Indian Journal of Science and Technology, *Vol 10(24), DOI: 10.17485/ijst/2017/v10i24/115002, June 2017*
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Improving Smell Prediction: Developing an Improved Model with Supervised Learning Techniques

Neha Kumari and Satwinder Singh

Department of Computer Science and Technology, Central University of Punjab, City Campus, Manasa Road, Bathinda – 151001, Punjab, India; Neh.glb.cs.in@gmail.com, satwindercse@gmail.com

Abstract

Objectives: To build a model for the prediction of the code smells using the supervised learning techniques. The motive to build code smell prediction model is to propose a model with less false positive code smells. Also, the proposed model is cross validated using 10-fold cross validation **Methods/Statistical Analysis:** To build a smell prediction model, two code smell detection tools are used i.e. IPlasma and PMD. The metrics are extracted using understand and IPlasma. To achieve the above mentioned objective, two experiments are performed. One is using the code smells of the PMD and the other one is using the smells of the IPLasma. The smells of the PMD are associated with the metrics that are extracted using Understand. Then, the model is trained using the different supervised learning algorithms that are called classifiers i.e. Random Forest, Naïve Bayes and Kstar. **Findings:** In this research work, two experiments are performed. One is using the code smells of the PMD and other one is using the code smells of the IPlasma. From the results obtained i.e. with the code smells of the PMD, it is concluded that Random Forest predicts small number of false positive and false negative code smells as the precision and Recall of the Random Forest in each dataset is larger than the other two classifier's. Moreover, the ROC value of Random Forest is higher in some datasets and in some datasets the ROC value of KStar is higher. The results obtained i.e. with the code smells of the IPlasma, it is concluded that again Random Forest predict code smells more correctly than the other two classifiers and give less number of false positive and false negative code smells. Moreover, there is an exception, there is one dataset in which Random Forest and KStar both shows 100% accuracy i.e., Precison and Recall both are equal to 1, which shows that both classifiers predicts no false positive and false negative code smells. Moreover, the ROC value of Random Forest is higher than the other two classifiers, even in some datasets it is equals to 1. Using this it is concluded that Random Forest gives the best code smell predicting model. **Application/Improvements:** The results of this experiments shows only one case where the false positive and false negative code smells are not predicted by the models. This can improve such that on apply each and every dataset it gives zero false positive code smells.

Keywords: Automatic Tools, Bad Code Smells, Code Smell, Code Smell Prediction, Supervised Learning Techniques

1. Introduction

Code smells are indications of inadequate design, which are implemented by the programmer at the time of the development of the software. KENT Beck defines the term code smells as:

"Note that a code smell is a hint that something might be wrong, not a certainty. A perfectly good idiom may be considered a code smell because it's often misused, or because there's simpler alternative that works in most cases. Calling something a code smell is not an attack; it's simply a sign that a closer look is warranted".[1]

The structural characteristics of software are code smells that indicates that something is wrong, or the software may have some design problem and it can make difficult to maintain the software. Code smells are very helpful in refactoring the software; it will enhance the interior quality of the software. The quality and mainte-

*Author for correspondence

nance of the software are negatively affected by the bad code smells. Removal of these bad code smells is very crucial and to remove them, a process called refactoring is required. There are a number of techniques available for refactoring. The technique required to remove the code smell will differ for smell to smell. Refactoring is used to enhance the quality of the software and also to improve the maintenance of the software by removing the code smells from the system.[2]

There is no exact definition of code smells as these are subjective in nature. The different author explains different code smells, even gives different definitions of code smells. Back and Fowler gives an informal definition of a set of a 22 bad code smells and also provide the refactoring strategies to remove them.[1]

Whenever the developers finds any bad code smells, the very first thing that they should do is to evaluate whether the presence of code smell clues the relevant degradation in the code structure, and if they found the results positive, then they should decide the refactoring used to remove them. There is no need to remove all the code smells from the code, what type of code smells needs to be removed is completely depends on the system. If a particular code smell is found it needs to be removed, then it's better to remove it as early as possible. To remove them, it is required to detect the code smells first. For the detection of the code smells there is a support of automatic detection tools. These tools are very helpful as there are many smells which go unobserved by the programmers. However, the conceptions of bad smells is undefined and are subjectively interpreted. Different tools may provide different results when they analyze the same system as the various detection techniques used by the tools.[2]

1.1 Code Smell Detection

Nowadays there is raising a number of tools available for the analysis of the software that detects bad programming errors, identifying bad programming practices, highlighting unusual errors, and in general raise the knowingness of the software engineer about the structural characteristics of the program under development. Some of these tools detect bad code smells at the time of programming, and some of them are used to identify after the completion of the programming phase. As different tools use different detection rules, their results may also differ even if the tools are applied on the same source code.[3]

