Journal of Information Technology Management

Print ISSN: 2008-5893

Online ISSN: 2423-5059

Investigating the Role of Code Smells in Preventive Maintenance

Junaid Ali Reshi

*Corresponding author, PhD Candidate, Department of Computer Science and Technology, Central University of Punjab, Bhatinda, Punjab, India. E-mail: jreshi14@gmail.com.

Satwinder Singh

Assistant Prof., Department of Computer Science and Technology, Central University of Punjab, Bhatinda, Punjab, India. E-mail: satwindercse@gmail.com

Abstract

The quest for improving the software quality has given rise to various studies which focus on the enhancement of the quality of software through various processes. Code smells, which are indicators of the software quality have not been put to an extensive study for as to determine their role in the prediction of defects in the software. This study aims to investigate the role of code smells in prediction of non-faulty classes. We examine the Eclipse software with four versions (3.2, 3.3, 3.6, and 3.7) for metrics and smells. Further, different code smells, derived subjectively through iPlasma, are taken into conjugation and three efficient, but subjective models are developed to detect code smells on each of Random Forest, J48 and SVM machine learning algorithms. This model is then used to detect the absence of defects in the four Eclipse versions. The effect of balanced and unbalanced datasets is also examined for these four versions. The results suggest that the code smells can be a valuable feature in discriminating absence of defects in a software.

Keywords: Preventive maintenance, Code smells, Machine learning, Random forest.

DOI: 10.22059/jitm.2019.274968.2335 © University of Tehran, Faculty of Management

Introduction

In current times, technology plays an important role in our day to day lives. In every sphere of life, we use gadgets to make the work easier and faster. We use smart phones, smart watches, heartbeat trackers, and many personified gadgets. These gadgets and devices, apart from the hardware need software to work perfectly. In research and business, we use a variety of software for data analysis, account management, project management and human resource management etc. In short, we heavily depend on software for almost all the automations in our lives. These software sometimes do not behave in order and can be a huge pain for all of us. In order to maintain these software, there are various people working constantly and many frameworks have been built for its maintenance. This has given rise to various standards and practices for software maintenance. ISO/IEC 14764:2006(E)[١] and IEEE Std 14764-2006[٢] define three types of software maintenance: Corrective, Preventive, Adaptive, and Perfective maintenance. Preventive maintenance deals with tackling potential errors/defects/bugs in a software. One of the sub-parts of preventive maintenance is software defect prediction. Software defect prediction involves predicting probable defective components in a software much before they cause any problem. Various efforts have been made to predict defects in software, so as to make the product robust and reducing corrective maintenance. There have been various approaches in determining the defects in software. Various aspects of software maintenance have always been researched and put to experimentation so as to improve software maintenance. Some of these methods rely on statistical measures while others employ software metrics thresholds (Kapila & Singh, 2013; Catal, 2011). This has resulted in the evolution of various software metrics, their treatment, and the development of code smells (Singh & Kaur, 2017). One of the emerging field is the application of Machine learning algorithm to the problem of fault prediction.

The field of machine learning is an emerging and fascinating field of research, which focuses on the improvement of perception, cognition and action of computers through continuous learning and evolving with experience. It is a field, which makes machines efficient enough to handle large amounts of diverse information of various disciplines for making decisions, providing estimates and predictions, each with applied knowledge of the field that has previously been learned. Supervised learning is the application of machine learning algorithms to learn a pattern on the basis of already available data about a phenomenon, referred to as training, and then make predictions about a scenario. There are a lot of applications of the machine learning techniques. Among other fields, software engineering also uses the services of machine learning algorithm to augment various activities of software maintenance, the defect prediction being one of them (Lessmann, Baesens, Mues, & Pietsch, 2008).

[١] https://www.iso.org/obp/ui/#iso:std:iso-iec:14764:ed-2:v1:en
[٢] https://standards.ieee.org/standard/14764-2006.html

The code smells have been found to be efficient descriptors of software code quality (Yamashita & Moonen, 2013). Code smells have been described in various literatures and have been constantly a matter of research. Various researchers have defined code smells and their detection strategies (Singh & Kaur, 2017).The pioneering work in the field of code smells has been done by Martin Fowler who has described 22 types of code smells and the techniques for their detection (Fowler, Beck, Brant, Opdyke, & Roberts, 2002). Code smells have not yet been extensively used as a factor in determining the presence or absence of defects in a software.

This study takes a step forward to look for the possible ways to improve and augment the process of defect prediction through the aid of software code smells. The study is based on the hypothesis that the code smells have a definitive role in the process of defect prediction and that the absence of code smells can be utilised as a factor for in the process of defect prediction through machine learning.

Data Extraction and Analysis

The methodology employed for the task of predicting non-faulty classes contains essential data mining task as well. All data mining tasks require some of the data pre-processing techniques for the data to be in shape so that it can be fed to a machine learning algorithm. We carried out some basic processes to suitably prepare data for the machine learning algorithms. The processes that we carried out are listed as under:

Dataset Selection: Dataset selection is the important task in the problem of machine learning. Classification too performs better if the dataset is more relevant to the problem. The more optimal the database, the better the accuracy and less the time and resources consumed. The dataset selected in the case was relevant to the software as previous studies have shown the object oriented metrics data to be efficient in the detection of the defects and metrics as depicters of software quality is a well-established fact (Catal, 2011).

Source code selection: The basic process of a study is always determined by the type of data to be studied. The type of data determines the validity of inferences and their extensions. For this study, we choose Eclipse framework which is a very popular object oriented software. The object oriented software are extensively found in every field of application. The ease and applicability of object oriented framework has made object oriented software the most popular line of software which are in vague as well. The results inferred from the study of this software will be extensible to the software products which are similar in nature. As Eclipse is an industry sized and having similar characteristics as that of industry level software so we used it for the analysis so that the inferences could be extended to industry level software. The platform in which the software is written in Java, which is the widely used language in the development of software. The Eclipse software is open source software and it allows open access to its bug repository. This was another reason to select Eclipse for the analysis as the work can be easily reproducible and verifiable. There have been many studies conducted on the Eclipse software that make it a kind of standard to be analysed. In addition, the software being open source will contribute healthily towards research on the open source platform, which will make the research replicable and inferable and can help in setting benchmarks. The information pertaining to the source code selected is as:

Table 1. Eclipse Source Code Information

Build name	Build Date
Eclipse 3.2	Thu, 29 Jun 2006
Eclipse 3.3	Mon, 25 Jun 2007
Eclipse 3.6	Tue, 8 Jun 2010
Eclipse 3.7	Mon, 13 Jun 2011

Data acquisition and compilation

The data acquisition is another important aspect of the process. The metrics and the smell data were obtained from Understand and iPlasma tools. The bug data was acquired from official bug repository for Eclipse, Bugzilla[1]

Metrics extraction

Metrics, as fault depicters have been used in various studies. The metric values have been utilised to train various defect prediction models. The defect prediction models have proven to be efficient as concluded by various studies (Cartwright & Shepperd, 2000; Catal, 2011; Hall, Beecham, Bowes, Gray, & Counsell, 2011).

