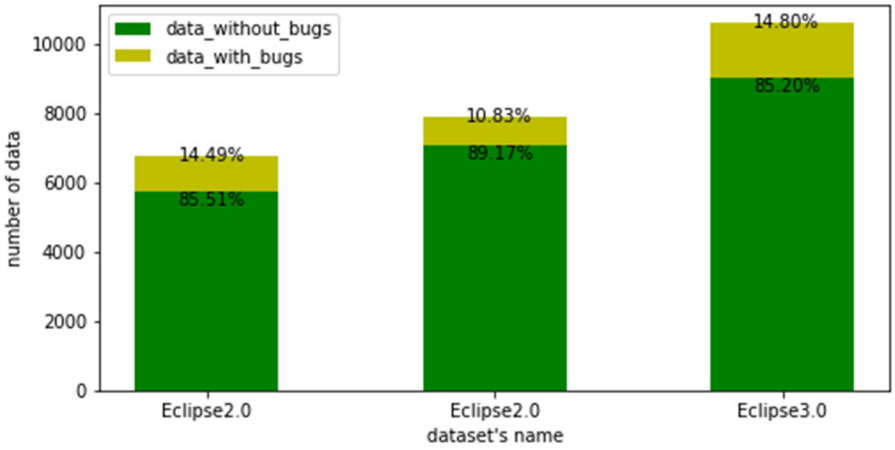

Figure 8: Distribution of Zimmermann's data set (Zimmermann, 2007)

5.2 Preprocessing

From the Section 4.1, we can see that the data without buggy is much larger than the data with buggy. This imbalanced distribution will make the classifiers to predict all the data are not buggy since it will get at least near 85% accuracy score. Other researchers' work is mainly focused on getting high accuracy score while the recall rate is below 20%. We think that in the bug prediction fields, the recall rate weighs much more than the accuracy score, since even the smallest bug can cause the crash of the software or be hacked by the hackers. So, we applied the following data preprocessing methods to adjust the data distribution to improve our model's performance on the recall rate:

- Over sample: oversampling can be defined as adding more copies of the minority class. Oversampling can be a good choice when you don't have a ton of data to work with.
- Under sample: under sampling can be defined as removing some observations of the majority class. Undersampling can be a good choice when you have a ton of data, millions of rows. But the drawback is that we are removing information that may be valuable. This could lead to underfitting and poor generalization to the test set.
- Syn sample: a technique similar to over sampling is to create synthetic samples. Here we will use imblearn's SMOTE or Synthetic Minority Oversampling Technique. SMOTE uses a nearest neighbors' algorithm to generate new and synthetic data we can use for training our model.
- Standardization: standardization is the process of implementing and developing technical standards based on the consensus of different parties that include firms, users, interest groups, standards organizations and governments. We let the numerical distribution of the data follow a normal distribution

In all the datasets, the data for some columns is orders of magnitude larger than the data for the other columns. So, standardization is necessary for all sampling attempts because it will let the

data's size obey normal distribution to reduce the classifier's cost to converge and boost the performance. We first applied the Over Sample combined with standardization. The number of the data and the distribution are shown in Figure 9. We can see that now the data is balanced and the number of the data is about 1.5 times as before.

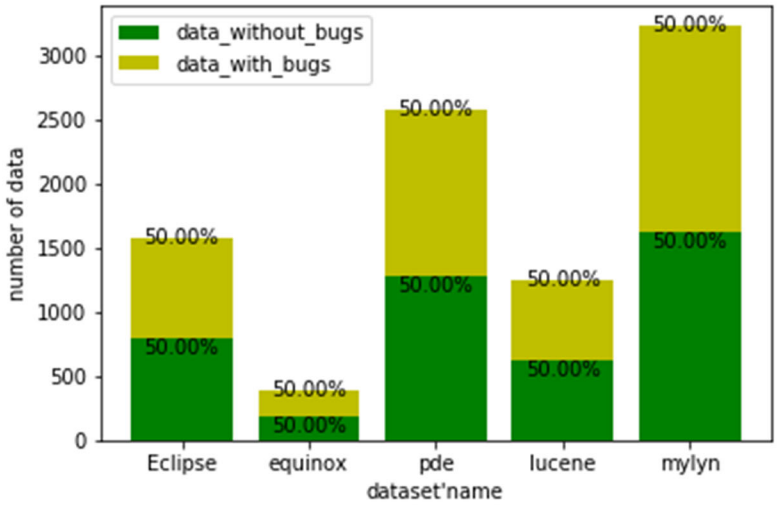

Figure 9: Distribution of D'Ambros's dataset (D'Ambros, 2010) after over sampling

The under sample is not favored in this research is because the datasets used are relatively small. In this way, insufficient data will affect the performance of the classifiers. Therefore, we abandon the under sample and tried syn sample which is the combination of over sample and under sample by using KNN algorithms. The distributions of the dataset after syn sample is shown in Figure 10. The data with bugs are now taking up more percentage than the data without bugs in all the software dataset. Meanwhile the total number of the data in most datasets is larger than the original.

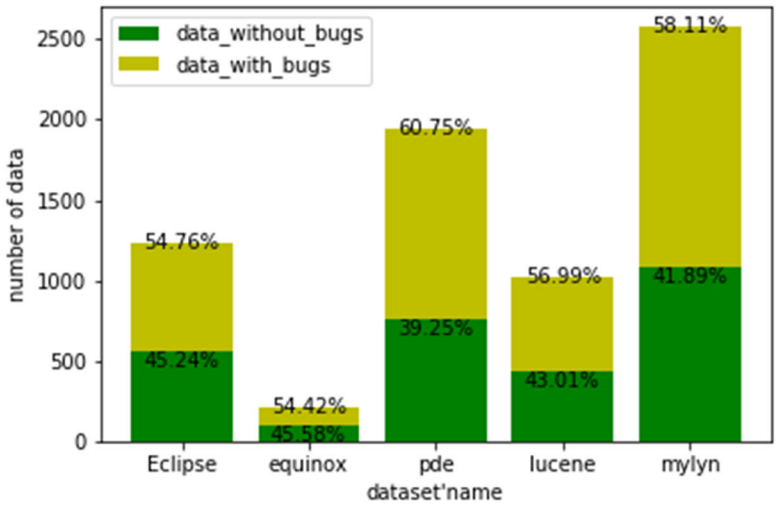

Figure 10: Distribution of D'Ambros's dataset (D'Ambros, 2010) after syn sampling