These tools are very useful in detecting the bad code smells and improving the quality and maintainability of the software project. But there is a problem with this software's. The problem is that they give many false negative results as well. Means when these bad code smells are manually validated by researchers they found that among the detected smells, there are many smells detected by the tools which are not bad smells in actual, i.e. false negative results. Because of this weak point, it is required to build a model that can give better results, or a model that can reduce the number of false negative bad code smells. In this research work, detected bad code smells of open source software source code are collected from the different tools and extract metrics of the same source code using open source tool. After that, the code smells are associated with the metrics. Then, this dataset is used to build a model for the detection/prediction of bad code smells with less false negative results using supervised learning techniques. In other words, a model with a better accuracy. The extracted metrics represent the independent variables, and the code smells dependent variables in the machine learning approach. After the experiment, comparison will be done to compare the performance of the classifiers that are used to train the models. At last, compare the results of the proposed model using cross validation techniques. For this purpose, two experiments are performed and in both experiments different tools are used for detection of bad code smells and different tools for the extraction of metrics. Even different code smells and metrics are used in both the experiments. The reason behind performing two tests is to analyze the results of the classifier. To check the performance of the classifier in each case in a manner that does the performance of the classifier affected with the change of the dataset. To answer these few questions, two experiments are performed.

The codes smells are subjective, different authors explain them differently. This section discusses an overview of the smells that are defined by various authors. Substantial work has been carried in the area of bad smells. The review of those work covered in this literature review.

In the reported article introduced the concept of code smells and produced an original catalog of twenty-two smells providing heuristics of qualitative nature for their detection.[4] After evaluation of Fowler's code smell, the author investigated the relationship between the bad code smells and class error probability in three severity levels as High (Blocker and Critical), Medium (Major), and

Low (Normal and Minor).[5] He also suggested that refactoring a class besides improving the architectural quality, reduces the probability of the class having errors after a system is released, because the probability of these classes having errors is very small. The author extended his previous work in which author investigated the relationship between the bad smells and class error probability.[5,6] In the examined three releases of Eclipse project to answer the question "whether the software metrics are still able to guess the class error-proneness of the post-release evolution of the system's?"[6] The author found out that there are still some software metrics which can still predict the class proneness in three error severity categories. In the reported article had worked on the effectiveness of encapsulation and object oriented metrics to re-factor code and identify error-prone classes using bad smells.[7]

Till now the focus of several authors are on the bad code smells, but in the coming paragraphs, the author concentrated on the detection tools of these bad code smells. The detection of code smells can either be manual or automated. The manual approach frequently involves subjective assessment, whereas the automated include software metrics, heuristics, and tools. In the proposed a procedure which is based on manual detection to detect the code smells for quality assessments.[8] The manual approach poses several limitations, such as lack of scalability and subjective bias. In the reported article that manually detected code smells depend much on the experience level of the subjects performing the detection (e.g. experienced developers identified complex smells more frequently than the less experienced).[9] Most detection approaches for code smells are automated. Earlier work suggested the use of metrics to identify situations where particular refactoring is needed but did not provide any formal specification of code smells. In the proposed article a language to detect code smells and other violations of code quality principles.[10] In the proposed article a detection and resolution sequence for duplication code., Long method, Large class, Long parameter list, Feature envy, Primitive obsession, Useless fields, method, and class to simplify their detection and resolution to support more efficient refactoring.[3] Despite the latest advances in automated code smell detection and several upcoming commercial tools, there are no conclusive answers to whether automated approaches provide better support for refactoring decision than manual procedures. The author in the Studies several tools for the automate detection of bad smells.[3] In the study the author found 84 tools; 29 of them are available online for download, and these tool.

Aim to detect 61 bad smells. The author also target different programming language, such as C, C++, and C#. The author also present a comparative study of four detection tools concerning two bad smells: Large Class and Long Method. The author also focuses on the detection tools and reviews the current panorama of the tools for automatic code smell detection.[11] Last two reviews focused on the detection tools, but the author develops a metrics model to identify the smelly classes and also validate that through the identification of the smelly and error prone classes.[12] The author also investigated two metrics (encapsulation and information hiding) for identifying smelly and faulty classes in software code. To develop this model the author examined the three version of the open source project i.e. Mozilla Firefox. The author studied several refactoring tools mainly based on Java on its usability and discussed the automation of different code smells.[13]

