International Journal of Rough Sets and Data Analysis
Volume 6 • Issue 2 • April-June 2019

Detection of Shotgun Surgery and Message Chain Code Smells using Machine Learning Techniques

Thirupathi Guggulothu, University of Hyderabad, Hyderabad, India
Salman Abdul Moiz, University of Hyderabad, Hyderabad, India

ABSTRACT

Code smell is an inherent property of software that results in design problems which makes the software hard to extend, understand, and maintain. In the literature, several tools are used to detect code smell that are informally defined or subjective in nature due to varying results of the code smell. To resolve this, machine leaning (ML) techniques are proposed and learn to distinguish the characteristics of smelly and non-smelly code elements (classes or methods). However, the dataset constructed by the ML techniques are based on the tools and manually validated code smell samples. In this article, instead of using tools and manual validation, the authors considered detection rules for identifying the smell then applied unsupervised learning for validation to construct two smell datasets. Then, applied classification algorithms are used on the datasets to detect the code smells. The researchers found that all algorithms have achieved high performance in terms of accuracy, F-measure and area under ROC, yet the tree-based classifiers are performing better than other classifiers.

KEYWORDS

Code Smells, Machine Learning, Software Refactoring, Stratified Sampling, Supervised Learning

INTRODUCTION

Code smells or bad code smells refers to an anomaly in the source code that may result in deeper problems which makes software difficult to understand, evolve, and maintain. According to (Booch, 2006) smell is a kind of structure in the code that shows a violation of basic design principles such as Abstraction, Hierarchy, Encapsulation, Modularity, and Modifiability. Even if the design principles are known to the developers due to inexperience, the competition that is in the market and deadline pressure are leading to violation of these principles. Fowler et al. (Fowler, 1999) have defined 22 informal code smells which are removed through refactoring techniques. These techniques are used to enhance the internal structure of the code without varying the external behaviour and to improve the quality of the software. The (Opdyke, 1992) authors have defined 72 refactoring techniques.

There are various methods and tools available in the literature to detect the code smells. Each technique and tool produces different (Fontana, 2012). Bowes et al. (Bowes, 2013), compared two code smell detection tools on message chaining and shown disparity of results between them. The three main reasons for varying results are: 1) The code smells can be subjectively interpreted by the developers, and hence detected in different ways. 2) Agreement between the detectors is low, i.e., different tools or rules detect a different type of smell for different code elements. 3) The threshold value for identifying the smell can vary for the detectors.

DOI: 10.4018/IJRSDA.2019040103

To address the above limitations, in particular the subjective nature, Fontana et al. (Fontana, 2016) proposed a machine learning (ML) technique to detect four code smells (Long Method, Data Class, Feature Envy, Large Class) with the help of 32 classification techniques. The authors have built 4 datasets, one for each smell. These datasets have been prepared based on the tools and manual labelling process. Tools are used to identify whether the code elements (instances) are smelly or not. But the tools may produce some false positive instances so, the authors manually validated the instances to avoid the biasness. In this paper, instead of using tools and manual validation, the authors have prepared two new method level code smell datasets of Fowler et al. (Fowler, 1999) from the literature; based on the detection rules and unsupervised learning i.e., clustering to validate the instances as smelly or not.

In the proposed work, an attempt is made to detect two code smells namely Shotgun surgery and Message chaining with supervised learning techniques. It is an application of machine learning (ML) classification approach used for code smell detection. It uses known data to determine how the new instances should be classified into binary classification i.e., based on the metrics used for a particular method, the ML approach helps in classifying a method to be prone to code smell or not. In this paper, the dataset instances are methods of 74 heterogeneous java systems. The metrics of object-oriented systems have been computed on method instances, which are the features or attributes of the dataset. For each smell, one dataset is prepared by using detection rules from the literature (Ferme, 2013). The researchers applied a random stratified sampling on the method instances to balance the datasets. Sample instances of the dataset are validated through unsupervised learning and added to the training dataset. Then applied some known classification algorithms on the trained datasets to detect the code smells, by using 10-fold cross validation method. To evaluate those algorithms, standard metric measures such as F-score, accuracy and the area under the ROC are used. The experimented algorithms have achieved high performance in both the smells.

The paper is been arranged as follows; The second section, introduces a work related to detection of code smells; The third section, defines two proposed approaches of code smell detections; The fourth section, detecting code smells using ML approach; The fifth section, presents experimental results; The sixth section, presents the code smell detection rules; and the final section, gives conclusion and future directions.