The metrics extraction was carried out by a static code analyser tool called as Understand™. The source code was analysed for the object oriented metrics. The metrics that were taken into consideration are as under:

- LCOM (Percent Lack of Cohesion)
- IFANIN (Count of Base Classes)
- RFC (Count of All Methods)
- DIT (Max Inheritance Tree)
- NIV (Count of Instance Variables)
- NIM (Count of Instance Methods)
- CBO (Count of Coupled Classes)
- WMC (Count of Methods)
- NOC (Count of Derived Classes)

[1] https://bugs.eclipse.org

Bug association and compilation

The association of bugs with the metrics file was carried out by examining the online bug repository, Bugzilla, for the purpose. The bugs were manually sort out by a team of Scholars of masters' level who had an adequate knowledge about object oriented concepts and were able enough to read and understand the code. The products that were analysed for the presence of bugs were Eclipse JDT and PDE. The parameters that were used to search the bugs are:

- **Severity:** blocker, critical, major, normal, minor, trivial
- **Priority:** P1, P2, P3, P4, P5
- **Resolution:** Fixed, Invalid, Wontfix, Duplicate, Worksforme, Moved, Not_Eclipse
- **Classification:** Eclipse
- **OS:** All
- **Hardware:** All
- **Product:** JDT, PDE
- **Versions:**3.2,3.3,3.6,3.7

The most important criteria that were followed while associating the bugs were:

- The Bug reports containing patches were only considered.
- The patches were examined carefully and the affected class was identified through the manual patch analysis.

The Bug reports, which did not have a clear distinction of the presence of a bug within a class were not be considered. This means that if there is any ambiguity in associating a bug with a particular class, although the bug is present, the bug was not filed in the dataset created.

If there were one or more than one bugs in a particular class, the class was considered as faulty. In the association of the bugs, only the bugs which had been resolved were considered. This is because, many a times a bug is in its initial stage of resolution and is marked as a bug. But, on the later stage, either that is considered as not bug or duplicate which means that it was not a different bug or it was not a bug altogether. Thus, marking it as bug in the database can lead to a false bug. On the other hand, the bugs marked as resolved are confirmed bugs whose status as a bug would not change. Same strategy has been implemented in the creation of promise data repository (Zimmermann, Premraj, & Zeller, 2007).

Smell detection and association

Code smell is a subjective property of a code which can be interpreted differently by different researchers and tools. Although there is no clear cut definition of code smells, but the code smell definitions do not vary too much as the standard for the code smells have been defined by fowler and implemented by some researchers (Fowler et al., 2002). There have been some tools which have been used by the researchers and are established as a standard in the field. One such standard tool is iPlasma which is freely available and is one of the famous code smell detectors (Fontana, Mäntylä, Zanoni, & Marino, 2016).

The code smells to be considered were chosen on the basis of the literature survey and the ease of availability of the analysis of the smells through open source platform. The code smells were obtained from iPlasma platform and they were associated with the metrics files which were obtained from iPlasma and Understand tool as already defined.

Four class level and three method level code smells were used to create the code smell dataset. They were then consolidated to from a single dataset which simply indicated whether a class is smelly or not, based on the presence or absence of these smells.

Table 2. Code Smells extracted from the Source code

Method level Code Smells	Class level Code smells
Brain method	God Class (God Class + Brain Class)
Shotgun Surgery	Data Class
Feature Envy	Schizofrenic Class
	Refused Bequest

The method level smells were not associated at the method level but rather at class level as the metrics computed were of the class level and not the method level.

Data Assessment

The data thus obtained from the bug repository was assessed and validated for its correctness. A team of 6 M.Tech level students was deployed for the purpose.

After the mapping of the bugs with the classes was done with the parameters mentioned above, and re-validated within the team, the resultant files contained the distribution of the defects per version as:

Table 3. Distribution of defective and non-defective classes in dataset

Source Code	Total No of Bugs filed	Defective Classes	Non Defective Classes
Eclipse 3.2	839	615	4095
Eclipse 3.3	1201	733	4460
Eclipse 3.6	1143	700	5273
Eclipse 3.7	963	502	5465

For the smells, the data was also validated and classes that were resolved by iPlasma were checked with that of the ones given by Understand. The results of the extraction of the consolidated code smells prevalence per version, resolved at the class level is as under:

Table 4. Distribution of the code smells in the original dataset prepared

Source Code	Smelly Classes	Non Smelly Classes	Total Classes
Eclipse 3.2	1689	3020	4718
Eclipse 3.3	1633	3559	5192
Eclipse 3.6	2122	3850	5972
Eclipse 3.7	2114	3852	5966

Data Cleansing

During the mapping of the data, some of the classes were not resolved properly or were redundant in the already prepared data. In this step, the data was examined for the corrupt, incomplete and inaccurate data. The data was thus refined as some of the cells were consolidated while others were deleted and a lot of cells got further refined.

Data Transformations

The necessary data transformations were carried out in the data so that the algorithms can be applied efficiently. As, the number of fields had different types of data, they were homogenised and made to conform to a single standard throughout the various data files. Although, not much data transformations were needed, yet it was an important step to weed out any problems that could possibly hamper the efficient functioning of an algorithm.

Data Balancing

Machine learning algorithms are found to be sensitive to the data imbalance. They tend to perform worse on the data that is not in proper ratio and skews towards a class. There are various techniques for balancing the data. Every dataset needs analysis before choosing a particular data balancing technique. The dataset was analysed and it was concluded that the most suitable technique for balancing the data would be under-sampling. Only those datasets were balanced in which the minority class was below the set threshold. The threshold that was set as a standard was 1/3 of the total data, which means that the minority class should be at least 1/3 of the total dataset. No specific ratio was considered as a standard for already balanced data-sets and they were used in the original form. The majority class was reduced proportionately through random sampling. The criteria for balancing the datasets that was followed was one-third (1/3) positive and two-third (2/3) negative instances which is a popular strategy (Gueheneuc, Sahraoui, & Zaidi, 2004). Accordingly the datasets, when balanced, had the following prevalence:

Table 5. Distribution of defective and non-defective classes after applying under-sampling

Source Code	Defective Classes	Non Defective Classes
Eclipse 3.2	614	1228
Eclipse 3.3	722	1444
Eclipse 3.6	699	1398
Eclipse 3.7	501	1002

The training dataset (metrics-smell dataset) was already balanced except for one version, which was then balanced accordingly. Subsequently, the distribution of classes in that dataset were as under:

Table 6. Distribution of smelly and non-smelly classes after applying under-sampling

Source Code	Smelly Classes	Non Smelly Classes	Total
Eclipse 3.2	1689	3020	4718
Eclipse 3.3	1633	3266	4899
Eclipse 3.6	2122	3850	5972
Eclipse 3.7	2114	3852	5966

Selection of Algorithms

Algorithms are an important component of a Machine learning system. Many Algorithms have been studied for their performances with different types of data and to different fields. There have been various efforts to predict defects and smells using machine learning algorithms. In this study, we use SVM, J48 and RandomForest algorithms for creating Machine learning Models. One of the prominent benchmarking study carried out by Fontana et al. have shown RandomForest and J48 to be in the top 10 performing classification algorithms for smell prediction task (Fontana et al., 2016). Many other benchmarking studies indicate these algorithms to be good performers for defect prediction as well (Lessmann et al., 2008). Hence, Fontana et al. have also concluded RandomForest to be the best overall performing algorithm for both defects and smells (Fontana et al., 2016).SVM, on the other hand , has been an algorithm which tends to perform well on certain data (particularly when feature selection is done) and does not perform well on some data.But, there has been use of SVM for detecting antipatterns or smells (Maiga et al., 2012) and that makes it an interesting candidate for experimentation.

