Predicting Software Change-Proneness with Code Smells and Class Imbalance Learning

Arvinder Kaur, Kamaldeep Kaur and Shilpi Jain
USICT
Guru Gobind Singh Indraprastha University
Delhi, India
arvinderkaurtakkar@yahoo.com, kdkaur99@gmail.com, shilpijain0203@gmail.com

***Abstract*—The objective of this paper is to study the relationship between different types of object-oriented software metrics, code smells and actual changes in software code that occur during maintenance period. It is hypothesized that code smells are indicators of maintenance problems. To understand the relationship between code smells and maintenance problems, we extract code smells in a Java based mobile application called MOBAC. Four versions of MOBAC are studied. Machine learning techniques are applied to predict software change-proneness with code smells as predictor variables. The results of this paper indicate that codes smells are more accurate predictors of change-proneness than static code metrics for all machine learning methods. However, class imbalance techniques did not outperform class balance machine learning techniques in change-proneness prediction. The results of this paper are based on accuracy measures such as F-measure and area under ROC curve.**

Keywords—Code smells; Exception handling smells; Software change-proneness; Machine learning techniques; Class imbalance learning

I. INTRODUCTION

Efficient software maintenance is dependent on high quality software code [1]. Poor implementation of code leads to a difficult to maintain code. Fowler and Beck [2] coined the term code smell as symptom of design and code problems in software. It is hypothesized that software modules with high degree of code smells are prone to complex and more changes during maintenance period. Khomh et al. [3] investigated 29 code smells on 9 releases of Azeuras and 13 releases of Eclipse. They found that object-oriented software classes having code smells are more change-prone than others. The relationship between codes smells and change-proneness has not been studied in domain of mobile application software development. Further the relationship between code smells and change-proneness is complex, therefore this motivates us to use machine learning techniques for predicting software changes during maintenance period with code smells as predictor variables.

Three research questions are addressed in this paper:

- Are code smells better predictors of change-proneness than static code metrics?
- Which machine learning techniques are more accurate for predicting change-prone modules in mobile application software?
- Are class imbalance learning techniques more accurate for predicting change-prone modules?

A number of code smells which are indicators of code quality problems have been defined in the literature [4]. In this paper 10 code smells have been studied for predicting software change-proneness These smells are Feature Envy, God Class and Long Method [2, 23]. In addition to this 7 exception handling code smells are studied. They are Empty Catch Block, Dummy Handler, Unprotected Main Program, Nested Try Statement, Careless Cleanup, Overlogging and Exceptions Thrown from Finally Block [5, 22]. Only 10 code smells are considered because rigorously tested and validated open-source code smell detection tools are not available for other code smells. To the best of our knowledge no previous study has considered prediction of software changes using exception handling code smells as predictors. The rest of this paper is organized as follows: Section 2 presents related work and literature review. Section 3 presents predictor and response variables. Section 4 presents empirical data collection. Section 5 presents research methodology. Section 6 presents result analysis and section 7 presents conclusions and future work.

II. RELATED WORK

Fontana et al. [6] compared different machine learning algorithms to detect code smells in source code of 74 software systems using 16 machine learning algorithms and detecting 4 different types of code smells. The code smells identified were Feature Envy, Large Class, Long Method and Data Class. The authors [6] used two advisors namely PMD and iPlasma. On the basis of performance parameters like accuracy, F-measure and ROC it was concluded that J48 and Random Forest algorithm were best whereas Support Vector Machine (SVM) algorithm was worst in code smell detection. Liu et al. [7] proposed a new detection and resolution sequence for different types of bad code smells. Chen et al. [5] studied exception handling code smells on a real world banking application and used these to propose refactoring to eliminate the code smells. They argued that bug fixing drives exception handling refactoring in best possible way.

978-1-5090-2029-4/16/$31.00 @2016 IEEE

In another study, Dag I.K. Sjøberg et al. [8] found that the effects of 12 code smells on maintenance effort were limited. They suggested focusing on reducing code size and better work practices to limit the number of changes in order to reduce maintenance effort. Aiko Yamashita et al. [9] found that code smells have minor effect on the overall maintainability of the system. They also found that the interactions between collocated smells and coupled smells are important.

Hall et al. [10] investigated the relationship between faults and five smells in code: Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man. They concluded that arbitrary refactoring is unlikely to reduce fault proneness. To the best of our knowledge only one study [3] investigates the relationship between code smells and change-proneness.

No study investigates the relationship between exception handling code smells and change-proneness. Exception handing design is considered to be one of the less understood parts of software development [5]. If a programmer fails to write robust exception handling code, it is likely to give rise to bugs and changes during maintenance phase. Traditional method of error handling leads to spaghetti code and results in poor code quality [11].

In the current empirical study it is investigated whether exception handling code smells lead to more change-prone code or not. Machine learning techniques are used to predict change-proneness using code smells as predictors. Previous studies on change-proneness prediction are based on static code metrics as predictors [3, 19].

Wang and Yao [12] concluded that the training datasets for fault prediction and change-proneness prediction are affected by imbalanced data distribution as the number of faulty or change-prone modules is much less than non-faulty or non-change-prone modules in any software system. However, their study is based on static code metrics as defect predictors. There are few more studies that apply class-imbalance learning in static code metrics based change-proneness prediction [13-18]. To the best of our knowledge no previous research has considered class imbalance learning in context of code smells based change-proneness prediction.