RELATED WORK

According to (Kessentini, 2014) approaches of code smell detection are classified into 7 categories (i.e., cooperative-based approaches, visualization-based approaches, search-based approaches, probabilistic approaches, metric-based approaches, symptoms based approaches, and manual approaches). In the manual approach developers and maintainers follow different reading guidelines to detect smells. As it requires human involvement, it consumes more time for large systems. In the metric-based approach, smell detection is based on source code metrics. Symptoms based approach uses different notations to detect smells. But the problem with this approach is, it requires analysis to convert symptoms or notations into detection algorithms. Probabilistic approach is based on applying fuzzy logic rules to detect smells. The visualization approach uses semi-automated processes to detect and visualize the smells with the integration of human capabilities. But the problem with this approach is that, it requires human effort, with increase in large systems. The search-based approach applies different algorithms to detect the smells. Most of the techniques use ML approaches. The success of this approach depends upon the training datasets. The cooperative approach performs different activities in a cooperative way.

Fontana et al. (2015) proposed a detection strategy for the code smells. The authors have derived metric thresholds to detect code smells, from a benchmark of 74 java software systems.

Fontana et al. (2016) experimented and compared the supervised ML algorithms to detect the code smells. The authors have used 74 java systems to prepare the training dataset which are manually

validated instances. Then used 16 different classification algorithms and in addition to it, boosting techniques are applied on 4 code smells viz., Long Method, Data Class, Feature Envy, and Large Class.

In this proposed approach an attempt is made to detect two additional code smells called shotgun surgery and message chaining through ML classification techniques. In Fontana et al. (Fontana F. A., 2016) advisors (Tools) are used to identify whether the class is smelly or not, but the authors have considered the tools to be subjective to errors and not biased. So, they went for manual validation on the instances. But in the proposed approach, the researchers have used the code smell deterministic rules from literature (Fontana F. A., 2015) to identify whether the class is smelly or not and instead of manual validation unsupervised learning is used. In the proposed work, the training dataset size is larger than the previous work. A large dataset would generalize the instances effectively. In both proposed and previous work, the tree based classifiers are giving better performance than other classifiers.

CODE SMELL BASICS

In this work, the researchers have considered two code smells among 22 code smells identified by Fowler et al. (Fowler, 1999), to experiment on the smell detection approach. The reason for choosing these two code smells is to cover potential problems related to object-oriented quality dimension called coupling. Coupling is the relational strength between entities of the systems. High coupling may negatively impact the software quality dimension. In Table 1, the researchers have outlined the selected code smells and reported the smell definitions. There are usually two levels of affected entities in the code smell i.e., class level and method level and each code smell affect either an intra or inter class. Intra class means code smell affecting a single entity in the source code and inter class means code smell affecting more than one entity in the source code. In this paper, work is carried out on method level smells.

Shotgun surgery says, to introduce a small new change, a developer has to change many classes and methods, and most of the time writes duplicated code, which violates the "Don't Repeat Yourself" principle.

The message chaining, code smell refer to a particular class or method which has high coupling with other classes or methods in chain-like delegations, i.e., methods that contain long sequences of method calls to get data from other classes.

CODE SMELL DETECTION USING MACHINE LEARNING APPROACH

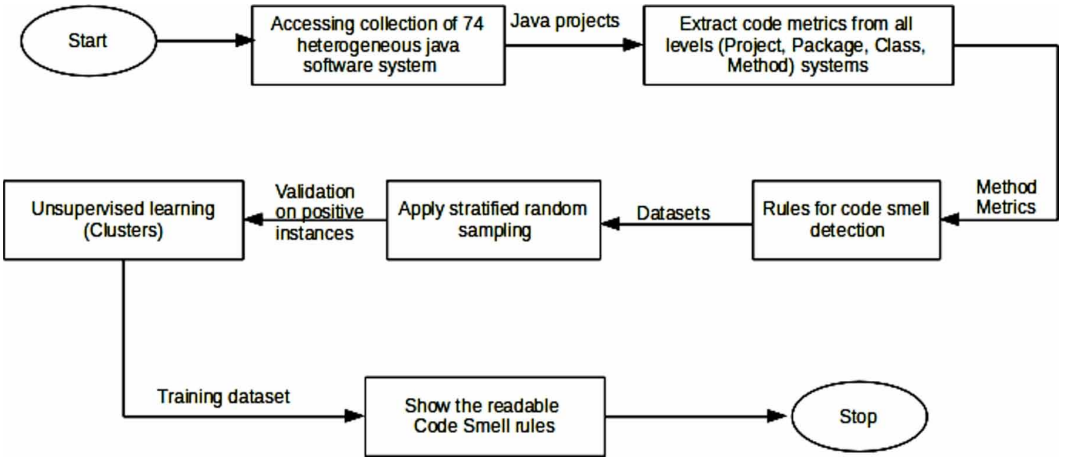
The application of ML classification approach is to detect the code smell using known data to determine how the new data must be classified into a binary classification (code is smelly or not), Figure 1 describes the flow of activities in the proposed approach to detect code smells.