2. Experimental Approach

To predict a model for the prediction of smells, it is required to follow an approach. The very first thing is to select open source software system on which the experiment is performed. Next step is to extract the metrics for the selected software system. In this work, only object oriented metrics are extracted. Later on these metrics are considered as independent variables in the supervised learning approach which is used to train the model. After the extraction of metrics, the smells are being detected using the detection tools. In this research work PMD and IPlasma is used for the detection of smells. Next the extracted code smells are associated with the metrics. This association is done manually. Using this approach, one more label is added that shows whether the class is affected by smell or not in the dataset. This will be the final dataset on which the experiment is going to be performed and different machine learning algorithm is applied to trained the model which is used for the prediction of the smelly classes. After the training phase testing is performed using the same supervised learning algorithms and check for the accuracy of the model.

2.1 Data Collection

2.1.1 Selected Code Smells

For this research work, number of smells has been chosen that are detected using two automatic detection tools

i.e. PMD and IPlasma, IPlasma is a standalone software, whereas PMD is a Plug in of Eclipse. The smells that are detected by PMD are Data Clumps, Long Parameter List, Long Method, Large Class, and Dead code whereas IPlasma is used to detect God Class, Brain Class, Refused Parent Bequest and Data class code smells.

2.1.2 Selection of Open Source Software System

For this research work, different versions of open source system ArgoUML are considered. The reason for the selection is that this is an open source system, and is freely available online; also these source codes are compiled correctly and computed its metrics value. For this research work five version of Argouml are selected i.e., Argouml_0.24, Argouml_0.28, Argouml_0.30, Argouml_0.34

2.1.3 Metrics Extraction

For this research work, object-oriented metrics are computed that are considered as independent variables in our machine learning approach. The metrics are computed using two tools: Understand and IPLasma. These metrics are widely used in the literature and are well known as well. The chosen metrics are reported in Table 1, classified under six quality dimensions of object-oriented software.[14-17]

2.1.4 Smell Detection Tools

For this experiment, number of tools analyze many tools, some of them are PMD, Checkstyle, JDeodrant, Robust, etc. but there are only two tools that are finalized for this experiment, i.e., IPlasma and PMD. These tools are selected as both are simply available online, understandable, and the best thing is their results are easily decodable. PMD shows smells in the form of violations. These violations are then categorized into smells. Table 2 shows the mapping of violations with code smell.

2.2 Association and Labelling

In this phase, the association of detected code smells with the extracted metrics is to be done. Here two tools are used to obtain the code smells i.e. IPlasma and PMD. The smells obtained from PMD are associated with the metrics that are extracted from the tool Understand, whereas there is no need to manually associate the smells of IPlasma, as the metrics are used in this experiment are derived from IPlasma itself. After the procedure of association, labelling is to be performed. Labelling of bad code smells with the metrics means that we are going to label each class of the source code with an entry i.e. smelly or non-smelly which indicates whether that particular class is affected by any smell or not.

Finally, a dataset is obtained. This data set is then used for the training and testing purpose of the smell prediction model.

Using this procedure total ten datasets are obtained, two for each version of ArgoUML (one dataset is formed using PMD smells and with the metrics of the Understand, and the other one is formed using IPlasma).

The number of instances is different in various version of ArgoUML. The details of the instances and number of smelly instances from them of all the version is given in Table 3.

3. Experimental Setup

This work is experimented, by selecting three suitable supervised learning algorithms i.e. Naïve Baye KStar and Random Forest and testing them on the generated datasets by trained them using immediate previous version of ArgoUML.[18-21] After that, cross-validation is done to validate the results of the performed experiments using ten-fold cross-validation.

For each version, two datasets have been created. One is created using the smells of the PMD, and the other is

Table 1. List of Metrics

Tools	Size	Complexity	Cohesion	Coupling	Encapsulation	Inheritance
UNDERSTAND		WMC	LCOM	CBO RFC		DIT NOC NIM
IPLASMA	NOM NOA LOCC	AMW WOC WMC	TCC	ATFD CBO CC CM FANOUT	NOAM	DIT

being created using the smells of IPlasma.[22,23] The smells that are selected for this experiment are already mentioned in the above sections, also both the datasets have different metrics, and the list of metrics is also given in Table 1.

Table 2. Mapping of PMD Violations with code smells

Sl. No	Smell	Violations
1.	Data clumps	Too many fields
2.	Comments	Comment is too large
3.	Long parameter list	Parameter list is too long
4.	Duplicate code	Duplicate imports
		Consecutive strings
		Occurrence
5.	Large class	God class
		Too many methods
		Long class
6.	Long method	Long method
		NPath complexity
		Cyclomatic complexity
7.	Dead code	Empty catch block
		Empty if statement
		Empty block statement
		Document empty method body
		Document empty constructor

In each dataset, each row represents class instances and has one attribute for each metric. Besides, a binary value represents the label that shows whether the instance is affected by code smells or not.