Malhotra and Khanna [19] studied the relationship between object-oriented metrics and change-proneness. Performances of machine learning methods were compared with Logistic Regression. RFC metric was found to be an important indicator of change-proneness. Bagging and Random Forests outperformed Logistic Regression method. However, their investigation was based on source code metrics only. Our study shows that code smells are better predictors for all machine learning methods.

III. Predictor and Response Variables

A. Predictor Variables

In this study two sets of experiments are carried out. In first set of experiments, object-oriented (OO) software metrics also known as static code metrics, extracted by CKJM tool [20] are used as predictor variables for predicting change-prone modules. These OO metrics are related to size, inheritance, complexity, encapsulation, cohesion, coupling and abstraction. The 18 object-oriented metrics (OO) are listed in Table I. In second set of experiments, 10 code smells [2,5,22,23] are used as predictor variables to predict change-prone modules. The 10 code smells are listed in Fig. I and are explained as under.

TABLE I. Object Oriented Software Metrics

Metric type	Metric Name
Size metrics	Lines of Code (LOC), Number of Public Methods (NPM)
Complexity metrics	Weighted Method per Class (WMC), Average Method Complexity (AMC)
Encapsulation metrics	Data Access Metric (DAM)
Inheritance metrics	Depth of Inheritance Tree (DIT), Number Of Children (NOC)
Cohesion Metrics	Lack of Cohesion in Methods (LCOM), Lack of Cohesion in Methods 3 (LCOM3), Cohesion Among Methods (CAM)
Coupling Metrics	Response For Class (RFC), Coupling Between Object (CBO), Afferent Coupling (Ca) , Efferent Coupling (Ce), Inheritance Coupling (IC), Coupling Between Methods (CBM)
Abstraction	Measure Of Aggregation (MOA), Measure Of Functional Abstraction (MFA)

B. Response Variable

After software is released new source lines of code may be added, deleted or lines may be modified due to corrective, adaptive or perfective maintenance. If a software module undergoes changes in lines of code, the module is labelled as change-prone. Otherwise it is labelled as non-change-prone. Thus change-proneness of a software module is a binary variable. The response variable used in this study is Change-Proneness (CP). CP variable is calculated by identifying such classes in the Java application which are modified from one software release to another.

IV. Empirical Data Collection

This section gives a detailed description of the procedure undertaken to collect data empirically. Android based open source software called MOBAC (Mobile Atlas Creator) for change-proneness prediction is taken. This open source software is coded in Java whose source code can be downloaded from http://sourceforge.net/projects/mobac/. A class is said to be change-prone if there are changes in the lines of code in that class in different software versions. The response variable CP is found by comparing 4 versions of the MOBAC software. The versions and the release dates for MOBAC are shown in Table II.

TABLE II. Software Versions Release Dates

Version	Date(Last date modified)
MOBAC v1.9.1	2011-09-09
MOBAC v1.9.7	2012-05-17
MOBAC v1.9.16	2014-02-06
MOBAC v2.0 (alpha 3)	2015-04-16