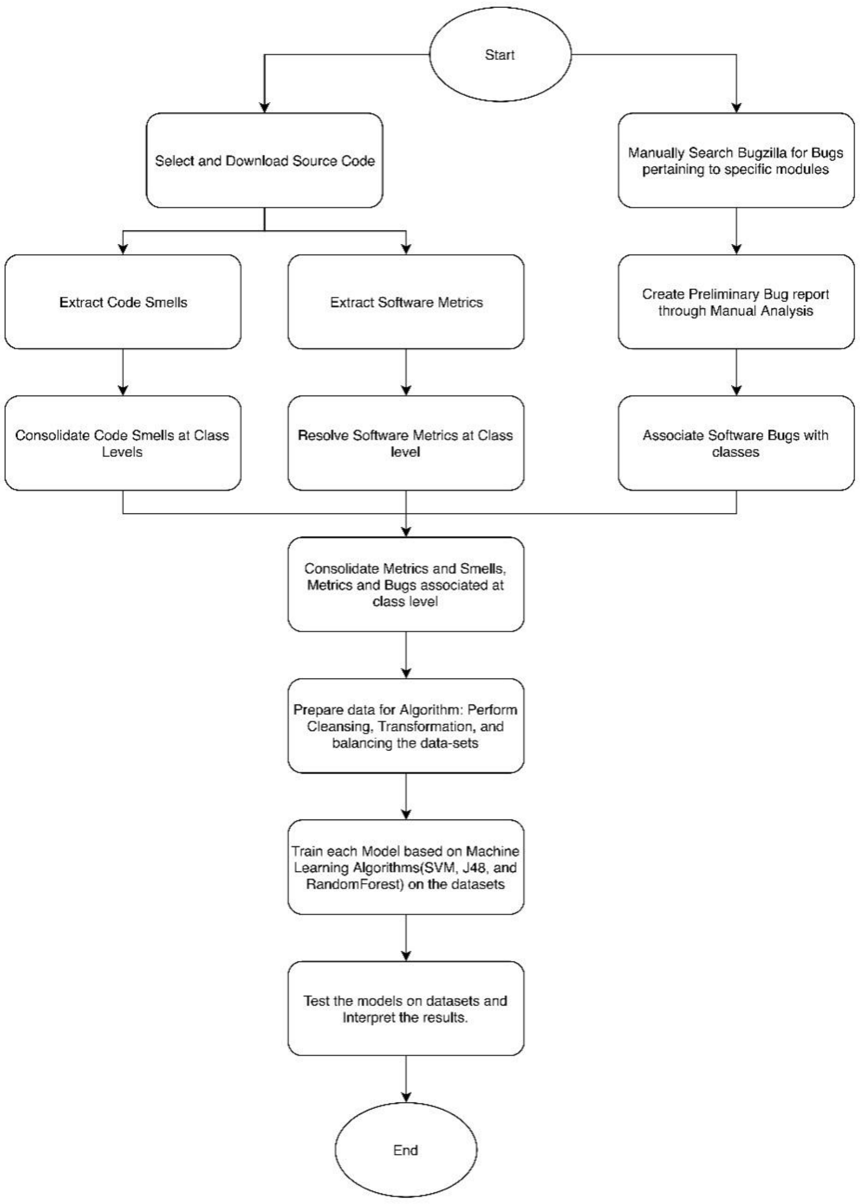
Start
Select and Download Source Code
Manually Search Bugzilla for Bugs pertaining to specific modules
Extract Code Smells
Extract Software Metrics
Create Preliminary Bug report through Manual Analysis
Consolidate Code Smells at Class Levels
Resolve Software Metrics at Class level
Associate Software Bugs with classes
Consolidate Metrics and Smells, Metrics and Bugs associated at class level
Prepare data for Algorithm: Perform Cleansing, Transformation, and balancing the data-sets
Train each Model based on Machine Learning Algorithms(SVM, J48, and RandomForest) on the datasets
Test the models on datasets and Interpret the results.
End

Figure 1. Flow graph of the Research process

Results and Discussion

Various algorithms have been used in the previous studies to study different aspects of software maintenance. In the area of software defect prediction, the application of machine learning techniques for detection of the defects have been in vogue and used extensively, Singh, Kaur, & Malhotra, 2010). In this work, we used Random forest, J48 (Weka implementation of C4.5), and SVM for the construction of the models.

Smell prediction models

The creation and training of smell prediction models was done through the aid of Weka software which is openly available. They were evaluated for their efficiency through tenfold cross validation. Training and testing through this methodology ensures that the model is not over-fitted by the data. The tenfold cross validation strategy divides the dataset into 10 different parts and uses the different combinations (9 for training and one for testing) of the dataset alternatively. The cross-validation procedure is an established standard guaranteeing a stratified sampling of the dataset and reducing the overfitting phenomenon (Bengio & Grandvalet, 2004; Stone, 1974; Cohen & Jensen, 1997), thus providing an efficient way for evaluating our model. Table 7 lists the various performance measures of the smell prediction model. The smell prediction models that were formed had ROC and F-measure value of above 90% in the detection of the smells. ROC and F-measure have been established parameters in the evaluation of machine learning models (Provost, Tom, & Kohavi, 1998; Kim Sang, & De Meulder, 2003).

Table 7. Performance measures of the smell prediction models

Algorithm	Source Code	Precision	Recall	F-Measure	ROC
Random Forest	Eclipse 3.2	0.942	0.942	0.941	0.974
	Eclipse 3.3	0.942	0.942	0.941	0.974
	Eclipse 3.6	0.936	0.937	0.936	0.978
	Eclipse 3.7	0.933	0.933	0.933	0.977
J48	Eclipse 3.2	0.933	0.933	0.933	0.948
	Eclipse 3.3	0.933	0.933	0.933	0.948
	Eclipse 3.6	0.925	0.926	0.925	0.943
	Eclipse 3.7	0.924	0.924	0.924	0.940
SVM	Eclipse 3.2	0.924	0.924	0.923	0.909
	Eclipse 3.3	0.924	0.925	0.924	0.901
	Eclipse 3.6	0.925	0.925	0.924	0.911
	Eclipse 3.7	0.925	0.925	0.924	0.910

Creation of Smell-defect models

As we are already aware that the presence of smells is indicative of the fact that the code is prone to defects or faults (by the definition of code smells), we train our models with the metrics (as already defined) that are the indicators of smells to find out the absence of bugs. The non-smelly code was examined for the absence of bugs to find out the classes which needed less attention for maintenance (reciprocally, the classes which need more attention) and thus dividing the burden in the developers in the same context.

The prediction of the absence of defects was carried out for each versions of the software. The model built on the Eclipse version 3.2 was used to predict defects in the subsequent versions (as listed in the table 4.2).The testing of the absence of bugs was done down the release version. Exemplifying, Eclipse 3.6 trained model was used to test the data on Eclipse 3.6 and Eclipse 3.7 only and not Eclipse 3.2 or Eclipse 3.3.

The testing of data was carried out for each of the models formed on the RandomForest, J48 and SVM. Another testing of data was performed on the balanced datasets, wherever data balancing was required.

The dataset preparation for SVM was a little different. Data pre-processing technique WrapperSubsetEval with Evolutionary search was used to treat the data before creating smell prediction models. The performance of SVM is measured on the imbalanced data as well as the balanced and normalised data.