The supervised learning algorithm that selected covers different machine learning approaches, i.e., Decision tree, the Bayesian network, K-Nearest neighbor. The implementation of these algorithms is available in weka.[24]

The aim of the cross-validation is to validate the results of the predicted models. For this, we selected four stand-

Table 3. Detail summary of smelly classes

Dataset	Argouml_Version	Total no. of classes	Smelly class	Percentage of smelly classes
PMD_SMELLS + METRICS OF UNDERSTAND	Argoum_0.24	1703	335	19.67%
	Argouml_0.28	2396	316	13.19%
	Argouml_0.30	2707	312	11.53%
	Argouml_0.30.1	2738	312	11.39%
	Argouml_0.34	2412	474	19.65%
IPLASMA (BOTH METRICS AND SMELLS)	Argouml_0.24	1771	92	5.19%
	Argouml_0.28	2377	162	6.82%
	Argouml_0.30	2687	167	6.22%
	Argouml_0.30.1	2708	168	6.20%
	Argouml_0.34	2355	168	7.13%

Table 4. Extracted smells summary using PMD

Dataset	PMD SMELLS					
	Dataclumps	Long parameter_ list	Duplicate code	Large class	Long method	Deadcode
Argouml_0.24	20	1	22	257	203	113
Argouml_0.28	15	0	16	222	178	141
Argouml_0.30	14	0	16	219	177	144
Argouml_0.30.1	14	0	16	219	177	144
Argouml_0.34	21	3	91	317	240	212

alone performance measures: F-Measures, ROC, Recall and Precision. These four criteria explain the different point of views of the performance of the predictive models.

3. Experimental Results

In this section, the data collected data is described, the experiments performed on them and their results as well.

Table 4 shows the detailed summary of the smells that are extracted using PMD. At the time of smell extraction and association of PMD code smells with the metrics, a difference between the smells that are detected by PMD and the code smells after the association of smells is found. This means that there are some classes which is smelly according to the PMD but are not present in` the metrics dataset that are extracted by Understand. But these differences is found in few classes only and are ignorable. Table 5 explains the detailed summary of the smells which are extracted by IPLasma. Table 5 shows the total number of instances/classes, a number of smells found of the particular type and the last columns shows the total number of classes which are affected by the smells.

Table 5. Extracted smells detailed summary using IPLasma

Dataset	Total number of classes	Brain class	Data class	God class	Refused parent bequest	Total smelly classes
Argouml_0.24	1771	10	17	62	6	92
Argouml_0.28	2377	21	71	61	14	162
Argouml_0.30	2687	21	71	68	13	167
Argouml_0.30.1	2708	21	71	69	13	168
Argouml_0.34	2355	20	74	69	10	168

Table 6. PMD evaluated instances summary details

Dataset	Classifier	Total instances evaluated	Correctly evaluated instances (%)	Incorrectly evaluated instances (%)
Dataset training dataset-Argouml_0.24 && testing dataset-Argoum_0.24	Naïve Bayes	1703	86.7293	13.2707
	KStar		98.121	1.879
	Random Forest		98.7082	1.2918
Training dataset-Argouml_0.24 && testing dataset-Argoum_0.28	Naïve Bayes	2396	85.5593	14.4407
	KStar		89.1486	10.8514
	Random Forest		89.2738	10.7262
Training dataset-Argouml_0.28 && testing dataset-Argoum_0.30	Naïve bayes	2707	88.5482	11.4518
	KStar		97.0816	2.9184
	Random Forest		97.3033	2.6967
Training dataset-Argouml_0.30 && testing dataset-Argoum_0.30.1	Naïve Bayes	2738	88.6413	11.3587
	KStar		98.466	1.534
	Random Forest		98.8678	1.1322
Training dataset-Argouml_0.30.1 && testing dataset-Argoum_0.34	Naïve Bayes	2412	84.8611	15.1389
	KStar		90.5848	9.4152
	Random Forest		90.6263	9.3737