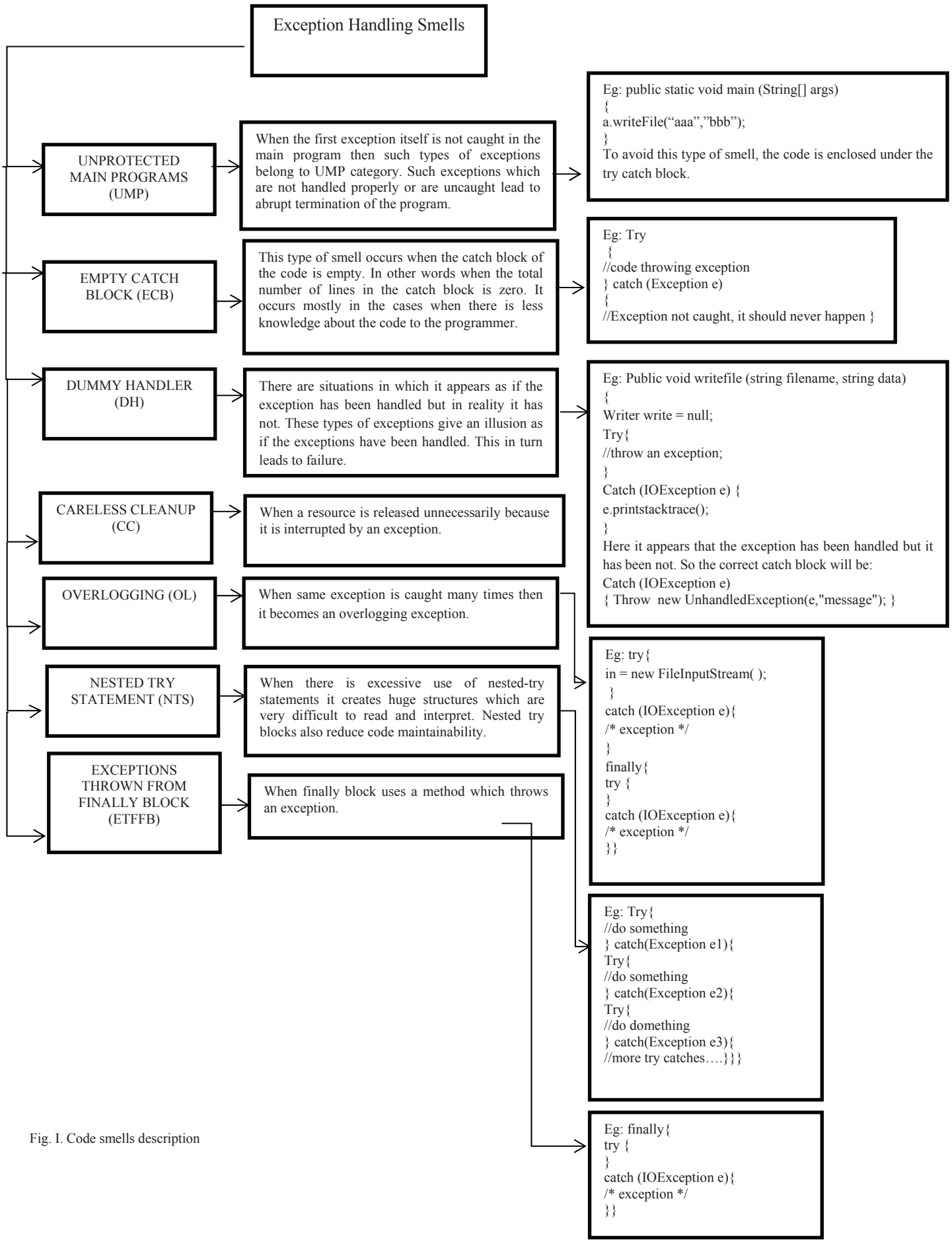


Fig. I. Code smells description

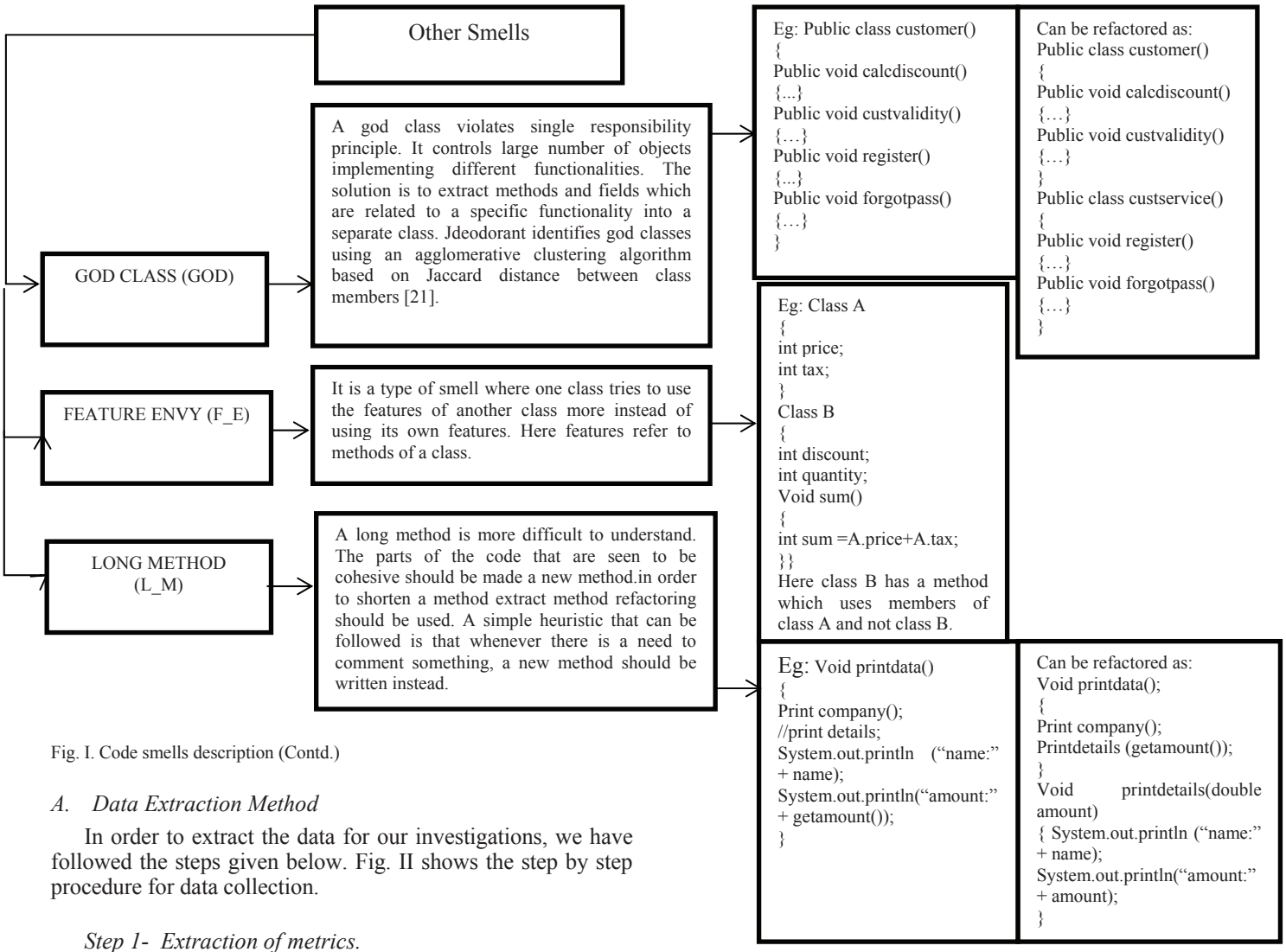


Fig. I. Code smells description (Contd.)