Random Forest based prediction Models

Random Forest is a machine learning algorithm that consists of an ensemble of simple tree predictors, capable enough of individually producing a result on input of a set of predictor values. In classification, the problem is deduced to a class membership problem, binary or multi-class. The results from these simple tree predictors is in the form of class membership, associating or classifying a set of independent predictor values with the categories of the dependent variable. Each of these tree predictors give a response which is dependent on a set of predictor values.

In Classification problems, Random Forest algorithm measures the average number of votes (responses) for the correct class, exceeding the average vote for other classes in the dependent variable through a margin function. The margin function helps us to associate a confidence measure with the predictions that are essentially done through its help.

The non-faulty class prediction models that were based on Random Forest algorithm are trained on the smell data of the corresponding version and tested on the defect data in order of chronology. The performance of the models obtained are illustrated in Table 8 and figure 2. The results obtained on the analysis of the model on different versions shows that the performance of Random forest based Smell-defect prediction model shows some variation on different versions of Eclipse.

As is clearly evident from the Table 8,the models based on the Random Forest algorithm have an ROC lying in the range of 68.2 % to 80.5%.This implies that there is poor to good perceptiveness among the different models built, in determining the absence of defects through the Random Forest based model.

F-measure, the harmonic mean of precision and recall, also gives an indication of the average performance of the algorithm in predicting the defects. With the values of F-measure lying between 77 % and 83.6 percent, it can be concluded that the model performed well on discriminating the non-faulty classes.

Table 8. Performance measures of Random Forest based Smell-Defect model

Smell Prediction Model	Defects Predicted in	Precision	Recall	F-Measure	ROC
Eclipse 3.2	Eclipse 3.2	0.913	0.673	0.775	0.682
	Eclipse 3.3	0.912	0.696	0.789	0.693
	Eclipse 3.6	0.929	0.685	0.788	0.717
	Eclipse 3.7	0.970	0.692	0.808	0.779
Eclipse 3.3	Eclipse 3.3	0.913	0.728	0.810	0.705
	Eclipse 3.6	0.926	0.718	0.809	0.724
	Eclipse 3.7	0.975	0.732	0.836	0.805
Eclipse 3.6	Eclipse 3.6	0.927	0.677	0.782	0.707
	Eclipse 3.7	0.995	0.703	0.824	0.797
Eclipse 3.7	Eclipse 3.7	1.000	0.705	0.827	0.800

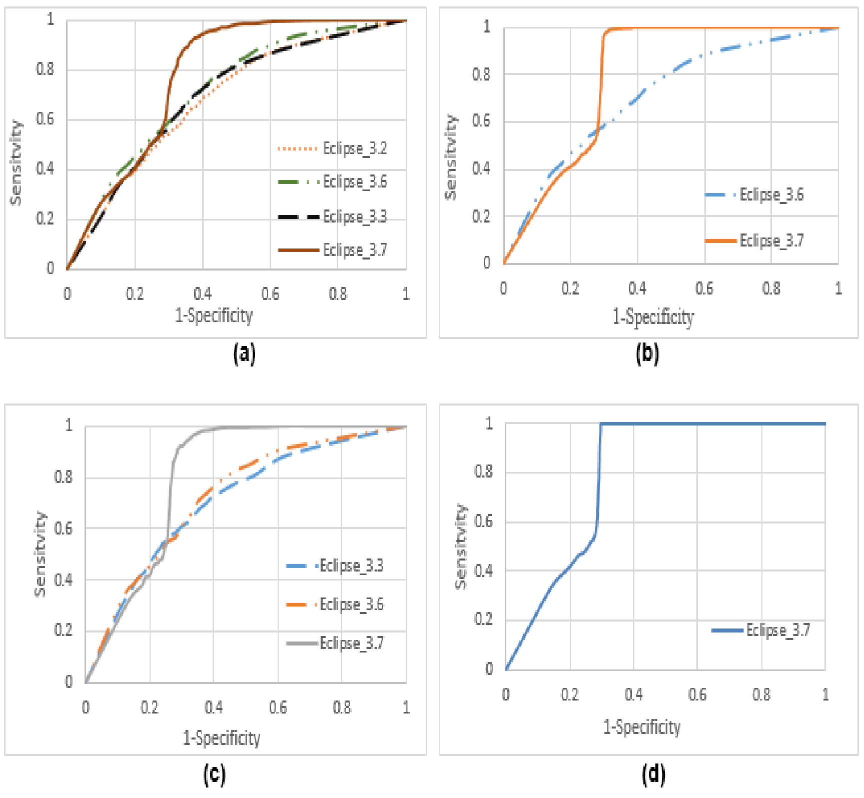

Figure 2. ROC curves on application of Random Forest based prediction model for prediction of non-faulty classes in the subsequent versions. (a)Performance of Eclipse 3.2 based prediction model. (b) Performance of Eclipse 3.3 based prediction model. (c) Performance of Eclipse 3.6 based prediction model. (d) Application of Eclipse 3.7 based prediction model.

After the balancing of data, Random forest algorithm based prediction models do not show significant improvement in the prediction of defects. The performance of Random forest based smell-defect model is given in the Table 9.

Table 9. Performance measures of Random Forest based Smell-Defect model after Data balancing

Smell Prediction Model	Defects Predicted in	Precision	Recall	F-Measure	ROC
Eclipse 3.2	Eclipse 3.2	0.759	0.673	0.714	0.678
	Eclipse 3.3	0.770	0.683	0.724	0.692
	Eclipse 3.6	0.772	0.670	0.717	0.713
	Eclipse 3.7	0.854	0.683	0.759	0.771
Eclipse 3.3	Eclipse 3.3	0.771	0.708	0.738	0.695
	Eclipse 3.6	0.763	0.692	0.726	0.719
	Eclipse 3.7	0.889	0.751	0.814	0.823
Eclipse 3.6	Eclipse 3.6	0.766	0.661	0.710	0.701
	Eclipse 3.7	0.971	0.690	0.806	0.793
Eclipse 3.7	Eclipse 3.7	1.000	0.696	0.820	0.792