Table 7. IPLasma evaluated instances summary details

Dataset	Classifier	Total instances evaluated	Correctly evaluated instances (%)	Incorrectly evaluated instances (%)
training dataset-Argouml_0.24 && testing dataset-Argoum_0.24	Naïve Bayes	1771	95.8216	4.1784
	KStar		100	0
	Random Forest		100	0
Training dataset-Argouml_0.24 && testing dataset-Argoum_0.28	Naïve Bayes	2377	94.1944	5.8056
	KStar		96.0034	3.9966
	Random Forest		97.2655	0.7345
Training dataset-Argouml_0.28 && testing dataset-Argoum_0.30	Naïve bayes	2687	93.8221	6.1779
	KStar		99.2929	0.7071
	Random Forest		99.5162	0.4838
Training dataset-Argouml_0.30 && testing dataset-Argoum_0.30.1	Naïve Bayes	2708	93.72	6.28
	KStar		99.8892	0.1108
	Random Forest		99.9631	0.0369
Training dataset-Argouml_0.30.1 && testing dataset-Argoum_0.34	Naïve Bayes	2355	92.7389	7.2611
	KStar		99.3631	0.6369
	Random Forest		99.7028	0.2972

Table 8. Classifier results (PMD)

Dataset	Classifier	Precision	Recall	F-Measure	Roc	Best/Classifier	Worst Classifier
Training Dataset-Argouml_0.24 && Testing Dataset-Argoum_0.24	Naïve Bayes	0.858	0.867	0.858	0.835	Random Forest	Naïve Bayes
	Kstar	n0.982	0.981	0.981	0.998		
	Random Forest	0.987	0.987	0.987	0.999		
Training Dataset-Argouml_0.24 && Testing Dataset-Argoum_0.28	Naïve Bayes	0.857	0.856	0.856	0.798	Kstar	Naïve Bayes
	Kstar	0.904	0.891	0.897	0.912		
	Random Forest	0.907	0.893	0.898	0.909		
Training Dataset-Argouml_0.28 && Testing Dataset-Argoum_0.30	Naïve Bayes	0.867	0.885	0.873	0.811	Kstar	Naïve Bayes
	Kstar	0.97	0.971	0.97	0.988		
	Random Forest	0.974	0.975	0.974	0.986		
Training Dataset-Argouml_0.30 && Testing Dataset-Argoum_0.30.1	Naïve Bayes	0.872	0.886	0.877	0.817	Kstar And Random Forest	Naïve Bayes
	Kstar	0.985	0.985	0.984	0.998		
	Random Forest	0.988	0.988	0.988	0.998		
Training Dataset-Argouml_0.30.1 && Testing Dataset-Argoum_0.34	Naïve Bayes	0.85	0.849	0.824	0.842	Random Forest	Kstar
	Kstar	0.865	0.906	0.895	0.788		
	Random Forest	0.873	0.907	0.897	0.889		

Table 9. Classifier results (IPLasma)

Dataset	Classifier	Precision	Recall	F-Measure	ROC	Best/Classifier	Worst Classifier
Training Dataset-Argouml_0.24 && Testing Dataset-Argoum_0.24	Naïve Bayes	0.963	0.958	0.96	0.936	Kstar And Random Forest	Naïve Bayes
	Kstar	1	1	1	1		
	Random Forest	1	1	1	1		
Training Dataset-Argouml_0.24 && Testing Dataset-Argoum_0.28	Naïve Bayes	0.938	0.942	0.94	0.788	Random Forest	Naïve Bayes
	Kstar	0.957	0.96	0.955	0.945		
	Random Forest	0.972	0.973	0.97	0.95		
Training Dataset-Argouml_0.28 && Testing Dataset-Argoum_0.30	Naïve Bayes	0.939	0.938	0.938	0.868	Random Forest	Naïve Bayes
	Kstar	0.993	0.993	0.993	0.993		
	Random Forest	0.995	0.995	0.955	1		
Training Dataset-Argouml_0.30 && Testing Dataset-Argoum_0.30.1	Naïve Bayes	0.938	0.937	0.938	0.857	Random Forest	Naïve Bayes
	Kstar	0.999	0.999	0.999	0.999		
	Random Forest	1	1	1	1		
Training Dataset-Argouml_0.30.1 && Testing Dataset-Argoum_0.34	Naïve Bayes	0.929	0.927	0.928	0.842	Random Forest	Naïve Bayes
	Kstar	0.994	0.994	0.994	0.991		
	Random Forest	0.997	0.997	0.997	0.996		

Table 10. Cross-Validation Results (PMD)