A. Data Extraction Method

In order to extract the data for our investigations, we have followed the steps given below. Fig. II shows the step by step procedure for data collection.

Step 1- Extraction of metrics.

In this paper, software metrics are calculated using CKJM tool extended version 2.2 [20]. This tool is a command prompt based tool which is useful in calculating 18 object-oriented software metrics. We have calculated software metrics for all the 4 versions of MOBAC source code separately.

Step2- Smell Identification and Extraction

Software code smells are an indication of faulty software. To identify the classes which have code smells, we have used two Eclipse plugins - Robusta [22] and Jdeodorant [23]. Robusta tool identifies seven exception handling smells namely- ECB, DH, UMP, ETFFB, CC, NTS and OL [5]. Jdeodorant tool is used to calculate other smells namely- F_E, GOD and L_M [2]. A brief explanation of code smells is listed in Fig. I.

Step3- Comparing Versions

To calculate change between two software versions, only those classes which belong to both the versions and are change-prone are extracted.

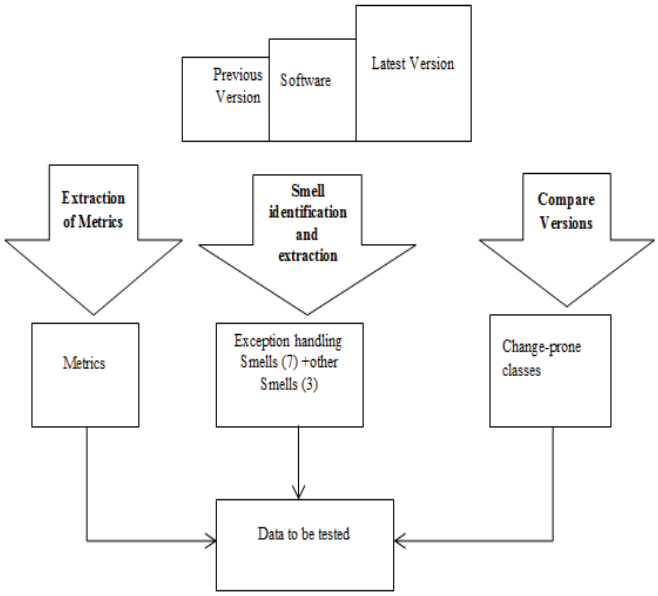


Fig. II. Data collection procedure

Step 4 – Identifying Change-Prone classes

After software is released it is updated with new features regularly. A Java class is said to be changed if SLOC are added or deleted in that class. To identify the classes which are affected by such changes, we have compared versions by analyzing commits on GITHUB repository. For our MOBAC source code we have used commits of https://github.com/larroy/mobac.

TRUE- If a Java class was changed in the next version then CP variable was made true.

FALSE- All the other remaining non-change-prone classes were marked false.

V. MODELLING METHODOLOGY

This section discusses about the methodologies [26] used to determine the relationship between different types of object-oriented software metrics, code smells and actual changes in software code. Algorithms are applied at 10-cross validation.

A. Class Balance Learning Methods

1) Naïve Bayes(NB)

In machine learning, Naïve Bayes classifier belongs to the class of probabilistic classifiers. These are the classifiers which predict the output class on the basis of probability distribution. Naïve Bayes uses Bayes theorem assuming that the features are strongly independent. Features are regarded as independent features if the presence or absence of one feature does not affect the presence or absence of another feature.

2) Bayes Network (BN)

It is a directed acyclic graph (DAG) model where each random variable is represented as a node and their conditional probabilities as directed edges. If two nodes are not connected in a directed acyclic graph, it means that they are not conditionally dependent on each other. Bayes Networks can be used for inferring from unobserved variables, parameter learning and structure learning.

3) Sequential Minimal Optimization (SMO)

SMO classifiers are used to classify support vector machines (SVM).While training SVM's, they suffer from large quadratic programming (QP) problems. These problems can be solved by using sequential minimal optimization training method. SMO breaks the large unsolvable QP problems into smaller solvable QP problems.

4) Decision Table (DT)

Decision table is a classification algorithm which does not assumes that the features as independent. It has a visualizer and an inducer. The inducer uses an algorithm to identify the most important feature whereas the visualizer models the result graphically using pie charts. For a given dataset, it forms a hierarchy of features such that there are two features at each level.

5) Radial Basis Function Network (RBFnet)

The radial basis function network is a classification algorithm which uses k-means algorithm to determine the basis function. The k means algorithm finds the distances from a point C called the center. For discrete class problems logistic regression is used for learning whereas for numeric class problems linear regression is used for learning.