5.3 Experiments of training and testing on the same project

In this section, we test Zimmermann's dataset (Zimmermann, 2007) on six different machine learning algorithms. We use the train set and test set from the same project which will give us five results for each data preprocessing method and machine learning algorithm. To get better results, we use grid search to tune parameters for each algorithm. Table 3 is the optimized parameters for each algorithm. The first column shows the name of different algorithms, the second column shows the parameters with which we can obtain the best performance after tuning the parameters.

Table 3: Parameters for different algorithms on the same project

Algorithm name	Parameters	Dataset	Range of Parameters
K-nearest neighbors	n_neighbors=3, weights = 'distance', algorithm='auto', leaf_size=30, p=2, metric='minkowski'	Zimmermann's data set(Zimmermann, 2007)	n_neighbors = [1:9], weights = {'uniform','distance'}, eaf_size=[20:50], p = [1,2]
Naive Bayes	GaussianNB()	Zimmermann's data set(Zimmermann, 2007)	BernoulliNB() GaussianNB()
Logistic Regression	solver='liblinear', max_iter=1000, class_weight='balanced', penalty='l2', dual=False, tol=0.0001, C=1.0	Zimmermann's data set(Zimmermann, 2007)	solver = {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'} C = {0.1, 1.0, 10.0} max_iter = {100, 1000} class_weight = {'balanced',None}
Decision Tree	random_state=0, class_weight='balanced', criterion='gini', splitter='best'	Zimmermann's data set(Zimmermann, 2007)	criterion = {'gini', 'entropy'} class_weight = {'balanced',None}

Random Forest	n_estimators = 200, criterion = 'entropy'	Zimmermann's data set(Zimmermann, 2007)	n_estimators = {10, 20, 50, 100, 200} criterion = {'gini', 'entropy'}
Neural Networks	solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(100, 100), activation='relu', alpha=1e-05, batch_size='auto', learning_rate='constant', shuffle=True	Zimmermann's data set(Zimmermann, 2007)	hidden_layer_sizes = {(50,), (100,), (200,), (50, 50), (100, 100), (200, 200)} solver = {'lbfgs', 'sgd', 'adam'} alpha = {1e-3, 1e-4, 1e-5}

Figure 11 is the result we got for training and testing for the same project Zimmermann's dataset (Zimmermann, 2007). Figure 11a shows the results we got without any sampling methods. We can see that except for logistic regression which can balance each class's weight, all algorithms' recall rates are just about 20%. Figure 11b shows the results with over sample preprocessing method. The accuracy is slightly lower for KNN and neural networks, while the recall rates increase dramatically. Figure 11c shows the result after under sampling the data. The accuracy is much lower than the first row. Figure 11d is the result after syn sample. The accuracy we got is slightly lower than the result of original dataset. However, the recall rates are much better than the original ones. For logistic regression, random forest and neural networks, the accuracy and recall rate are above or near 70%.