Testing Dataset	Classifier	Precision	Recall	F-Measure	Roc	Best/Classifier	Worst Classifier
Argouml_0.24	Naïve Bayes	0.851	0.861	0.852	0.831	Random Forest	Naïve Bayes
	Kstar	0.853	0.861	0.855	0.888		
	Random Forest	0.875	0.88	0.876	0.907		
Argoum_0.28	Naïve Bayes	0.86	0.879	0.864	0.806	Random Forest	Naïve Bayes
	Kstar	0.878	0.89	0.882	0.881		
	Random Forest	0.881	0.891	0.884	0.889		
Argoum_0.30	Naïve Bayes	0.868	0.883	0.874	0.813	Random Forest	Naïve Bayes
	Kstar	0.887	0.898	0.89	0.888		
	Random Forest	0.894	0.905	0.897	0.891		
Argoum_0.30.1	Naïve Bayes	0.87	0.884	0.875	0.808	Random Forest	Naïve Bayes
	Kstar	0.883	0.896	0.887	0.886		
	Random Forest	0.887	0.9	0.89	0.888		
Argoum_0.34	Naïve Bayes	0.85	0.861	0.848	0.836	Random Forest	Naïve Bayes
	Kstar	0.865	0.872	0.867	0.892		
	Random Forest	0.873	0.879	0.874	0.911		

Table 6 shows the detailed summary of the instances that are evaluated by the classifiers. In Table 6, the first column shows the detail of dataset, here the dataset which is created by using PMD code smells. The next column shows the classifier. Each dataset is evaluated using three classifiers (Naïve Bayes, KStar and Random Forest). In the next column, total number of instances that are evaluated. The last two columns show the details of correctly classified instances and incorrectly classified instances in percentage Table 6 shows that Random Forest classifier evaluates instances more correctly than the other two classifiers.

Table 7 shows the detailed summary of instances of the dataset which is created by using the code smells of IPLasma. This dataset is also evaluated using the same classifiers. Table 7 shows that Random Forest more correctly classifies the instances than the other two. Even in the first dataset, it classifies all the instances correctly i.e. 100%. But there are exception in one case, that is in Dataset 4. In dataset 4 KStar classifies the instances more correctly than Random Forest.

Table 8 shows the results of the classifier. There are Four performance parameters are used to measure the performance of the proposed model i.e. Precision, Recall, F-measure, and ROC. Table 8 shows the results of the classifiers which are applied on the PMD dataset. The Table 8 shows that the best performance is given by Random Forest in some datasets and in some datasets it is given by KStar whereas Naïve Bayes gives the worst results. It is also noticed that in dataset 1 the ROC value of Random classifier is 0.999, which shows that its accuracy is outstanding.

Table 9 shows the results of the classifier which are applied on the IPlasma dataset. Table 9 shows that the best performance is obtained by using the classifier Random Forest. Even in some cases, it shows the value of ROC

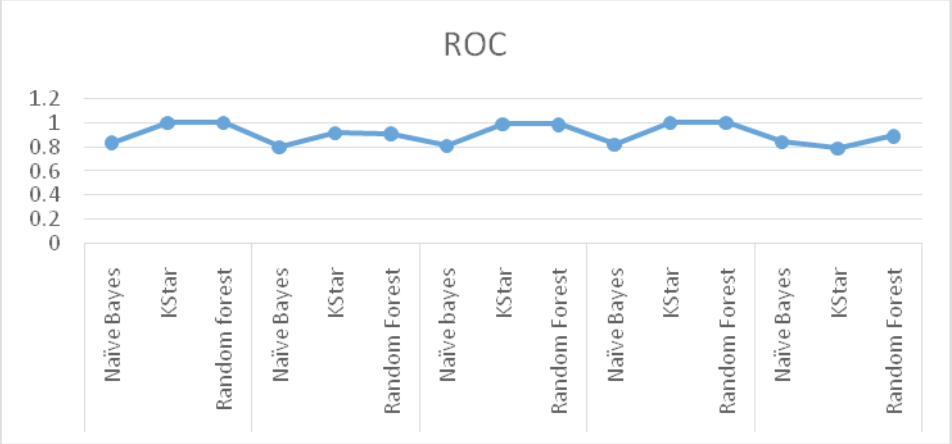

Figure 1. ROC Curve for the PMD datasets.