6) Locally Weighted Learning (LWL)

It is lazy learning method which builds models around a particular point of interest. This mode does not create a single global model for all the data points. Instead it creates models locally. For each data points weights are assigned. These weights determine the influence of a data point in the neighborhood around a query point. If a data point is close to the current query point then it will be given higher weight as compared to a faraway data point.

7) Logistic Regression (LR)

It is a model which determines the relationship between the response variable and the predictor variables using a logistic function. The response variable can be categorical in nature. Categorical variable is a variable which can acquire one value from a fixed set of values.

8) Simple Cart (SCart)

It is classification algorithm which works by generating binary decision trees. A binary tree produces an output containing only two children. The splitting attribute is decided by calculating entropy. This method can handle missing values.

B. Class Imbalance Learning Methods

When we have huge datasets, the class balance machine learning algorithms don't work properly due to huge differences in the number of instances of one type (majority classes) as compared to another (minority classes). Such datasets are called imbalanced datasets (IDS). This led to the introduction of a new field in data mining methods called as class imbalance learning (CIL) techniques. Class imbalance learning techniques are used to balance the majority to minority class ratio. Balancing the minority classes and majority classes is important to reduce the skewness in the data distribution.

1) Data level

In data level CIL techniques the original datasets are altered by balancing the majority to minority ratio. The most widely used method is called Resampling. Resampling is a method of reconstructing the original dataset by over sampling or under sampling.

- *Over sampling* is a resampling technique where the minority classes are duplicated to improve the overall distribution of minority classes among majority classes. Sometimes they can lead to over-fitting. Different oversampling algorithms include [24]: Wilson editing (WE), Cluster Based Oversampling (CBOS), Random over sampling (ROS), Synthetic minority over sampling (SMOTE) and Borderline SMOTE (BSM). For our case study we have used SMOTE 300 with the base learners.
- *Under sampling* is a method of removing some of the majority classes to balance out the majority to minority class ratio. Different under-sampling algorithms include [24]: Random under sampling (RUS) and one sided selection (OSS). For our case study we have used RUS 1.5 using spread subsample filter with the base learners.

2) Algorithm Level

In the algorithm level class imbalance learning method the original data set is not modified. They use already defined algorithms to increase the importance of minority classes compared to majority classes. Classifier ensembles are algorithm level CIL technique. Ensembles are a group of base learners which are applied together on the same dataset to improve the performance of individual base learning algorithms. In ensembling individual outputs of each classifier (learner) is combined in some way to compute an overall combined output. There are four types of ensembling techniques: Bagging (Bootstrap Aggregation), Voting, Boosting and Stacking (stacked generalization). For this case study bagging is used with base learners like Yun Qian et al. [25].

- *Bagging (Bootstrap Aggregation)* is a technique which creates new datasets from the original training dataset by using combinations with repetitions. If we have a training data set T of size N, then bagging creates new datasets T(i) having same size N as the original dataset using bootstrapping i.e. sampling with replacements. Replacements mean that the sample can have two instances of similar type.

VI. Result Analysis

This section describes the analysis performed on the MOBAC application to find out the relationship between code smells, object-oriented metrics and changes. The analysis is performed at two levels: class-balance learning and class imbalance learning (CIL). We have applied machine learning algorithms using WEKA (3.6.13) [26]. For non CIL analysis: Naïve Bayes (NB), Bayes Network (BN), Sequential Minimal Optimization (SMO), Decision Table (DT), Logistic Regression (LR), Radial Basis Function Network (RBFnet), Locally Weighted Learning (LWL) and Simple cart (SCART) are used. Whereas for class imbalance learning, oversampling is done using SMOTE (300) and under sampling is done using spread subsampling (RUS) techniques. WEKA [26] classifiers are used as base learners for bagging. The following performance measures are used to evaluate the correctness and performance of the applied change-prone model:

- *ROC analysis:* To evaluate the performance of the different methods used we have recorded their Area under the ROC curve (AUC) values. ROC stands for Receiver operating characteristics curve. It plots 1-specificity versus sensitivity.
- *F-measure:* it is the harmonic mean of recall and precision. Recall measures the percentage of correct items that are selected. Precision refers to the percentage of selected items that are correct. F- measure (F-Score) combines the two.

  F-measure =2(Precision * Recall)/(Precision + Recall)

A. Result analysis for MOBAC 1.9.1

After applying machine learning algorithms on MOBAC 1.9.1, Tables III and IV shows the results. Table III gives details about number of smells of each type which are found in

TABLE III. Change and Smell Statistics for All Versions

NAME	MOBAC 1.9.1		MOBAC 1.9.7		MOBAC 1.9.16	
	TRUE	*FALSE*	*TRUE*	*FALSE*	*TRUE*	*FALSE*
ECB	27	1085	27	1123	31	1145
DH	37	1075	38	1112	41	1135
UMP	9	1103	9	1141	8	1168
NTS	20	1092	20	1130	21	1155
CC	32	1080	34	1116	36	1140
OL	5	1107	5	1145	5	1171
ETFFB	5	1107	6	1144	7	1169
GOD	71	1041	69	1081	73	1103
F_E	26	1086	28	1122	28	1148
L_M	126	986	130	1020	139	1037
CP	69	1043	123	1027	46	1130

MOBAC 1.9.1. Table III shows that maximum number of exception handling smells was Dummy Handler and the least were OL and ETFFB. While the maximum number of non-exception handing smell was Long Method. It has 69 change-prone classes out of 1112 classes. Therefore the dataset is skewed which motivates us to use class imbalance learning techniques. Following observations were made from Table IV:

In class-balance learning

- If metrics are taken as predictor variables then LR has maximum AUC value of 0.677 and maximum F-measure of 0.91. SMO has minimum AUC value of 0.5 and NB has minimum F-measure of 0.736.
- If all the smells are taken as predictor variables then RBFnet outperforms all the other methods with AUC value 0.75 and F-measure 0.917.SCART has least AUC and F-measure.