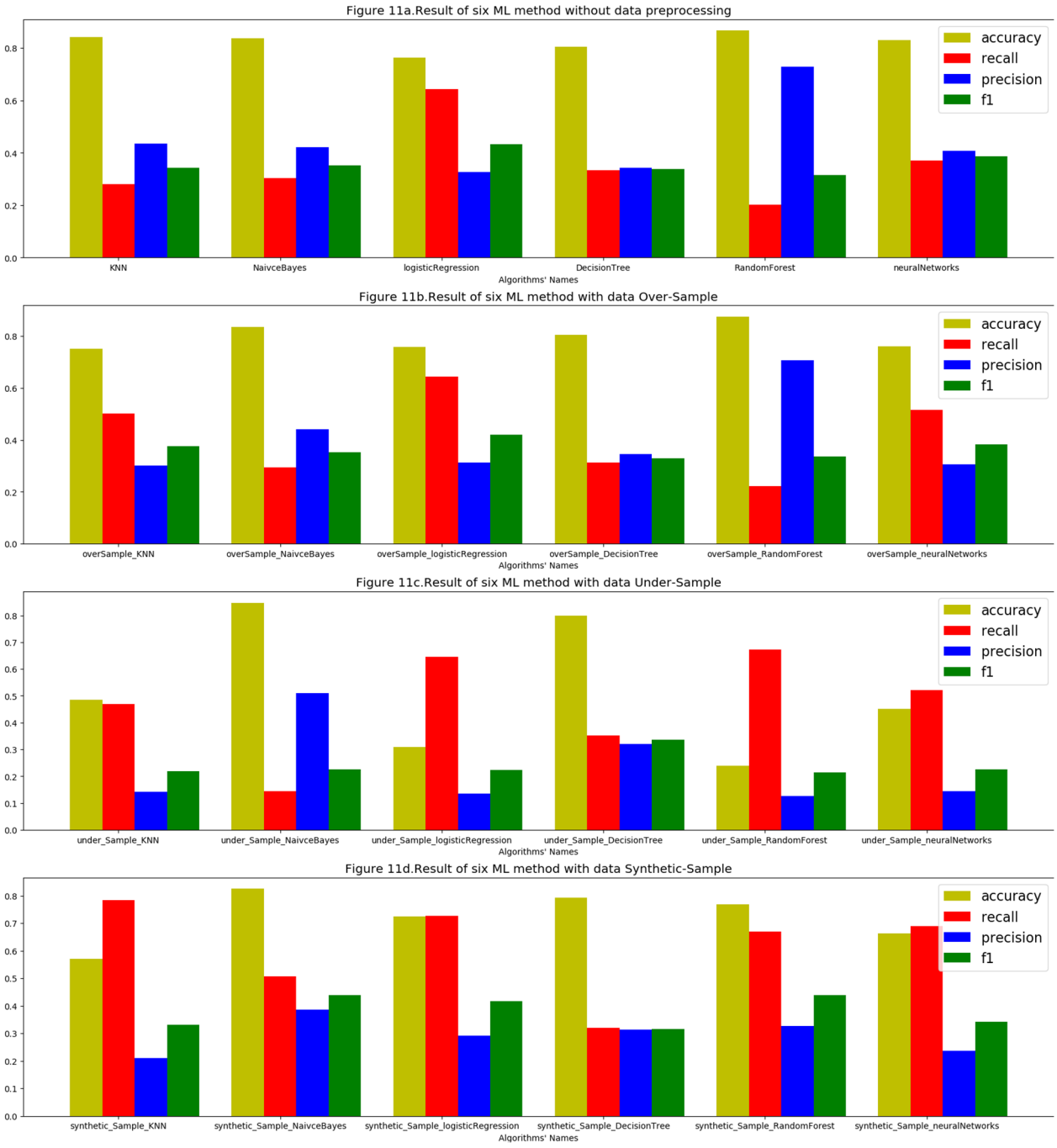

Figure 11: Results with the methods with the same project on Zimmermann's dataset (Zimmermann, 2007)

5.4 Experiments of training and testing on the different projects

In this section, we test D'Ambros's dataset (D'Ambros, 2010) on different machine learning algorithms. We use the train set from one project and test on a different projects. In this way, we will get 25 results on each preprocessing method and machine learning algorithm. To get better

results, we use grid search to tune parameters for each algorithm. Table 4 is the optimized parameters for each algorithm. The first column shows the name of different algorithms, the second column shows the parameters with which we can obtain the best performance after tuning the parameters and the third column shows the dataset we use.

Table 4: Parameters for different algorithms on the different projects

Algorithm name	Parameters	Dataset	Range of Parameters
LogisticRegression	solver = 'liblinear', C = 1.0,max_iter = 1000, class_weight = 'balanced'	D'Ambros's dataset (D'Ambros, 2010)	solver = {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'} C = {0.1, 1.0, 10.0} max_iter = {100, 1000}
MLP with OverSample	solver = 'lbfgs', alpha = 1e-5, hidden_layer_sizes = (100, 100)	D'Ambros's dataset (D'Ambros, 2010)	hidden_layer_sizes = {(50,), (100,), (200,), (50, 50), (100, 100), (200, 200)} solver = {'lbfgs', 'sgd', 'adam'} alpha = {1e-3, 1e-4, 1e-5}
RF	n_estimators = 200, criterion = 'entropy'	D'Ambros's dataset (D'Ambros, 2010)	n_estimators = {10, 20, 50, 100, 200} criterion = {'gini', 'entropy'}
RF with OverSample	n_estimators = 200, criterion = 'entropy'	D'Ambros's dataset (D'Ambros, 2010)	n_estimators = {10, 20, 50, 100, 200} criterion = {'gini', 'entropy'}

RF with SMOTEENN	n_estimators = 200, criterion = 'entropy'	D'Ambros's dataset (D'Ambros, 2010)	n_estimators = {10, 20, 50, 100, 200} criterion = {'gini', 'entropy'}
RF with SMOTE	n_estimators = 200, criterion = 'entropy'	D'Ambros's dataset (D'Ambros, 2010)	n_estimators = {10, 20, 50, 100, 200} criterion = {'gini', 'entropy'}

Figures 12 - 16 includes the results we got for training and testing cross the different projects on D'Ambros' dataset (D'Ambros, 2010). Figure 12, 13, 14, 15, 16 apply Eclipse_JDT_Core, Eclipse_PDE_UI, Equinox_Framework, Lucene and Mylynas as train set. In each figure, the four subplots show Accuracy score, Recall score, Precision score and F1 score separately. In each subplot, the x axis means we use different project, including Eclipse_JDT_Core, Eclipse_PDE_UI, Equinox_Framework, Lucene and Mylynas as test set and different lines represent different algorithms, including LogisticRegression, Multilayer perceptron with OverSample, Random forest, Random forest with OverSample, Random forest with SMOTE and Random forest with SMOTEEN.

Generally, we found that fandom forest and fandom forest with overSample give us the better accuracy score, which usually above 0.8, sometimes even above 0.9 and that random forest with smoteen gives us the better recall score, which is usually above 0.7 and that fandom forest and fandom forest with overSample give us the better precision score, which is usually above 0.5 and that fandom forest with smote and fandom forest with smoteen give us the better f1 score, which is sometimes above 0.5.

Generally, the performance of multilayer perceptron with overSample is better than the performance of logistic regression and the performance of random forest is better than the performance of multilayer perceptron with overSample. Usually, just applying random forest can give us the highest accuracy score while the recall score is low, which can be explained by the imbalance of our data. Our samples without bugs is far more than our samples with bugs. After applying the data preprocessing methods, including oversample, smote and smoteen, the accuracy score may slightly decrease, but usually recall score and f1 score has increased significantly.