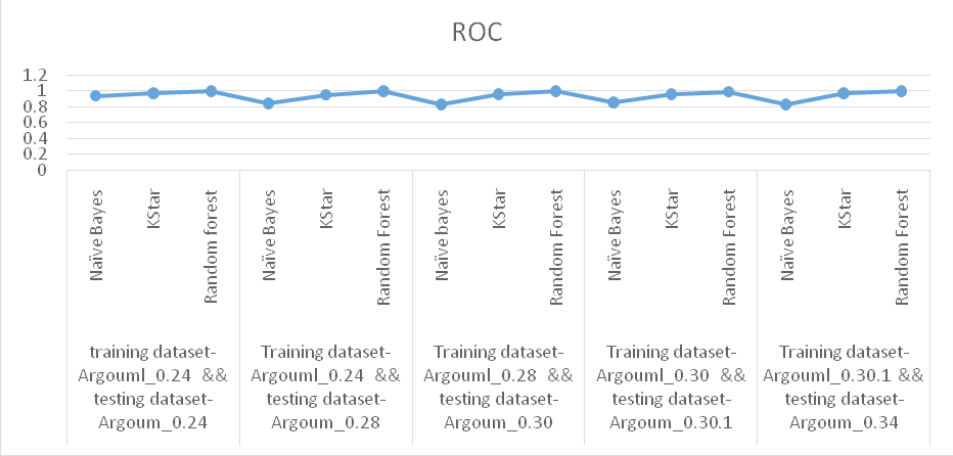

Figure 2. ROC Curve for the IPlasma datasets.

Table 11. Cross-Validation Results (IPlasma)

Dataset	Classifier	Precision	Recall	F-Measure	Roc	Best/Classifier	Worst Classifier
Training Dataset-Argouml_0.24 && Testing Dataset-Argoum_0.24	Naïve Bayes	0.962	0.957	0.959	0.937	Random Forest	Naïve Bayes
	Kstar	0.969	0.984	0.97	0.964		
	Random Forest	0.987	0.987	0.986	0.989		
Training Dataset-Argouml_0.24 && Testing Dataset-Argoum_0.28	Naïve Bayes	0.935	0.935	0.935	0.839	Random Forest	Naïve Bayes
	Kstar	0.955	0.959	0.955	0.95		
	Random Forest	0.983	0.983	0.983	0.993		
Training Dataset-Argouml_0.28 && Testing Dataset-Argoum_0.30	Naïve Bayes	0.938	0.936	0.937	0.829	Random Forest	Naïve Bayes
	Kstar	0.959	0.962	0.962	0.957		
	Random Forest	0.986	0.986	0.984	0.994		
Training Dataset-Argouml_0.30 && Testing Dataset-Argoum_0.30.1	Naïve Bayes	0.938	0.936	0.937	0.853	Random Forest	Naïve Bayes
	Kstar	0.962	0.965	0.962	0.957		
	Random Forest	0.984	0.984	0.984	0.987		
Training Dataset-Argouml_0.30.1 && Testing Dataset-Argoum_0.34	Naïve Bayes	0.929	0.929	0.929	0.832	Random Forest	Naïve Bayes
	Kstar	0.96	0.963	0.959	0.964		
	Random Forest	0.984	0.985	0.984	0.995		

nearly 1, which means the model, predicts the code smells 100% accurately. There is one exception, where KStar also shows 100% accuracy. As in the case of PMD dataset, the worst performance in this case is also given by classifier naïve Bayes.

Figure 1-2 shows the line graph for the ROC values of each classifier for each dataset. ROC is a plot of true positive rate against false positive rate as the distinction.

Threshold of the classifier is varied. The area under ROC gets close to value 1 when the discrimination performs better, while a bad classification brings value close to 0.5.

Table 10 and 11 shows the cross-validation results of the experiments. The cross validation was done using the ten cross fold. In 10 fold cross validation, in which the data breaks into ten sets of equal size and then trained the model using nine datasets and perform testing on 1. This process is repeat for ten times, and the mean of the performance measures is taken. Table 10 shows the cross-validation results for the PMD datasets. This will show that the highest performance is achieved using the classifier Random Forest. There is a very small difference in the results of the classifier and cross- validation. There are two classifier (KStar and Random Forest) in the above results that shows the highest performance but on cross-validating it is found that only one of them is showing the best performance i.e. Random Forest, whereas the worst performance is given by naïve Bayes in both the cases. Table 11 shows the cross-validation results for the IPLasma dataset. Table 11 shows that the best performance is achieved by again by the classifier Random Forest. The proposed model also shows that the Random Forest is the good predicting classifier. The worst performance is obtained by naïve Bayes.

5. Conclusion

In this paper, study of different tools that are used for the detection of code smells has been done. From the study, it is concluded that there are two tools i.e. PMD and IPlasma which are easily available online and the code smells detected by them are easily interpretable. Although PMD is not as simple as IPlasma, but a small study of the violations make it very easy to interpret the smells of the

PMD. These tools are further used in this research work for the prediction of the code smells.