The summary of the flow chart will be described here. Following sections will give a detailed explanation of one each activities of the researchers approach during the code smell detection.

Table 1. Selected fowler code smell

Name of Code Smell	Affected Entities	Intra / Inter	Impacted on Object Oriented Quality Dimensions
Shotgun Surgery	Method	Inter class	Coupling
Message chaining	Method	Inter class	Coupling

Figure 1. Flow of activities to detect the code smells



- A collection of 74 heterogeneous java systems are been collected and considered as input instances (methods) for creation of the dataset.
- From the given 74 systems, metrics extraction was done from all the levels such as Project, Package, Class and Method. These metrics become features to the dataset.
- For the binary classification of code smells, class variables are considered. To assign class variable (smelly or not) the researchers chosen code smell rules from the literature.
- The above steps result in a dataset which is imbalanced. The researchers applied a random stratified sampling on the method instances to balance the datasets.
- Sample instances of the dataset are validated through unsupervised learning and added to the training dataset.
- Known supervised classification algorithms are applied on the training dataset.
- Among the supervised classification algorithms, J48 and JRIP produces human readable code smell rules.

Collection of Java Software Systems

In order to prepare code smell classification dataset, java software systems are collected from (Fontana F. A., 2016). The author has provided a collection of 74 java systems with the compiled version, collected from (Tempero, Anslow, Dietrich et al., 2010). The 74 systems are having different sizes and various application domains. Table 2 reports, the characteristics of all 74 projects. The data selected are large enough to experiment on the ML algorithms. The large dataset would lead to more generalized ML algorithm results. The number of instances (methods) create the dataset.

Extracting All Code Level Metrics

The metrics of source code are used to identify the problems and even used to improve the quality of the software system. The various types of metrics used to measure source code properties are coupling, encapsulation, cohesion, complexity, size and inheritance. Software quality dimensions cover different aspects of the source code. Usually, metrics are categorized into three: Process, Resource, and Product.

1. **Process Metrics:** These are the metrics used to measure the effectiveness and efficiency of various process. Process metrics are related to function points, percentage of defective detection, defective density etc.

Table 2. Summary of 74 projects

Number of Projects	Number of Lines in All Projects	Number of Packages in All projects	Number of Classes in All Projects	Number of Methods in All Projects
74	6,785,568	3420	56,225	4,15,995

2. **Resource Metrics:** These are the metrics used to measure the quantity of cost, defects, productivity, schedule and estimation of various project deliverables and resources. Resource metrics are also related to schedule, cost, productivity and number of developers.
3. **Product Metrics:** These are the metrics used to measure the internal structure of software. Product metrics are related to software quality dimensions like coupling, encapsulation, cohesion, complexity, size and inheritance.

In this paper, the researchers particularly focused on product metrics because, software refactoring changes the internal structure of the software. As mentioned in the above definition, product metrics measures the internal structure of software. The six object oriented software quality dimensions are related to code-smell characteristics. In appendix section, Figure 7 listed the metrics which are categorized into quality dimensions and with their abbreviations. Each dimension is related to few metrics list which are mentioned below, that are independent variable of the dataset:

- **Size:** The size of the system depends upon number of packages, number of classes, number of methods, and number of lines of code and so on. Larger the system, the more difficult it is to manage. Size related metrics are LOC, LOCNAMM, NOM, NOPK, NOCS, NOA, and NOMNAMM.
- **Complexity:** It is measured based on the level of difficulty in understanding the structure of the class (Bansiya, Jagdish and Davis, Carl G., 2002). As the complexity of the class increases, it would be hard to understand it. Complexity related metrics are CYCLO, WMC, WMCNAMM, AMW, MAXNESTING, WOC, CLNAMM, NOP, NOAV, ATLD, NOLV, and AMWNAMM.
- **Cohesion:** It is used to measure the strength of relatedness among methods and attributes in a class (Balmas, Francoise and Bergel, Alexandre and Denier, Simon and Ducasse, Stephane and Laval, Jannik and Mordal-Manet, Karine and Abdeen, Hani and Bellingard, Fabrice, 2010). Cohesion metrics are LCOM5, TCC.
- **Coupling:** It is used to measure the strength of dependence among the objects in a design. Therefore, the stronger the coupling between the objects, the more difficult to change, understand, and correct. Coupling related metrics are FANOUT, ATFD, FDP, RFC, CBO, CFNAMM, CINT, CDISP, CC, and CM.
- **Encapsulation:** It is defined as binding of data and behavior within a single block called class. Without encapsulation in classes an unauthorized person can able to access it directly. The metrics related to encapsulation are LAA, NOAM, and NOPA.
- **Inheritance:** Is-a relationship between classes is measured by inheritance. That means acquiring the properties of one class to another class. The level of nested classes depends upon the relationship related to the inheritance hierarchy. As the complexity of the hierarchy increases understanding it become difficult, because of the inherited methods and attributes from the ancestor classes. Inheritance related metrics are DIT, NOI, NOC, NMO, NIM, and NOII.