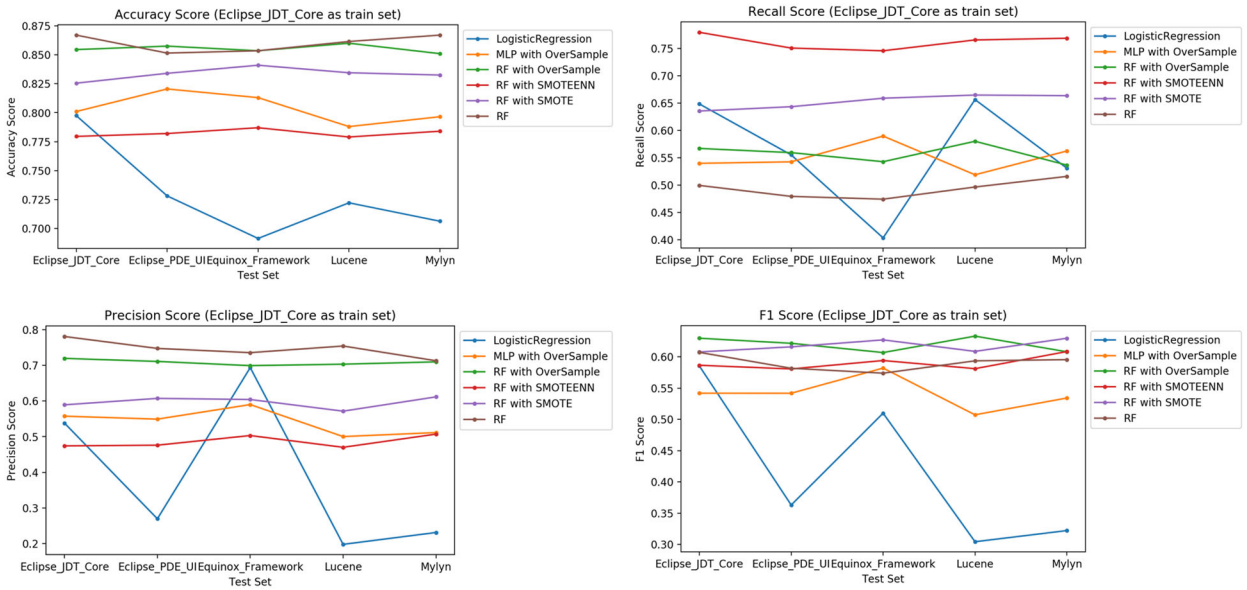

Figure 12: Accuracy, Recall, Precision and F1 score with different algorithms using Eclipse_JDT_Core as train set with different projects on D'Ambros's dataset (D'Ambros, 2010)

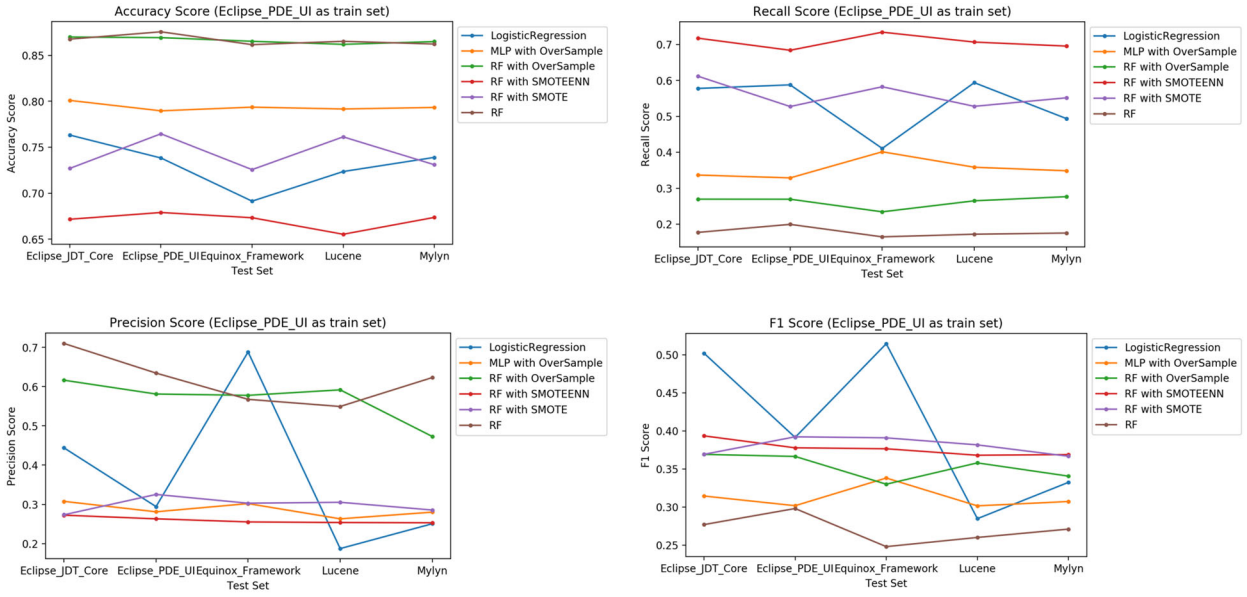

Figure 13: Accuracy, Recall, Precision and F1 score with different algorithms using Eclipse_PDE_UI as train set with different projects on D'Ambros's dataset (D'Ambros, 2010)