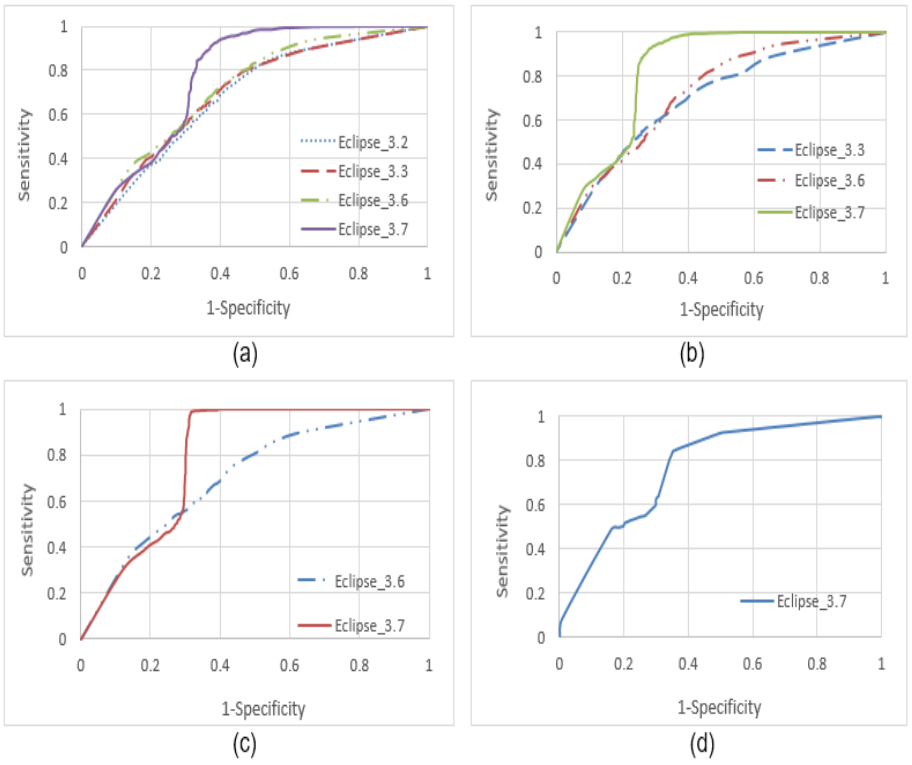

Figure 3. ROC curves on application of Random based prediction model for prediction of non-faulty classes in the subsequent versions using balanced dataset. (a)Performance of Eclipse 3.2 based prediction model. (b) Performance of Eclipse 3.3 based prediction model. (c) Performance of Eclipse 3.6 based prediction model. (d) Application of Eclipse 3.7 based prediction model

Figure 3 is clearly illustrating the fact that the model is performing good and that the data balancing did not significantly improve the performance of the prediction models. The models were rather found to be very much insensitive to the amount of data imbalance present in the original dataset.

J48 based prediction models

J48 is a Java based implementation of C4.5 algorithm in Weka, which, in essence, is an implementation of ID3 (Iterative Dichotomiser 3) algorithm, used to generate a decision tree. J48 is based on ID3 algorithm with some modifications, aimed at improving the disadvantages in the original ID3 algorithm. Basically, ID3 is a decision tree algorithm which provides the decision on the basis of the performance of attribute of a given instance in classifying the instances present in the dataset. It also determines those values of the ranges which give the best results in classification. In the implementation of J48 algorithm, there have been additional changes in the original ID3 algorithm to improve its performance. Some of the improvements are:

- The algorithm is able to handle training data with missing attribute values.
- The algorithm is able to hand different cost attributes.
- It has the capability to prune a decision tree after creating one.
- It can handle the attributes which have discrete and continuous distribution of the values.

The models based on J48 implementation were created on the same lines as RandomForest based models. The training was done on the smell data and the testing on the bug data.

Table 10. Performance measures of J48 based Smell-Defect model

Algorithm	Smell Prediction Model	Defects Predicted in	Precision	Recall	F-Measure	ROC
J48	Eclipse 3.2	Eclipse 3.2	0.825	0.681	0.730	0.648
		Eclipse 3.3	0.818	0.695	0.737	0.659
		Eclipse 3.6	0.845	0.685	0.739	0.665
		Eclipse 3.7	0.888	0.687	0.756	0.698
	Eclipse 3.3	Eclipse 3.3	0.818	0.724	0.758	0.696
		Eclipse 3.6	0.844	0.719	0.764	0.692
		Eclipse 3.7	0.885	0.719	0.779	0.735
	Eclipse 3.6	Eclipse 3.6	0.845	0.686	0.740	0.690
		Eclipse 3.7	0.892	0.693	0.761	0.757
	Eclipse 3.7	Eclipse 3.7	0.889	0.696	0.763	0.764

From Table 10, it can be inferred that values of F-measure and ROC imply that the model performs upto good level in the discrimination of non-faulty classes. The highest value of ROC in this model is 76.4 %. The F-measure values are also appreciative as they lie around 75%, the highest being 77.9%. The model, hence shows good discrimination in predicting non-faulty classes.

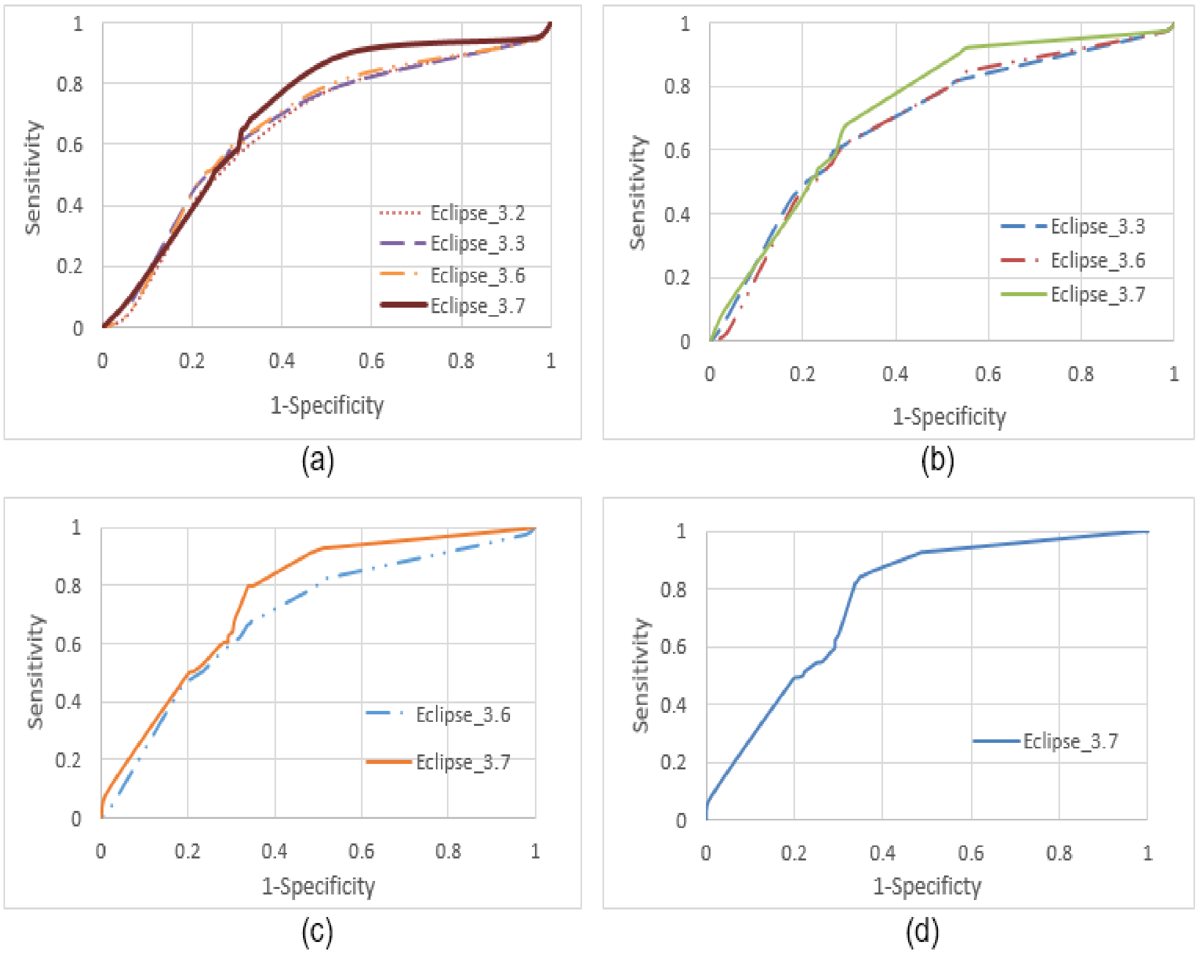

Figure 4. ROC curves on application of J48 based smell prediction model for prediction of non-faulty classes in the subsequent versions. (a)Performance of Eclipse 3.2 based prediction model. (b) Performance of Eclipse 3.3 based prediction model. (c) Performance of Eclipse 3.6 based prediction model. (d) Application of Eclipse 3.7 based prediction model

The graphical illustration of ROC curves also is indicative of the fact that the model performs well in its discrimination of the non-faulty classes.