The object oriented metrics (product) are grouped into four categories called class, method, package and project (LAB.(n.d)). Their corresponding few metrics are listed below, in the Table 3. These metric levels follow the containment relation, i.e., class is present in a package, method is

present in a class etc. There are usually two levels of affected entities in the code smell i.e., class level and method level. In this paper, researchers are working on method level smells. The method level smells, not only includes method level metrics, but also includes class, package and project level metrics in the dataset as an independent variable.

Rules for Smell Detection

The supervised classification algorithm needs a training dataset which consists of instances (methods or classes), features (all level metrics) and class labels (code smells). In the dataset preparation, the class label should specify whether a method or a class instance is affected by the code smell or not. The data of 74 systems are large enough and heterogeneous. Hence, it is difficult to assign class labels for such large dataset and requires massive human intervention. This gave way to create a dataset, using sampling approach. The simplest method of sampling method is random sampling. Even, random sampling gives less code smells in this domain, i.e., the selected instances are less affected by the code smells in the training dataset. Here in this case, the researchers assigned class label instances with the help of detection rules proposed in the literature (Ferme, 2013). The researchers labelled 1 as an affected instance and 0 as an unaffected instance. The detection approach, composed of binary logical conditions (AND, OR). Figure 2 and Figure 3 shows, shotgun surgery and message chaining detection rules respectively. These rules will help to detect whether the method instances are smelly or not. The reason for choosing these detection rules is that, the threshold of the metrics are derived from a benchmark of selected 74 software system (Fontana F. A., 2015). The metrics CC, CM, and FANOUT are used to identify the characteristics of the shotgun surgery smell. Similarly, MaMCL, NMCS, and MeMCL are used to identify the characteristics of the message chaining smell. The computation of these metrics are defined in (Ferme, 2013). The outcome of this step results in a dataset.

Unsupervised Learning

The researchers have applied unsupervised learning (clustering) on the entire method level dataset to know the number of clusters that can be formed. In this work, the method instances can be either smelly or not. So, the two clusters are considered to be binary classifiers. The algorithm used for clustering is k-means. The method instances which are affected (positive) by the above rules are compared with the formed clusters to validate the instances. If the instance produces same cluster as it's label then it is considered else discarded for training dataset.

Table 3.Object oriented metrics

Project Level Metrics	Package Level Metrics	Class Level Metrics	Method Level Metrics
NOPK, NOCS, NOI, NOMNAMM, LOC etc.	NOCS, NOMNAMM, NOI, LOC, NOM etc.	NOII, NOAM, NOCS NMO, ATFD etc.	CYCLO, NOP, NMCS LOC, LAA etc

Figure 2. Shotgun surgery detection strategy

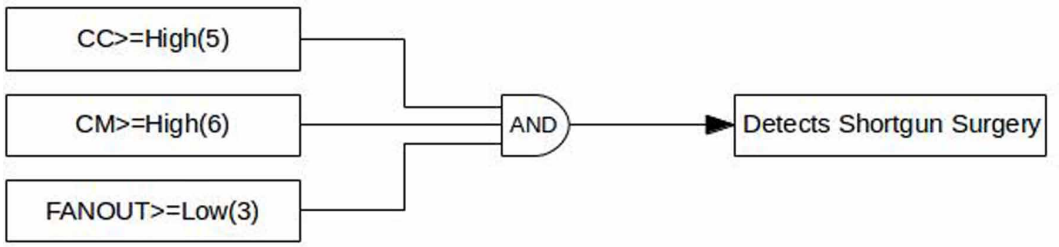
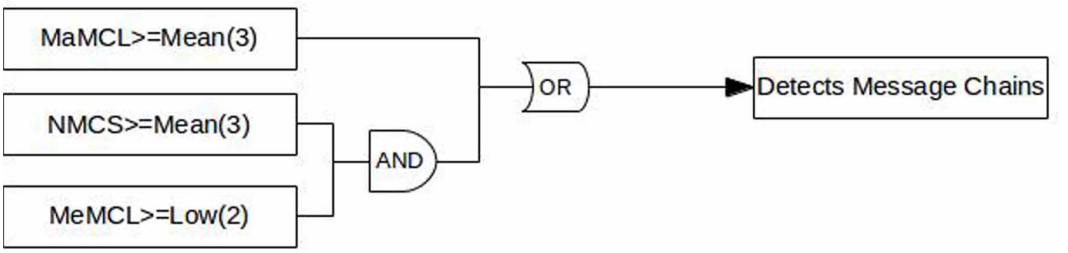


Figure 3. Message chain detection strategy



Stratified Random Sampling on Datasets

Table 4 reports, the results after applying detection rules on the entire Qualitus Corpus dataset.