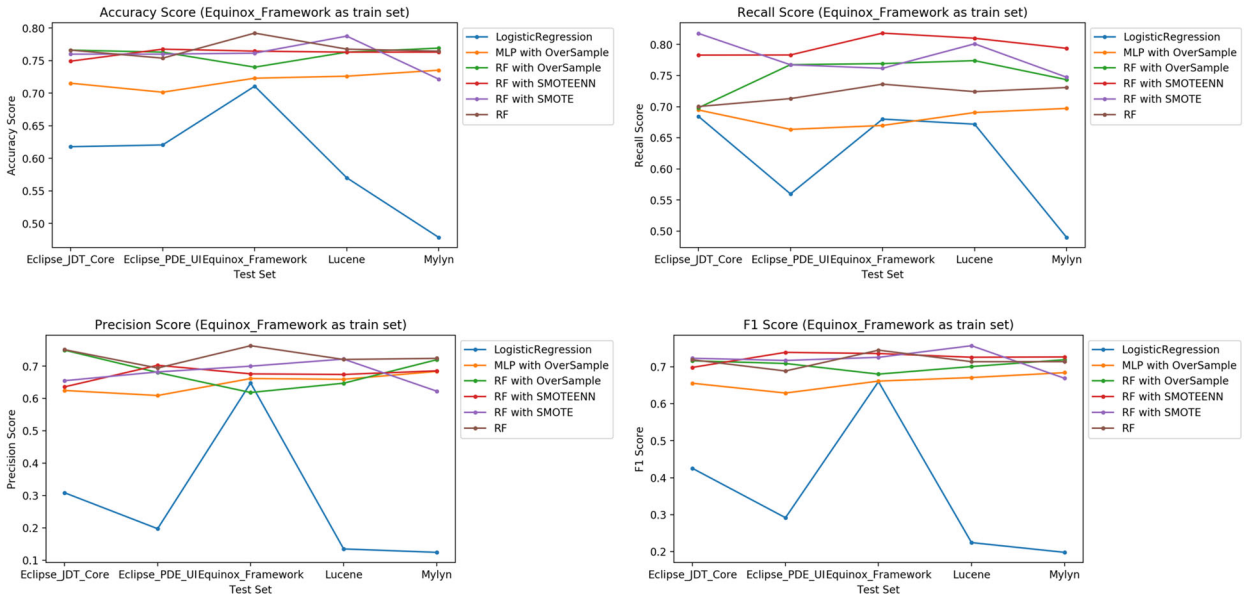

Figure 14: Accuracy, Recall, Precision and F1 score with different algorithms using Equinox_Framework as train set with different projects on D'Ambros's dataset (D'Ambros, 2010)

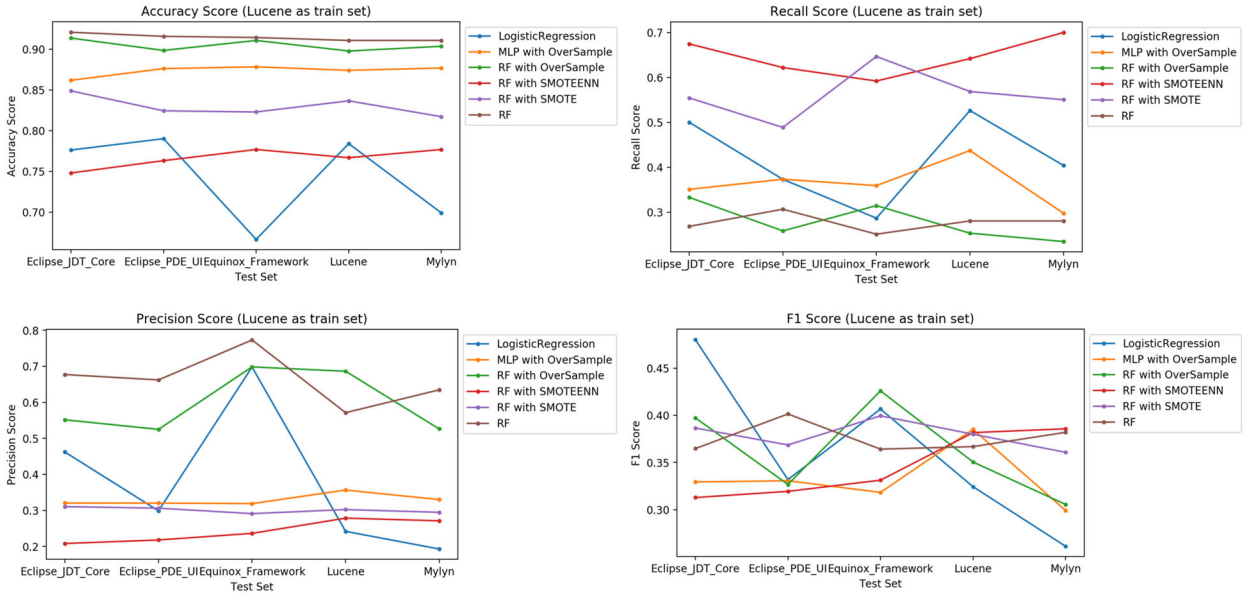

Figure 15: Accuracy, Recall, Precision and F1 score with different algorithms using Lucene as train set with different projects on D'Ambros's dataset (D'Ambros, 2010)

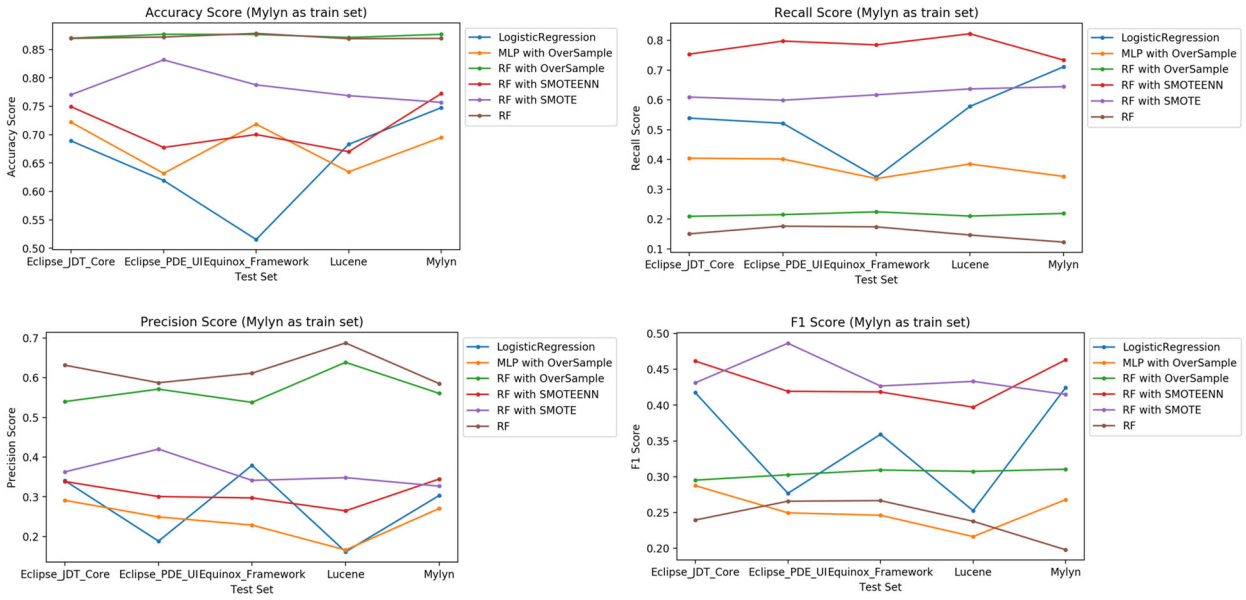

Figure 16: Accuracy, Recall, Precision and F1 score with different algorithms using Mylyn as train set with different projects on D'Ambros's dataset (D'Ambros, 2010)

5.5 Experiments of training and testing on the different versions of Eclipse project

In this section, we test Zimmermann's data set (Zimmermann, 2007) which we use one version of eclipse project as our train set and another version of eclipse project as test set. Table 5 is the optimized parameters for each algorithm. The first column shows the name of different algorithms, the second column shows the parameters with which we can obtain the best performance after tuning the parameters and the third column shows the dataset we use.