In class-imbalance learning

- If the data is over sampled and bagging is applied on the base learners then LWL gives the best results as compared to other classifiers with AUC value 0.745 using metrics as predictor variables. SMO has maximum F-measure. If data is under sampled, DT has maximum AUC and F- measure of 0.716 and 0.682 respectively.
- If smells are taken as predictor variables and data is oversampled, then SMO outperforms all the classifiers with AUC value 0.756 and F-measure 0.849. In under sampling, all NB, BN and RBFnet perform equally well with AUC 0.761.DT has best F-measure value of 0.782.

B. Result analysis for MOBAC 1.9.7

This section gives the results for MOBAC 1.9.7. From Table III, it can be observed that Dummy Handler smells are in majority and OL are in minority. And Long Method occurs more frequently as compared to other smells. It has 123 change-prone classes out of 1150 classes. The observations made from Table V are described as under.

In class-balance learning

- LR gives maximum AUC and F-measure of 0.756

and 0.872 respectively if metrics are taken as predictor variables. SMO and NB perform worst in terms of AUC and F-measure.

- Taking all the smells as predictor variables and AUC as performance measure, RBFnet performs best. NB and BN have maximum F-measure 0.9. LWL has minimum F-measure 0.879.

In class-imbalance learning

- If metrics are used as predictor variables, LWL gives best results when data is over sampled using SMOTE 300 and bagging having AUC 0.773. RBFnet has maximum F-measure. If the data is under sampled, DT has maximum AUC and LR has maximum F-measure.
- NB and SMO with AUC 0.801 and F-measure 0.844 show best results when data is oversampled taking all the smells as predictors. If the data is under sampled, SMO has maximum AUC and RBFnet has maximum F-measure.

C. Result analysis for MOBAC 1.9.16

This section gives the results for MOBAC 1.9.16.Table III shows that the code is affected by Dummy Handler exception handling smells and Long Method smells. MOBAC 1.9.16 has 46 change-prone classes out of 1176 classes. The following conclusions were drawn from Table VI.

In class-balance learning

- Using metrics as predictors LR gives maximum AUC 0.691 and F-measure 0.948. SMO and NB have least AUC and F-measure respectively.
- RBFnet has AUC values 0.805 if all smells are taken as predictors. Whereas NB and BN have maximum F-measure.

In class-imbalance learning

- Taking metrics as predictor variables and oversampling the data, RBFnet has best AUC value 0.755 and SMO with maximum F-measure 0.801. If data is under sampled then LR has maximum AUC and F-measure 0.649 and 0.645 respectively.
- Taking all the smells as predictors, for oversampling SMO outperforms all base learners with AUC 0.811 and F-measure 0.868.If data is under sampled, SMO has best AUC 0.779 while DT and LWL have maximum F-measure.

D. Discussion of results

From the analysis of the results for the three versions of MOBAC (1.9.1,1.9.7 and 1.9.16), it can be concluded that smells are better predictors of change-proneness as compared to object-oriented software metrics on the basis of AUC measure. For class balance learning, taking metrics as predictors LR classifier is the best method for classifying MOBAC all versions with AUC 0.677, 0.756, 0.691 and F-measure 0.91, 0.872 and 0.948, respectively. Taking smells as indictors for change-proneness, RBFnet is the best method having highest accuracy.

Using class imbalance learning, smells as predictors and data oversampling, SMO gives the maximum AUC and F-measure in all the MOBAC versions. Class imbalance learning does not always give better results as compared to class balance learning. Classes which are change-prone are more

TABLE IV. RESULTS OF MOBAC 1.9.1

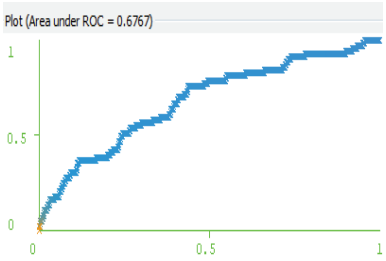
METHOD	METRICS		ALL SMELLS	
	AUC	*F-measure*	*AUC*	*F-measure*
NB	0.643	0.736	0.748	0.914
BN	0.598	0.902	0.748	0.915
SMO	0.5	0.908	0.529	0.916
DT	0.492	0.908	0.58	0.909
LR	0.677	0.91	0.736	0.916
RBFnet	0.582	0.908	0.75	0.917
LWL	0.669	0.908	0.721	0.908
SCART	0.492	0.908	0.492	0.908
SMOTE300+NB+BAG	0.685	0.489	0.743	0.807
SMOTE300+SMO+BAG	0.504	0.702	0.756	0.849
SMOTE300+RBFnet+BAG	0.701	0.698	0.747	0.827
SMOTE300+LWL+BAG	0.745	0.698	0.75	0.833
RUS1.5+NB+BAG	0.66	0.626	0.761	0.753
RUS1.5+BN+BAG	0.691	0.625	0.761	0.759
RUS1.5+SMO+BAG	0.658	0.613	0.755	0.755
RUS1.5+DT +BAG	0.716	0.682	0.753	0.782
RUS1.5+LR +BAG	0.638	0.605	0.746	0.746
RUS1.5+RBFnet+BAG	0.658	0.62	0.761	0.778
RUS1.5+LWL +BAG	0.677	0.652	0.751	0.755