The same experiment was repeated with the change in the dataset used. The datasets that were used were balanced. The results obtained are listed in the Table 11.

Table 11. Performance measures of J48 based Smell-Defect model after balancing

Smell Prediction Model	Defects Predicted in	Precision	Recall	F-Measure	ROC
Eclipse 3.2	Eclipse 3.2	0.761	0.683	0.720	0.643
	Eclipse 3.3	0.771	0.700	0.734	0.663
	Eclipse 3.6	0.772	0.677	0.722	0.659
	Eclipse 3.7	0.794	0.683	0.734	0.691
Eclipse 3.3	Eclipse 3.3	0.765	0.727	0.746	0.691
	Eclipse 3.6	0.771	0.710	0.739	0.692
	Eclipse 3.7	0.802	0.748	0.774	0.748
Eclipse 3.6	Eclipse 3.6	0.773	0.678	0.722	0.680
	Eclipse 3.7	0.807	0.693	0.745	0.763
Eclipse 3.7	Eclipse 3.7	0.793	0.694	0.740	0.770

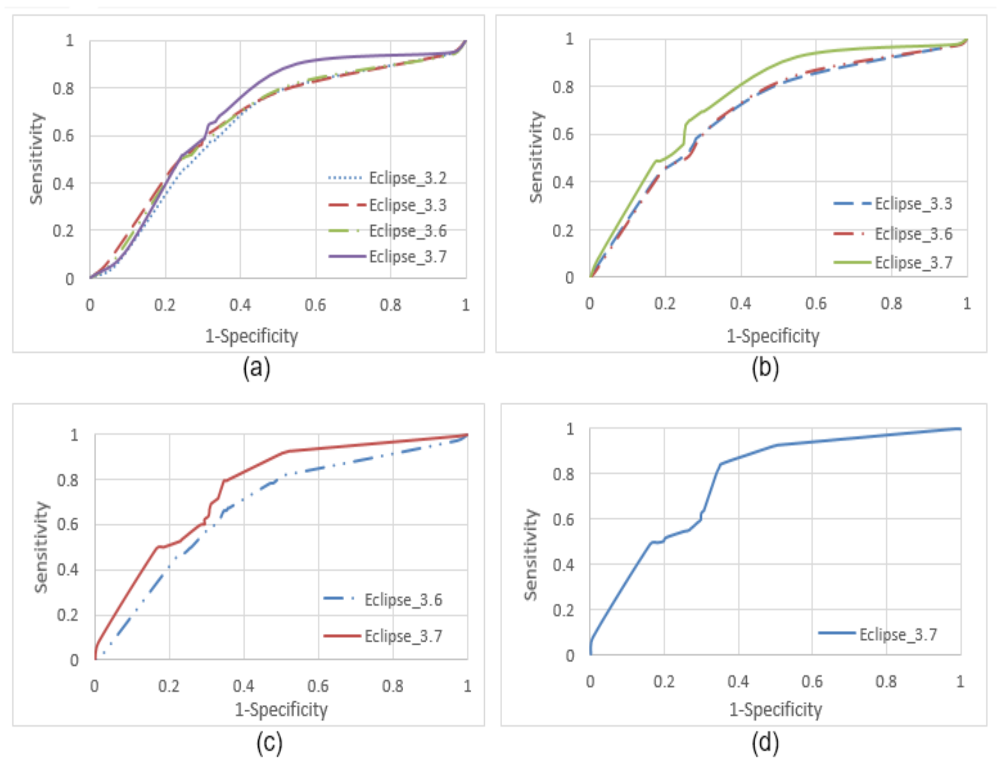

Figure 5. ROC curves on application of J48 based prediction model for prediction of non-faulty classes in the subsequent versions using balanced dataset. (a)Performance of Eclipse 3.2 based prediction model. (b) Performance of Eclipse 3.3 based prediction model. (c) Performance of Eclipse 3.6 based prediction model. (d) Application of Eclipse 3.7 based prediction model

For the balanced datasets, J48 algorithm did not show any significant improvement in the prediction of defective classes. With the average recall and precision values decreasing slightly, the model does not perform any better than the model based on the unbalanced dataset. Overall, the balancing of data does not have significant effect in the increase in the performance of the J48 algorithm.

SVM based Smell-defect models

Support Vector Machine (SVM) is one of the supervised machine learning algorithms which finds its use for classification as well as regression problems. For a given a set of training data, each being labelled in advance according to the category it belongs to, an SVM training algorithm builds a model by assigning new data to the one of the categories. It thus works as a non-probabilistic binary linear classifier. To construct a hyperplane which is optimal for a given problem, SVM uses an iterative training algorithm, which minimizes the error function given as:

1) $$\frac{١}{٢} w^T w + C \sum_{i=١}^{N} \xi$$

Subject to the constraints $y_i(w^T \emptyset(x_i) + b) \geq ١ - \xi_i$ & $\xi_i \geq ٠, \mathrm{i} = ١, \ldots., \mathrm{N}$

Where C is the capacity constant, b is a constant, ξ_i represents parameters for handling non-separable data (inputs), and w is the vector of coefficients. The index i goes from 1 to N, labeling the training cases. The kernel $\emptyset$ is a transformation function, transforming data from the input (independent) to the feature space. To avoid over-fitting, parameter C should be chosen with care.

Kernel function is given by:

2) $$K(X_i, X_j) = \begin{cases} X_i . X_j Linear \\ (\gamma X_i X_j + C)^d Polynomial \\ exp(-\gamma |X_i - X_j|)^{٢} RBF \\ tanh(\gamma X_i X_j + C) Sigmoid \end{cases}$$

Where , $K(X_i, X_j) = \emptyset(X_i) . \emptyset(X_j)$, the kernel function, is a dot product of input data points mapped into higher dimensional feature space by transformation $\emptyset$. γ is an adjustable parameter of some kernel functions.

Table 12 and figure 6 elucidate the fact that SVM based models show good value for F-measure, which indicates that it is quite effective in discriminating the non-faulty classes. As can be seen from the individual ROC curves in the figure 6, the ROC values show a comparatively low dip, which reflects that the overall model has poor perceptiveness. Given the results, it can be inferred that the model performs good, although not too good in prediction of non-faulty classes. SVM has been found to be sensitive to data imbalance and feature selection and as such may require more tuning for better results.