It can be observed from Table 4 that the number of negatively affected instances are more when compared to positively affected instances. Using of open source system, resulted in increase of negative instances. Open source systems are used to produce the source code with poor quality (Stamelos, 2002). The recent study indicates that the open source systems and commercial systems are having almost the same code quality (Spinellis, 2008). From the observation it was found that when compared to commercial systems, pure open sources are giving better software structures (Spinellis, 2008; Capra, 2011). The Qualitas Corpus (Tempero, Anslow, Dietrich et al., 2010) has both "pure open source" and "open source" systems where there is a commercial participation. Thus, it is observed that affected smells (i.e., positive instances) detected using open source systems are less. This leads to highly imbalanced datasets (He, 2009). To balance the dataset, the researchers have used stratified random sampling approach which is organized as follows:

- For each project, group the negative instances for the dataset.
- Randomly, sample the negative instances of each group until approximately the double of the positive instances are obtained.
- For positive instances, unsupervised learning (clustering) is used for validation.
- The obtained positive and negative instances are placed in the training dataset.

An overview of the procedure is shown in Figure 4. This covers different domain instances having different characteristics.

EXPERIMENTATION AND RESULTS

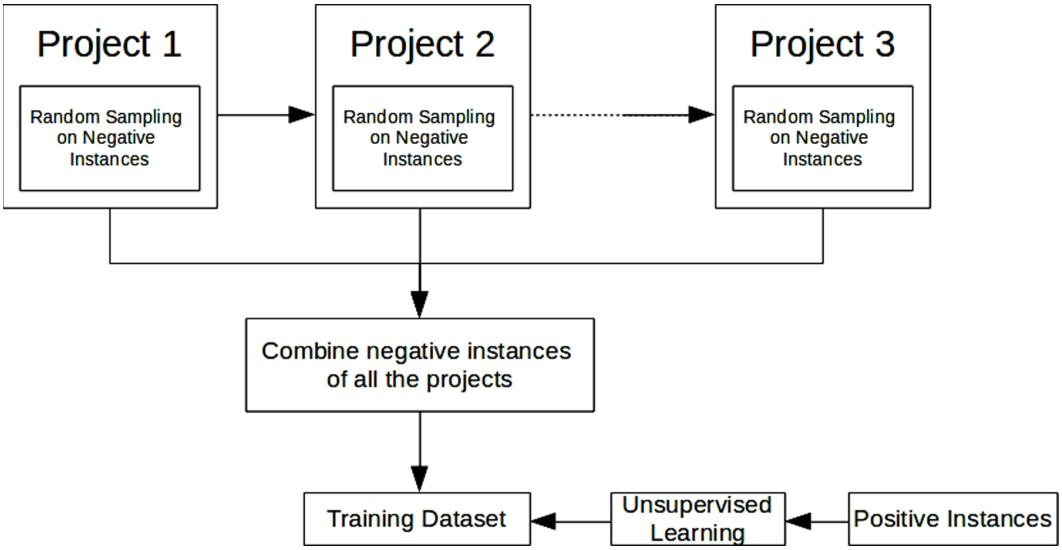
Experimental Setup

Researchers have used the application of supervised learning for the experiments and selected six known classification algorithms such as Bayesian networks, support vector machines, K-nearest neighbours, rule learner, decision trees, ensemble method (Random forest) and WEK (Hall, 2009) tool to provide the implementations of the selected algorithms. In general, tree based classifiers

Table 4. Detection strategies of two smells on entire dataset

Code Smells	Method Instances	Positive Instances	Negative Instances
Shotgun surgery	415995	600	415395
Message chaining	415995	673	415321

Figure 4.



are performing better than the other classifiers in (Fontana F. A., 2016) detecting the code smells. Following steps gives a short description of the selected algorithms:

1. **J48 (C4.5) algorithm** (Quinlan, 1993)**:** Decision tree tries to recursively partition the dataset into subsets by evaluating the normalized information gain (difference in entropy) resulting from choosing an attribute for splitting the data. The attribute with the highest information gain is used for every step. The training process stops when the resulting nodes contain instances of single classes or no such attribute that can be found with information gain. The authors have used the default parameters supported by WEKA which are given below.
    1. **Criterion:** It defines the function to measure the quality of split. J48 support Entropy (measure the level of impurity) for information gain.
    2. **minNumObj:** Minimum number of instances required to split the internal nodes. By default, the value is 2.
    3. **Confidence vector:** The parameter altered to test the effectiveness of post-pruning was labelled by WEKA as the confidence factor. It builds a full tree and then work back from the leaves, applying a statistical test at each stage. By default, the value is 0.25.
2. **Random Forest** (Breiman, 2001): It generates different decision trees based on randomly selected attributes and instances. These trees become a forest called "random forest tree". It conducts voting on all the instances to decide the class instance based on polling result. The default random forest tree parameters in WEKA are as follows:
    1. **maxDepth:** It indicates how deep the tree can be. The deeper the tree, the more splits it has, and captures more information about the data. By default, the value is 0 (unlimited) means the nodes are expanded until all leaves are pure or until all leaves contain less than minimum number of instances.
    2. **numTrees:** The number of trees to be generated. By default the value is set to 10.
    3. **minFeatues:** The number of attributes to be used in random selection is 7.
3. **JRip** (Cohen, 1995): It implements a propositional rule learner based on association rules. It is used to extract human understandable rules for code smells.

4. **Naive Bayes** (John, 1995)**:** It is a supervised classification algorithm based on the assumption that occurrence of certain attribute is independent of occurrence of other attribute.
5. **Sequential minimal optimization (SMO)** (Platt, 1998)**:** It is restricted to binary class. To apply support vector machine the researchers should implement the SMO algorithm in order to train the instances.
6. **K-nearest neighbours (k=2)** (Ah, 1991)**:** The learning is also known as "instance based" learning. It is based on similarity (distance) calculation between instances. In the classes – to - cluster evaluation, the researchers verified that the number of clusters are 2 in both the datasets. The error rate was minimal for 2 clusters, when compared to others.

Researchers used the cross validation (10-fold) technique to evaluate predictive models to find the best algorithms for the experimentation. They have applied three standard performance measures for ML algorithms.

- **Accuracy** is the percentage of correct prediction. It is insufficient to select a model when, the positive and negative instances are imbalanced, but this will never occur in this approach. As the researchers have balanced two datasets with stratified random sampling.
- **F-Measure** is the 2 * ((precision * recall) / (precision + recall)) i.e., harmonic mean of precision and recall.
- **Area under ROC** allows visualizing the performances of the classifier across all possible classification thresholds, thus helping to choose a threshold that approximately balances sensitivity and specificity.

Dataset Results

After applying stratified random sampling, the researchers have obtained two training datasets (one for each code smell) which are specified in Table 5. In Table 4, 600 positive instances are detected by the shotgun surgery rule. On applying unsupervised learning on the obtained instances, 141 instances belong to other cluster. So, researchers removed those instances and added remaining instances to the training dataset of shotgun surgery which are reported in Table 5. Similarly, among 673 positive instances detected by message chaining rule, 190 instances belong to other cluster are removed and remaining are added to the training dataset and reported in Table 5. From the table, it can be observed that the datasets are well balanced in terms of positive and negative instances. The supervised learning will take these datasets as input and trains the ML algorithms.

Algorithms Results

The results of the proposed method of two code smell (Shotgun Surgery, Message Chaining) datasets with 10-fold cross validation and performance metrics are shown in Table 6 and Table 7, respectively.

It can be observed from Table 6 report, J48 and JRip both gives 100 percentage accuracy. These two algorithms have given the best performance when compared to other algorithms, while the worst performance is shown by Naive Bayes based on F-measure and accuracy performance metrics. According to the area under the ROC metric, the best performance got from J48, JRip and Random Forest. The worst performance got from K-nearest neighbors (K = 1).

Table 5. Training datasets of two smells

Training Dataset	Method Instances	Positive Instances	Negative Instances
Shotgun surgery	1717	459	1258
Message chaining	1889	483	1406

Table 6. Shotgun surgery cross validation results

10-Fold Cross Validation							
Classifier	**True Positive Rate**	**False Positive Rate**	**Precision**	**Recall**	**Accuracy**	**F-Measure**	**Area Under ROC**
J48	100	0	100	100	100	100	1.00
JRip	100	0	100	100	100	100	1.00
Random Forest	99.9	0	99.9	99.9	99.8	99.9	1.00
SMO	94.5	0.074	94.6	94.5	94.4	94.5	0.93
KNN(k=2)	87.5	0.264	87.2	87.5	87.4	86.9	0.92
Naive Bayes	86.8	0.210	86.8	86.8	86.8	86.8	0.93

It can be observed from the Table 7 report that, JRip gives the best performance, while the worst performance is achieved by Naive Bayes based on F-measure and accuracy performance metrics. According to the area under the ROC metric, best performance is obtained by JRip, while the worst performance is seen in K-nearest neighbours (K=1).