In this work, models are predicted that can be used for the prediction of the code smells. This work includes two experiments, using two different smell detection tools i.e. PMD and IPlasma. Both tools are used to detect different code smells. Three classifiers (Naïve Bayes, KStar and Random Forest) are used for predicting the code smell prediction models. From the results obtained it is concluded that Random Forest is the best code smell predicting model in both the experiments.

Moreover, Naïve Bayes model for the code smell prediction, also predicted code smells well but Random Forest and KStar gives more useful output. Naïve Bayes models predicted IPlasma code smells more accurately than the code smells of the PMD.

Table 10 and Table 11 shows the results of cross-validation for both PMD code smells and IPlasma code smells. The results show that Random Forest models is the best code smell prediction models. Moreover,it is also noticed that for the prediction of code smells of IPlasma, Random Forest models are the best predicting model.

6. References

1. Fowler M, Back K, Brant J, Opdyke W, Roberte D. Refactoring: improving the design of existing code; 2002. p. 1–337.
2. Fontana FA, Braione P. Automatic detection of bad smells in code: An Experimental Assessment. Journal of Object Technology; 2012. p. 1–38.
3. Fernandes E, Oliveira J, Vale G, Figueiredo E. A Review-based Comparative study of bad smell detection tools. ACM-I; 2010. p. 1–12.
4. Beck K. Refactoring: improving the design of existing code. Address- Wasley professional; 1999.
5. Li W, Shatnawi R. An emprical study of the bad smells and class error probability in the post-release object-oriented system evolution. Science Direct; 2007. p. 1120–8.
6. Shatnami R, Li W. An effectiveness of software metrics in identifying error-prone classes in post-releases software evolution process. Science Direct; 2008. p. 1868–82.
7. Deligiannis I, Shepperd M, Roumeliotis M, Stamelos I. An emprical investigation of an object-oriented design heuristic for maintainability. The Journal of Systems and Software; 2003. p. 127–39. Crossref
8. Travassos G. Detecting defects in object-oriented designs in Object-oriented programming systems languages and applications; 1999.
9. Yamashita A, Moonen L. Do code smells reflect important maintainability aspects? IEEE; 2012. p. 306–15. Crossref
10. Alikacem EH, Sahraoui H. A metric extraction framework based on a high-level description language. Conference on Source Code Analysis and Manipulation; 2009. p. 159–67.
11. Fontana FA, Mariani E, Morniroli A, Sormani R, Tonello A. An experience report on using code smells detection tools. IEEE FOURTH International Conference on Software Testing Verification and Validation; 2011. p. 450–7.
12. Singh S, Kahlon KS. Effectiveness of refactoring metrics model to identify smelly and error prone classes in open source software. ACM SIGSOFT Engineering Notes; 2012. p. 1–11.
13. Mahmood J, Reddy Y. Automatic refactorings in java using intelliJ IDEA to extract and propagate constants. IEEE International Conference IACC; 2014.
14. Bansiya J, Davis CG. A Hierarchical Model for Object-Oriented design quality Assessment. IEEE; 2002. p. 4–17.
15. Dubey SK, Sharma A, Rana DA. Comparison Study and Review on Object-Oriented Metrics. Global Journal of Computer Science and Technology; 2012. p. 1–11. PMid:22078010
16. Chidamber SR, Kermerer CF. A metrics suite for object-oriented design. IEEE; 1994. p. 476–93.
17. Lanza M, Marinescu R. Object-Oriented Metrics in Practices. Springer; 2006.
18. Karthika S, Sairam N. A Naive Bayesian Classification for Educational qualification. Indian Journal of Science and Technology. 2015; 8(16):1–5. Crossref
19. Ren J, Lee SD, Chen X, Kao B, Cheng R, Cheung D. Naive Bayes Classification of Uncertain Data; 2009. p. 944–9.
20. Vijayarani S, Muthulakshi M. Comparative analysis of Bayes and Lazy classification algorithms. International Journal of Advanced Research in Computer and Communication Engineering. 2013; 2(8):3118–24.
21. Biau G. Anaysis of a Random Forest Model. Journal of Machine Learning Research; 2012. p. 1063–95.
22. Source Forge Maven [Internet]. 2014 [cited 2014 May 14]. Available from: http://pmd.sourceforge.net/snapshot/
23. Marinescu C, Marinescu R, Mihancea P, Wettle R, Ratiu D. IPlasma: An integral platform for quality Assessment of object oriented design. 21st IEEE International Conference on Software Maintenance (ICSM); 2005. p. 1–4.
24. Hall M, Frank E. The Weka Data Mining Software: An Update. SIGKDD. 2009; 11(1):10–8. Crossref