Table 12. Performance measures of SVM based Smell-Defect model

Algorithm	Smell Prediction Model	Defects Predicted in	Precision	Recall	F-Measure	ROC
SVM	Eclipse 3.2	Eclipse 3.2	0.821	0.682	0.730	0.625
		Eclipse 3.3	0.816	0.695	0.737	0.641
		Eclipse 3.6	0.846	0.690	0.743	0.652
		Eclipse 3.7	0.878	0.679	0.750	0.627
	Eclipse 3.3	Eclipse 3.3	0.818	0.728	0.761	0.647
		Eclipse 3.6	0.845	0.721	0.765	0.651
		Eclipse 3.7	0.877	0.713	0.774	0.628
	Eclipse 3.6	Eclipse 3.6	0.843	0.690	0.742	0.643
		Eclipse 3.7	0.879	0.683	0.752	0.630
	Eclipse 3.7	Eclipse 3.7	0.879	0.683	0.753	0.630

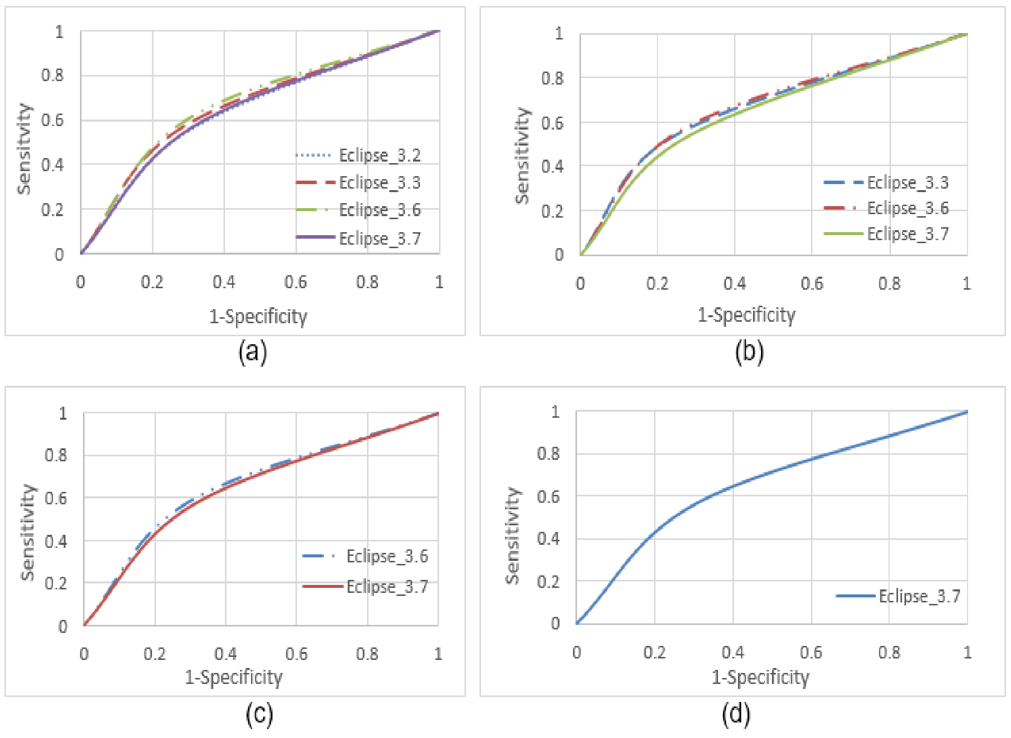

Figure 6. ROC curves on application of SVM based prediction model for prediction of non-faulty classes in the subsequent versions. (a)Performance of Eclipse 3.2 based prediction model. (b) Performance of Eclipse 3.3 based prediction model. (c) Performance of Eclipse 3.6 based prediction model. (d) Application of Eclipse 3.7 based prediction model.

The dataset, when balanced, does not considerably affect the performance of the model. The performance of the model is, by standards, not too good and needs improvement through feature selection and other tweaks. Table 13 lists the individual performances of each of the models based on a certain software version. The ROC values, if compared with those in Table

12 don’t show much deviation. Figure 7 is also indicative of the relative performance of the model on different data.

Table 13. Performance measures of SVM based Smell-Defect model after balancing

Algorithm	Smell Prediction Model	Applied on	Precision	Recall	F-Measure	ROC
SVM	Eclipse 3.2	Eclipse 3.2	0.652	0.633	0.640	0.612
		Eclipse 3.3	0.670	0.653	0.659	0.632
		Eclipse 3.6	0.678	0.656	0.663	0.642
		Eclipse 3.7	0.664	0.646	0.653	0.626
	Eclipse 3.3	Eclipse 3.3	0.673	0.666	0.669	0.634
		Eclipse 3.6	0.679	0.668	0.672	0.642
		Eclipse 3.7	0.683	0.680	0.681	0.645
	Eclipse 3.6	Eclipse 3.6	0.673	0.653	0.660	0.636
		Eclipse 3.7	0.665	0.646	0.653	0.626
	Eclipse 3.7	Eclipse 3.7	0.665	0.646	0.653	0.626

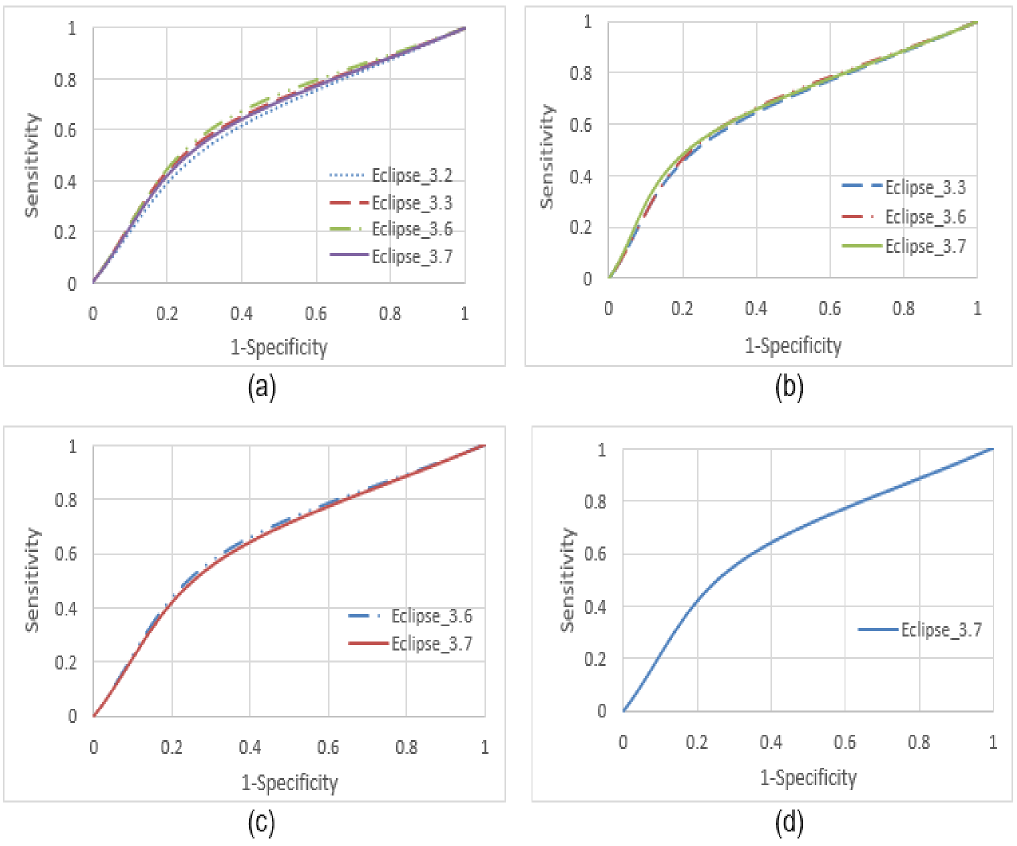

Figure 7. ROC curves on application of SVM based prediction model for prediction of non-faulty classes in the subsequent versions using balanced dataset.(a)Performance of Eclipse 3.2 based prediction model. (b) Performance of Eclipse 3.3 based prediction model. (c) Performance of Eclipse 3.6 based prediction model. (d) Application of Eclipse 3.7 based prediction model.