Table 5: Parameters for different algorithms on the different versions of Eclipse project

Algorithm name	Parameters	Dataset	Range of Parameters
LogisticRegression	solver = 'liblinear', C = 1.0,max_iter = 1000, class_weight = {0: 0.125, 1: 0.875}	Zimmermann's data set(Zimmermann, 2007)	solver = {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'} C = {0.1, 1.0, 10.0} max_iter = {100, 1000}
Random Forest with SMOTE	n_estimators = 200, criterion = 'entropy'	Zimmermann's data set(Zimmermann, 2007)	n_estimators = {10, 20, 50, 100, 200} criterion = {'gini', 'entropy'}
Random Forest with SMOTEENN	n_estimators = 200, criterion = 'entropy'	Zimmermann's data set(Zimmermann, 2007)	n_estimators = {10, 20, 50, 100, 200} criterion = {'gini', 'entropy'}

Figures 17 – 19 and Table 6 include the result we got for training and testing cross the different projects on Zimmermann's dataset (Zimmermann, 2007). Figure 17, 18, 19 apply eclipse 2.0, eclipse 2.1 and eclipse 3.0 as train set. In each figure, the four subplots show Accuracy score, Recall score, Precision score and F1 score separately. In each subplot, the x axis means we use different project, including eclipse 2.0, eclipse 2.1 and eclipse 3.0 as test set and different lines represent different algorithms, including LogisticRegression, Random forest with SMOTE and Random forest with SMOTEEN.

Generally, we found that fandom forest with SMOTE give us the better accuracy score, which usually above 0.8 and that logistic regression and random forest with smoteen gives us the better recall score, which is usually above 0.6 and that fandom forest with SMOTE give us the better precision score, which is usually above 0.4 and that logistic regression, fandom forest with smote and fandom forest with smoteen all give us the better f1 score, which is sometimes above 0.4.

Generally, our data is imbalanced because our samples without bugs is far more than our samples with bugs. After applying the logistic regression with appropriate class_weight or different data preprocessing methods, including smote and smoteen, the accuracy score we get is a little bit lower

than the accuracy score on the paper while the recall score and f1 score are significantly higher that the recall score and f1 score on the paper.

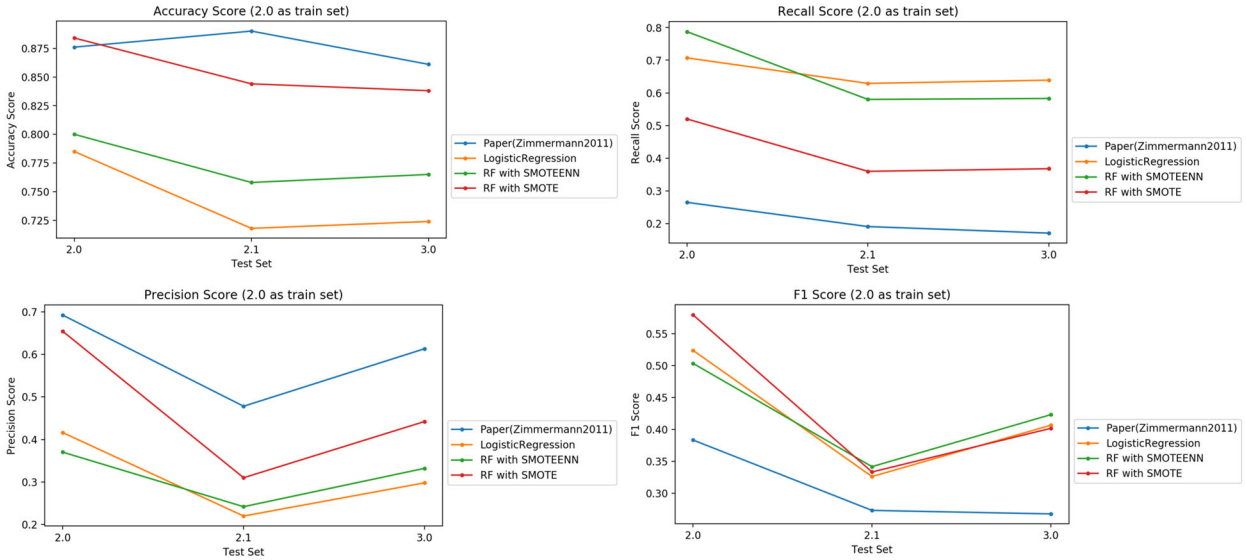

Figure 17: Accuracy, Recall, Precision and F1 score with different algorithms using Eclipse 2.0 as train set with different versions of Eclipse on Zimmermann's data set (Zimmermann, 2007)