TABLE V. RESULTS OF MOBAC 1.9.7

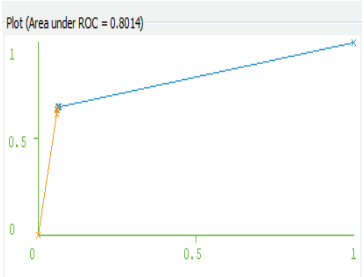
METHOD	METRICS		ALL SMELLS	
	AUC	*F-measure*	*AUC*	*F-measure*
NB	0.641	0.784	0.816	0.9
BN	0.727	0.835	0.817	0.9
SMO	0.5	0.843	0.636	0.89
DT	0.678	0.869	0.724	0.888
LR	0.756	0.872	0.817	0.895
RBFnet	0.622	0.843	0.824	0.892
LWL	0.717	0.845	0.813	0.879
SCART	0.659	0.869	0.78	0.896
SMOTE300+NB+BAG	0.682	0.526	0.801	0.844
SMOTE300+SMO+BAG	0.598	0.623	0.801	0.844
SMOTE300+RBFnet+BAG	0.727	0.662	0.8	0.842
SMOTE300+LWL+BAG	0.773	0.623	0.79	0.804
RUS1.5+NB+BAG	0.638	0.637	0.803	0.82
RUS1.5+BN+BAG	0.725	0.693	0.804	0.824
RUS1.5+SMO+BAG	0.657	0.58	0.816	0.803
RUS1.5+DT +BAG	0.768	0.699	0.784	0.795
RUS1.5+LR +BAG	0.76	0.715	0.797	0.82
RUS1.5+RBFnet+BAG	0.665	0.611	0.796	0.831
RUS1.5+LWL +BAG	0.714	0.704	0.784	0.76

TABLE VI. RESULTS OF MOBAC 1.9.16

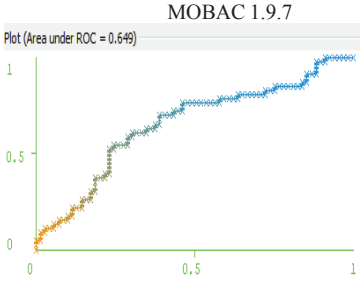
METHOD	METRICS		ALL SMELLS	
	AUC	*F-measure*	*AUC*	*F-measure*
NB	0.655	0.881	0.786	0.947
BN	0.587	0.942	0.786	0.947
SMO	0.5	0.942	0.509	0.942
DT	0.48	0.941	0.666	0.943
LR	0.691	0.948	0.777	0.945
RBFnet	0.603	0.942	0.805	0.945
LWL	0.639	0.942	0.756	0.942
SCART	0.474	0.942	0.474	0.942
SMOTE300+NB+BAG	0.706	0.726	0.798	0.861
SMOTE300+SMO+BAG	0.516	0.801	0.811	0.868
SMOTE300+RBFnet+BAG	0.755	0.797	0.8	0.866
SMOTE300+LWL+BAG	0.738	0.795	0.808	0.865
RUS1.5+NB+BAG	0.608	0.586	0.766	0.738
RUS1.5+BN+BAG	0.621	0.645	0.764	0.732
RUS1.5+SMO+BAG	0.596	0.549	0.779	0.75
RUS1.5+DT +BAG	0.629	0.593	0.764	0.763
RUS1.5+LR +BAG	0.649	0.645	0.726	0.724
RUS1.5+RBFnet+BAG	0.634	0.617	0.775	0.753
RUS1.5+LWL +BAG	0.599	0.61	0.771	0.763



a. ROC CURVE FOR LR WITH AUC 0.677 FOR MOBAC 1.9.1



b. ROC CURVE FOR SMOTE300+SMO+BAG WITH AUC 0.801 FOR MOBAC 1.9.7



c. ROC CURVE FOR RUS1.5+LR+BAG WITH AUC 0.649 FOR MOBAC 1.9.16

Fig. III. ROC curves for MOBAC (a, b, c)

likely to be affected by smells. Dependency of change-prone classes on smells makes it easier for software developers to save effort in maintenance phase. Fig. III shows ROC curves for MOBAC versions.