Threats to Validity

This section deals with the discussion of the Internal and External threats to validity of our approach. While the threats to the internal validity are concerned with the correctness of the experimental outcome, the threats to external validity are concerned with the extendibility of the results obtained.

Threats to Internal Validity

The research tries to approach the problem in the most viable way. Some of the factors would have influenced the results and pose threat to the validity. A limitation in the bug data collection is that it is done by the master level students and not by some professionals. Putting students to the task of bug data collection is justified as many studies have used students as substitutes of professionals in software engineering (Fontana et al., 2016).Additionally, the students were beforehand briefed about the process and adequately trained to perform the task. Another threat to validity is the nature of smells being subjective. As the smells are subjective in nature and extracted by a single tool through the analysis, the smells may show some bias to the underlying detection strategy, and in turn, metrics. This threat has been countered by using different tools for smell extraction and metrics extraction. The metrics extraction software is independent of the smell extraction software and hence the smell definitions become somewhat independent of the associated metrics. The metric definitions in both the tools vary and such this problem is alleviated to a good level. Although, it is suggested that the future experiments should include a broader and accumulated definition of code smells which encompasses variety of underlying methodologies. Another query that can be raised is that the interdependence of smells and/or dominance of smells may have impacted the smell labels in the modules. This problem is tackled by taking a balanced combination of class and method type smells, each one being a prominent in their impact(Fontana et al., 2016), and the final label being inclusive of both the types of smells.

Threats to External Validity

Application and generalisation of the results to other software is an important threat to the external validity. This threat has been encountered through selecting an open source, industry sized software for analysis. This mitigates this threat to a large extent. As the software is written in java language, its generalizability is limited to such similar software as a certain programming language affects many metric values. This limitation can be overcome by including different software from different domains and of varied platforms. The size of the actual modules observed could also limit the generalisability of the results to the other software. The inclusion of a large scale data set can enhance the generalisation of the results inferred. The experiment was limited by time and human resources and as such, the possibility of extraction of huge dataset was excluded. As most of the work in filing the bugs was done manually, it was not possible to extend the analysis further. Further, the amount of data collected was sufficient enough for machine learning algorithms to get trained on, the problem being binary classification. Future works may include large corpus of datasets of diverse platforms so as to generalise the result to greater extent. Another objection would be using ROC and F-measure as performance measures for the experiment. ROC and F-measure are the parameters that have been found to be quite reflective of the performance of the models in the machine learning and many researchers have used and supported the use of these performance measures as enough empirical evidence for concluding (Sang, De Meulder, 2003; Provost, Tom, Kohavi, 1998).

Conclusion and Future Work

This study focuses on the viability of code smell based model as predictors of non-faulty classes, with supervised machine learning algorithms in an industry sized, object-oriented software. It examines conjunction of various code smells for their effectiveness in maintenance of software by predicting the classes which require less or no attention in the maintenance of the software. The study also tries to find out whether the code smells can also be taken as a factor in prediction of non-faulty classes so as to enhance the code maintainability. The prediction of non-faulty classes, or the classes which are very uncertain to contain bugs in near future can help in the preventive maintenance by providing an idea of prioritising the classes which may be prone and leaving the classes which are not. The study concludes that code smells can be an effective factor in determining the defects in a software. As code smells may tend to be more objective, they can be more important in software maintenance. The effective combination of a standardised code smell set, objectively derived, can further improve the prediction of non-faulty classes. As the methodology used to extract smells and the consideration of smells has a considerable impact on the outcome of this study, as the smells are subjective in nature, future studies should include different treatment of smells vis-à-vis detection and grouping strategy. This study can also be used as a baseline to further explore the role of code smells in the prediction of absence of software bugs.

References

Bengio, Y., & Grandvalet, Y. (2004). No unbiased estimator of the variance of k-fold cross-validation. *The Journal of Machine Learning Research, 5*, 1089-1105.

Cartwright, M., & Shepperd, M. (2000). An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering, 26*(8), 786-796.

Catal, C. (2011). Software fault prediction: a literature review and current trends. *Expert Systems with Applications*, *38*, 4626-4636.

Cohen, P., & Jensen, D. (1997). Overfitting explained. *Preliminary Papers of the Sixth International Workshop on Artificial Intelligence and Statistics* (pp. 115-122). Self published,Printed proceedings distributed at the workshop.

Fontana, F. A., Mäntylä, M. V., Zanoni, M., & Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empir Software Eng*, 1143-1191.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (2002). In *Refactoring: Improving the Design of Existing Code* (pp. 63-72). Addison Wesley.

Gueheneuc, Y. G., Sahraoui, H., & Zaidi, F. (2004). Fingerprinting design patterns. *11th Working Conference on Reverse Engineering* (pp. 1095-1350). Deft: IEEE Computer Society .

Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2011). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering, 38*(6), 1-31.

Kapila, H., & Singh, S. (2013). Analysis of CK Metrics to predict Software Fault-Proneness using Bayesian Inference. *International Journal of Computer Applications, 74*(2), 1-4.

Kim Sang, E.F.T., & De Meulder, F. (2003). Introduction to the CoNLL-2003 Shared Task: Language-independent Named Entity Recognition. *Proceedings of the Seventh Conference on Natural Language Learning* (pp. 142-147). Edmonton, Canada: Association for Computational Linguistics.

Lessmann, S., Baesens, B., Mues, C., & Pietsch, S. (2008). Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering, 34*(4), 485-496.

Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Gùehèneuc, Y.G., & Aimeur, E. (2012). SMURF: A SVM-based Incremental Anti-pattern Detection Approach. *19th Working Conference on Reverse Engineering* (pp. 466-475). Kingston, Canada: IEEE.

Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Gùehèneuc, Y.G., Antoniol, G., & Aimeur, E. (2012). Support vector machines for anti-pattern detection. *27th IEEE/ACM International Conference on Automated Software Engineering* (pp. 278-281). Essen,Germany: ACM.

Provost, F. J., Tom, F., & Kohavi, R. (1998). The case against accuracy estimation for comparing induction algorithms. *International Conference on Machine Learning, 98*, 445-453.

Singh, S., & Kaur, S. (2017). A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*, 9(4), 2129-2151.

Singh, Y., Kaur, A., & Malhotra, R. (2010, March). Empirical validation of object-oriented metrics for predicting fault proneness models. *Software Quality Journal, 18*(3), 3-35.

Stone, M. (1974). Cross-Validatory Choice and Assessment of Statistical Predictions. *Journal of the Royal Statistical Society,Ser B, 36*(2), 111-147.

Yamashita, A., & Moonen, L. (2013). Exploring the impact of inter-smell relations on software maintainability: An empirical study. *Proceedings of the 2013 International Conference on Software Engineering* (pp. 682-691). IEEE Press.

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

Bibliographic information of this paper for citing:

Ali Reshi, Junaid, & Singh, Satwinder (2018). Investigating the Role of Code Smells in Preventive Maintenance. *Journal of Information Technology Management*, 10(4), 41-63.

Copyright © 2018, Junaid Ali Reshi and Satwinder Singh.