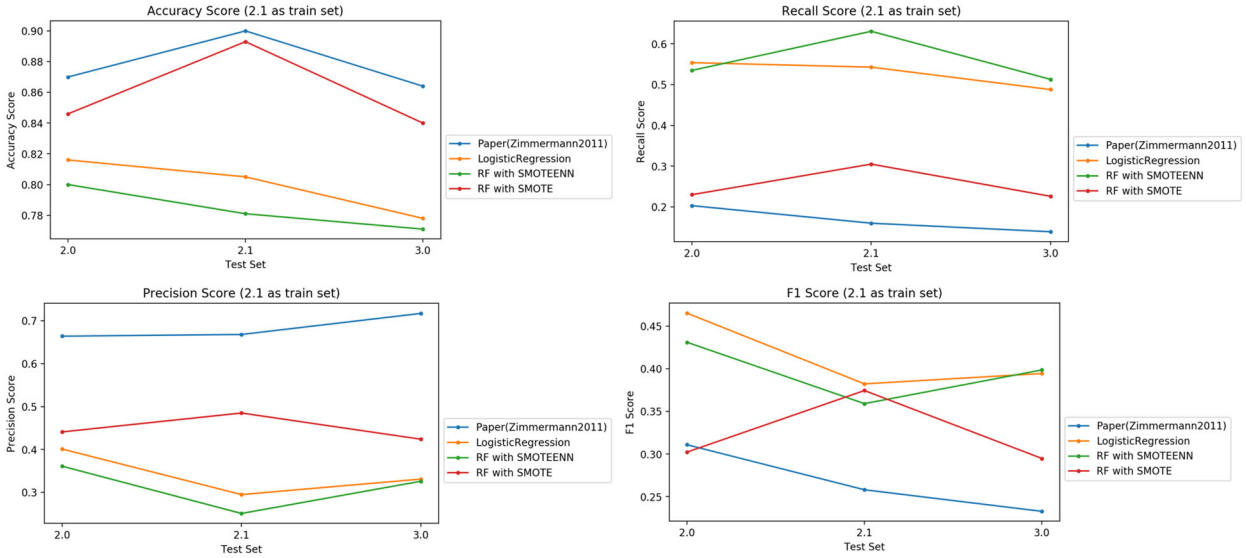

Figure 18: Accuracy, Recall, Precision and F1 score with different algorithms using Eclipse 2.1 as train set with different versions of Eclipse on Zimmermann's data set (Zimmermann, 2007)

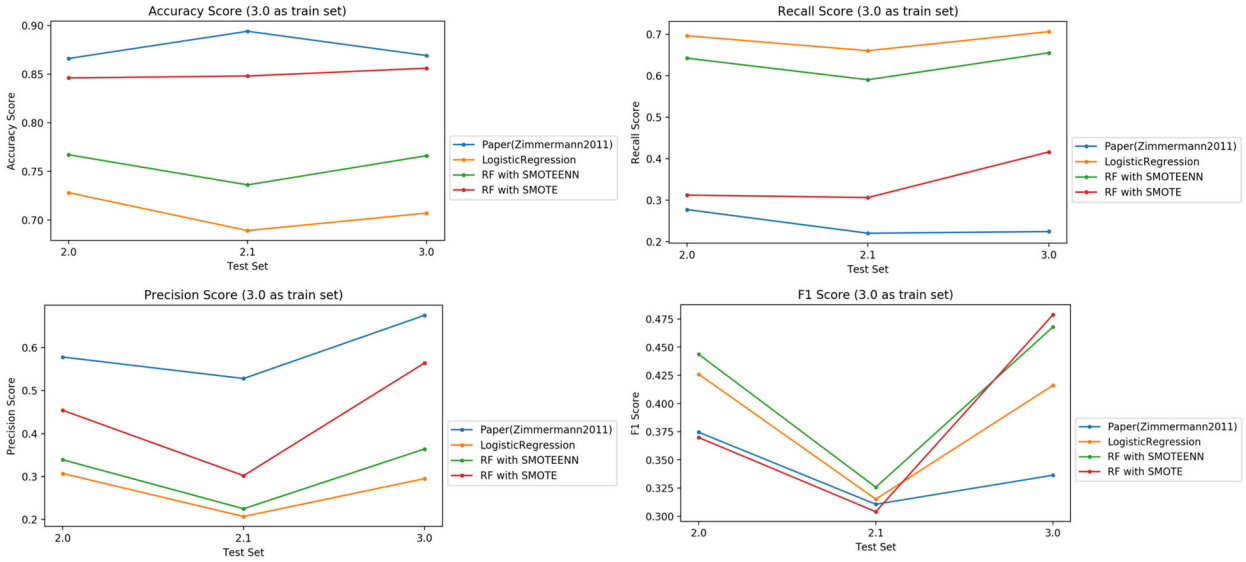

Figure 19: Accuracy, Recall, Precision and F1 score with different algorithms using Eclipse 3.0 as train set with different versions of Eclipse on Zimmermann's data set (Zimmermann, 2007)

Table 6: Results of Classification of files on different versions of Eclipse project with Zimmermann's paper and different algorithms we apply

Training	Testing	Defects	Accuracy_score	Recall_score	Precision_score	F1_score
2	2	0.145	0.876	0.265	**0.692**	0.383
			0.785	0.707	0.416	0.524
			0.8	**0.787**	0.37	0.503
			0.884	0.52	0.654	**0.579**
	2.1	0.108	**0.89**	0.191	**0.478**	0.273
			0.718	**0.629**	0.22	0.326
			0.758	0.58	0.242	**0.342**
			0.844	0.36	0.31	0.333
	3	0.148	**0.861**	0.171	**0.613**	0.267
			0.724	**0.639**	0.298	0.406
			0.765	0.583	0.332	**0.423**
			0.838	0.368	0.442	0.402
2.1	2	0.145	**0.87**	0.203	**0.664**	0.311
			0.816	**0.554**	0.401	**0.465**
			0.8	0.535	0.361	0.431
			0.846	0.23	0.441	0.302
	2.1	0.108	**0.9**	0.16	**0.668**	0.258
			0.805	0.543	0.295	**0.382**
			0.781	**0.631**	0.251	0.359