Classifiers Comparison with ROC Curve

ROC curve is generally used to visualize the binary classifiers performance over all possible thresholds, and AUC (arguably) is used to show the advisable way of summarizing the performance into a single value. ROC curve is a 2D graph in which, specificity (false positive rate) is plotted on x-axis and sensitivity (true negative) is plotted on y-axis. The area under the ROC curve ranges between 0 and 1. The common rule to evaluate the classification algorithm performance is to find the area under the ROC (A-ROC) (Fawcett, 2006).

- If A-ROC < 0.5 means, something wrong;
- If A-ROC=0.5 means, it is not a good prediction;
- If 0.5 < A-ROC < 0.6 means, it is a poor prediction;
- If 0.6 < A-ROC < 0.7 means, it is a fair prediction;
- If 0.7 < A-ROC < 0.8 means, it is an acceptable prediction;

Table 7. Message chaining cross validation results

10-Fold Cross Validation							
Classifier	**True Positive Rate**	**False Positive Rate**	**Precision**	**Recall**	**Accuracy**	**F-Measure**	**Area Under ROC**
J48	99.6	0.005	99.6	99.6	99.6	99.6	0.99
JRip	99.6	0.001	99.6	99.6	99.5	99.6	0.99
Random Forest	99.5	0.002	99.5	99.5	99.5	99.5	0.99
SMO	99.4	0.002	99.4	99.4	99.4	99.4	0.99
KNN(k=2)	96.1	0.073	96.1	96.1	96.1	96.1	0.98
Naive Bayes	95.1	0.017	95.9	95.1	95.1	95.3	0.98

- If 0.8 < A-ROC < 0.9 means, it is an excellent prediction;
- If A-ROC >= 0.9 means, it is an outstanding prediction;
- If A-ROC = 1 means, it is a perfect prediction.

Table 6 and Table 7 reports that the result obtained for different classifiers can be compared with the help of area under ROC curve on two smells shown in Figure 5 and Figure 6, respectively.

It can be observed from the Figure 5 and Figure 6 that, all the classifiers of the two smells in area under ROC curve are getting close to value 1 and from these observations, it can be said that the experimental results have achieved high performance under ROC metric.

CODE SMELL DETECTION RULES

The algorithms J48 and JRip gives the human readable detection rules for the shotgun surgery and the message chaining.

Shotgun Surgery

For shotgun surgery, J48 produces decision tree, which can be expressed in terms of logical conditions (AND, OR).

```
(CC_method > 4) AND  (FANOUT_method > 2)
```

The rule detects shotgun surgery, if the method has more than four classes, call the method and the method is subjected to being changed. These two conditions are included in the shotgun surgery definition.

For shotgun surgery, JRip produces the following rule.

```
(CC_method  >=  5) AND (FANOUT_method >= 3)
```

This gives the same rule as J48 algorithm, i.e., both J48 and JRips algorithm gives the same rule for shotgun surgery.