VII. CONCLUSION AND FUTURE WORK

The main aim of this paper was to study the relationship between object-oriented static software metrics, software code smells and change-prone classes. This is investigated by using class balance learning and Class imbalance learning methods. Machine learning algorithms are applied on Java based mobile application MOBAC. Metrics, smells and changes are identified. The following conclusions are described as under:

- Software Code Smells are better predictors of change-proneness as compared to object-oriented software metrics for all class balanced learning and class imbalanced leaning methods.
- Over sampling using SMOTE (300) predicts better as compared to class balance learning when metrics are used as indicators.
- LR classifier gives highest AUC and F-measure in class balance learning using metrics as indicators.
- SMO outperforms all the other machine learning algorithms when data is over sampled using SMOTE300 and smells are used as predictors.
- The above conclusions and results can be used by developers to identify change-prone classes and reduce maintenance effort. Smells can be used efficiently to determine the classes which are more prone to changes.

This research work can be further replicated to large software codes to further study the nature of the model.

REFERENCES

[1] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. Binkley, “Are test smells really harmful? an empirical study,” Emiprical Software Engineering, vol. 20, issue 4, pp. 1052-1094, 2015.

[2] M. Fowler, “Refactoring: improving the design of existing code,” Pearson Education India, 1999.

[3] F. Khomh, M. Di Penta, and Y. G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," IEEE 16th Working Conf. on Reverse Engineering (WCRE), pp. 75-84, 2009.

[4] A. Yamashita, “Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data, ” Empirical Software Engineering, vol. 19, issue 4, pp. 1111-1143, 2014.

[5] C. T. Chen, Y. C. Cheng, C. Y. Hseih, and I. L. Wu, "Exception handling refactorings: Directed by goals and driven by bug fixing," J. Systems and Software, vol. 82, issue 2, pp. 333-345, 2009.

[6] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," Empirical Software Engineering, vol. 21, issue 3, pp. 1143-1191, June 2016.

[7] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," IEEE Trans. on Software Engineering, vol. 38, issue 1, pp. 220-235, 2012.

[8] D. I. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," IEEE Trans. on Software Engineering, vol. 39, issue 8, pp. 1144-1156, 2013.

[9] A. Yamashita, and L. Moonen, "To what extent can maintenance problems be predicted by code smell detection?–An empirical study," Information and Software Technology, vol. 55, issue 12, pp. 2223-2242, 2013.

[10] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some Code Smells Have a Significant but Small Effect on Faults," ACM Trans. on Software Engineering and Methodology (TOSEM), vol 23, issue 4, p. 33, 2014.

[11] https://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html.

[12] S. Wang, and X. Yao, "Using class imbalance learning for software defect prediction," IEEE Trans. on Reliablity, vol. 62, issue 2, pp. 434-443, 2013.

[13] J. Ren, K. Qin, Y. Ma, and G. Luo, "On Software Defect Prediction Using Machine Learning," J. Applied Mathematics, 2014.

[14] Z. Sun, Q, Song, and X. Zhu, "Using Coding-Based Ensemble Learning to Improve Software Defect Prediction," IEEE Trans. on Systems, Man, and Cybernetics, Part C: Applications and Reviews, vol. 42, issue 6, pp. 1806-1817, 2012.

[15] S. Krishanan, "Evidence-based defect assessment and prediction for software product lines," PhD. Diss., Iowa State University, 2013.

[16] D. Rodriguez, I. Herraiz, R. Harrison, J. Dolado, and J. C. Riquelme, "Preliminary Comparison of Techniques for Dealing with Imbalance in Software Defect Prediction," Proc. 18th Int'l ACM Conf. on Evaluation and Assessment in Software Engineering, p. 43, 2014.

[17] Z. Mahmood, D. Bowes, P. C. R. Lane, and T. Hall, "What is the Impact of Imbalance on Software Defect Prediction Performance?," Proc. 11th Int'l ACM Conf. on Predictive Models and Data Analytics in Software Engineering, p. 4, 2015.

[18] P. Brennan, "A comprehensive survey of methods for overcoming the class imbalance problem in fraud detection," Institute of technology Blanchardstown Dublin, Ireland, 2012.

[19] R. Malhotra, and M. Khanna, "Investigation of relationship between object-oriented metrics and change proneness," Int'l J. Machine Learning and Cybernetics, vol. 4, issue 4, pp. 273-286, 2013.

[20] D. Spinellis, "Tool writing: A forgotten art?," IEEE, Software, vol. 22, issue 4, pp. 9–11, July/August 2005.

[21] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing Object-Oriented Class Modules Using an Agglomerative Clustering Technique," IEEE Int'l Conf. on Software Maintenance (ICSM), pp. 93-101, 2009.

[22] http://pl.csie.ntut.edu.tw/project/Robusta/Robusta_User_Instruction.pdf

[23] N. Tsantalis, M. Fokaefs, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of feature-envy bad smells," IEEE Int'l Conf. on Software Maintenance (ICSM), pp. 519–520, October 2007.

[24] J. V. Hulse, T. M. Khoshgoftaar, and A. Napolitano, "Experimental perspectives on learning from imbalanced data," Proc. 24th Int'l ACM Conf. on Machine learning, pp. 935-942, 2007.

[25] Y. Qian, Y. Liang, M. Li, G. Feng, and X. Shi, "A resampling ensemble algorithm for classification of imbalance problems," Neurocomputing, vol. 143, pp. 57-67, 2014.

[26] http://www.cs.waikato.ac.nz/ml/weka/