			0.893	0.305	0.485	0.374
	3	0.148	**0.864**	0.139	**0.717**	0.233
			0.778	0.488	0.331	0.394
			0.771	**0.513**	0.326	**0.399**
			0.84	0.226	0.424	0.295
3	2	0.145	**0.866**	0.277	**0.578**	0.375
			0.728	**0.696**	0.307	0.426
			0.767	0.642	0.339	**0.444**
			0.846	0.312	0.454	0.370
	2.1	0.108	**0.894**	0.22	**0.528**	0.311
			0.689	**0.66**	0.207	0.315
			0.736	0.59	0.225	**0.326**
			0.848	0.306	0.302	0.304
	3	0.148	**0.869**	0.224	**0.675**	0.336
			0.707	**0.706**	0.295	0.416
			0.766	0.655	0.364	0.468
			0.856	0.416	0.564	**0.479**
Paper (Zimmermann2011)			LogisticRegression (solver = 'liblinear', max_iter = 1000, class_weight = {0: 0.125, 1: 0.875})			
RandomForestClassifier (n_estimators = 200, criterion = 'entropy') with SMOTEENN			RandomForestClassifier (n_estimators = 200, criterion = 'entropy') with SMOTE			

6. Conclusions

In this study, we apply six machine learning methods (Naive Bayes, Logistic Regression, K-nearest Neighbors, Decision Tree, Random Forest, and Neural Networks) combined with three preprocessing (over sample, under sample and synthetic sample) methods on two different datasets (D'Ambros's dataset (D'Ambros, 2010) and Zimmermann's data set (Zimmermann, 2007)) to test the bug prediction performance. Our results show that:

1) All machine learning algorithms can achieve 70% - 85% accuracy score without any data preprocessing methods, while the recall score is only 20%-30% except for logistic regression. The synthetic sample preprocessing method can greatly improve the machine learning methods' performance of recall score.
2) Logistic regression, random forest and neural networks behave best after syn sampling which can achieve 70% - 80% both on accuracy score and recall score within the same open source software's dataset.
3) The f1 score cross the projects when using Eclipse_JDT_Core, Equinox_Framework as train set are better, which is between 50% - 70%. The overall result for accuracy score is 80% and for recall score is 50% while crossing the projects.
4) Although the accuracy score and precision score we got are lower than Zimmermann's result (Zimmermann, 2007), the recall rate after using random forest combined with syn sample (smoteenn or smote) or logistic regression on different versions of eclipse's dataset are much better (20% - 50%) than Zimmermann's result (Zimmermann, 2007). Besides, the F1 score we got is also better than Zimmermann's result (Zimmermann, 2007).

In fact, we focus more on finding the versions of software with bugs, especially when the dataset is imbalanced and the versions without bugs are much more than versions with bugs. Thus, the recall score is more essential than accuracy score and precision score. We conclude that the data preprocessing methods such as syn sample can improve the performance of the recall rate in all situations when testing within the dataset, testing crossing the dataset and testing crossing the versions.

7. References

[1] Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, *7007*(011), 429-489.

[2] Akiyama, F. (1971, August). An Example of Software System Debugging. In *IFIP Congress (1)* (Vol. 71, pp. 353-359).

[3] Kamei, Y., & Shihab, E. (2016, March). Defect prediction: Accomplishments and future challenges. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)* (Vol. 5, pp. 33-45). IEEE.

[4] Tantithamthavorn, C., McIntosh, S., Hassan, A. E., & Matsumoto, K. (2016, May). Automated parameter optimization of classification techniques for defect prediction models. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (pp. 321-332). IEEE.

[5] Tóth, Z., Gyimesi, P., & Ferenc, R. (2016, July). A public bug database of github projects and its application in bug prediction. In *International Conference on Computational Science and Its Applications* (pp. 625-638). Springer, Cham.

[6] Xia, X., Lo, D., Wang, X., Yang, X., Li, S., & Sun, J. (2013, March). A comparative study of supervised learning algorithms for re-opened bug prediction. In *2013 17th European Conference on Software Maintenance and Reengineering* (pp. 331-334). IEEE.

[7] Zhang, F., Zheng, Q., Zou, Y., & Hassan, A. E. (2016, May). Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering* (pp. 309-320). ACM.

[8] Couto, C., Pires, P., Valente, M. T., Bigonha, R. S., & Anquetil, N. (2014). Predicting software defects with causality tests. *Journal of Systems and Software*, *93*, 24-41.

[9] Osman, H., Ghafari, M., & Nierstrasz, O. (2017, February). Hyperparameter optimization to improve bug prediction accuracy. In *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)* (pp. 33-38). IEEE.

[10] Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K. I., Adams, B., & Hassan, A. E. (2010, September). Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance* (pp. 1-10). IEEE.

[11] Shivaji, S., Whitehead Jr, E. J., Akella, R., & Kim, S. (2009, November). Reducing features to improve bug prediction. In *2009 IEEE/ACM International Conference on Automated Software Engineering* (pp. 600-604). IEEE.

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

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

[14] Kim, S., Zhang, H., Wu, R., & Gong, L. (2011, May). Dealing with noise in defect prediction. In *2011 33rd International Conference on Software Engineering (ICSE)* (pp. 481-490). IEEE.

[15] Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W. M., Ohira, M., Adams, B., ... & Matsumoto, K. I. (2010, October). Predicting re-opened bugs: A case study on the eclipse project. In *2010 17th Working Conference on Reverse Engineering* (pp. 249-258). IEEE.

[16] Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W. M., Ohira, M., Adams, B., ... & Matsumoto, K. I. (2013). Studying re-opened bugs in open source software. *Empirical Software Engineering*, *18*(5), 1005-1042.

[17] D'Ambros, M., Lanza, M., & Robbes, R. (2010, May). An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)* (pp. 31-41). IEEE.

[18] Shirabad, J. S., & Menzies, T. J. (2005). The PROMISE repository of software engineering databases. School of Information Technology and Engineering, University of Ottawa, Canada, 24.

[19] Koehrsen, W. (2017). Random Forest Simple Explanation. Published in www.medium.com. Last accessed on 10/23/2019.

[20] Venelin, V. (2017). Creating a Neural Network from Scratch — TensorFlow for Hackers (Part IV). Published in www.medium.com. Last accessed on 10/29/2019.

[21] Zohar, K., & Amir, S. (2018). Amazon SageMaker supports kNN classification and regression. Published in www.medium.com. Last accessed on 10/29/2019.