Figure 5. Shotgun surgery area under ROC curve

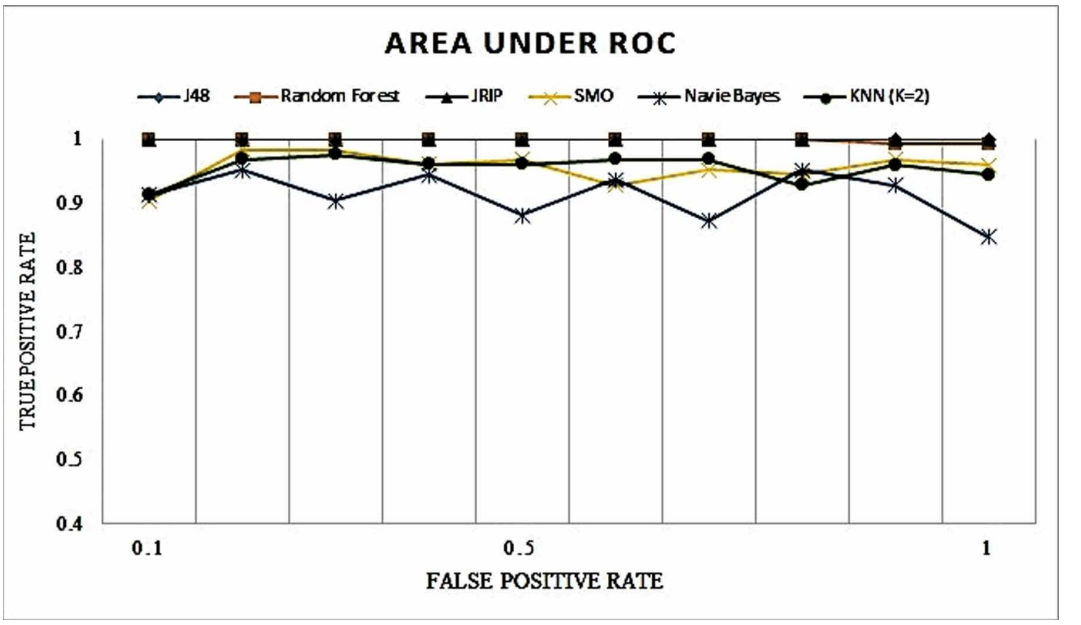
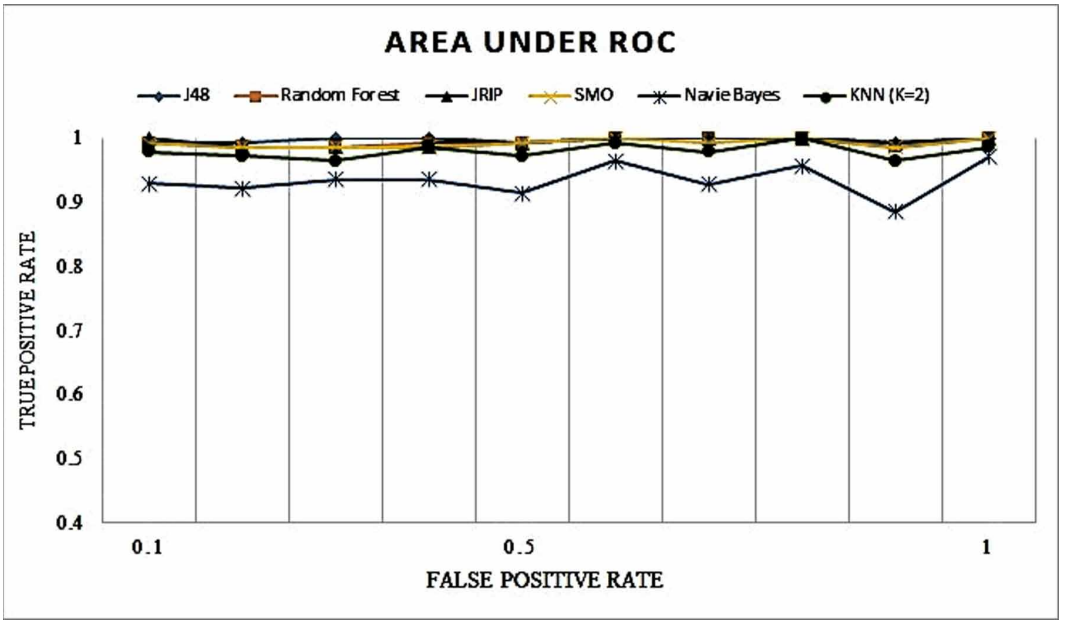


Figure 6. Message chaining area under ROC curve



Message Chaining

For Message chaining, J48 produces decision tree, which can be expressed in terms of logical conditions (AND, OR).

(*MeMCL_method > 0) AND (MaMCL_method >=3) OR (1)*

(MeMCL_method > 0) AND (NMCS_method > 2) OR (2)

(MeMCL_method > 0) AND (NMCS_method <= 2) AND (CINT_method > 5) OR (3)

(MeMCL_method > 0) AND (NMCS_method <= 2) AND (MaMCL_method > 2) OR (4)

The rules detect message chaining occurances, when at least one of the above four conditions is verified. The first, second and fourth rules are partly in the message chaining definition and in the third rule, third condition is not linked with the code smell definition.

For the message chaining, JRip produces the following rule.

(NMCS_method >= 3) AND (MaMCL_method >= 3)

Some parts of the detection rules are placed in the message chaining definition.

In J48, and JRip algorithm, all rules of conditions produced are part of the conceptual definition of message chain, except one (*CINT_method > 5*) condition which is not a part of conceptual definition. This is because, there is a noise in the dataset.

CONCLUSION AND FUTURE DIRECTIONS

In this paper, the researchers have evaluated and compared both code smell detection (the shotgun surgery and the message chaining) datasets with the help of supervised ML techniques. This technique is used to evolve a new instance to check whether a particular code smell is correctly classified or not. This methodology can be utilized to detect other code smell also.

The researchers used two detection rules from the literature to identify the code smells. These rules produce imbalanced dataset. In order to balance the training dataset, researchers have used stratified random sampling. Then applied unsupervised learning on the positive instances to remove some of the false positive instances. Researchers have considered six known learning algorithms to detect the code smells in the dataset. To evaluate the performances of each algorithm 3 standard performance measures are used, i.e. F-measure, accuracy and area under the ROC. For shotgun surgery, the best performance is shown by J48, JRip algorithms and for the message chaining, the best performance is produced by JRip. Both the algorithms J48 and JRip provide the human understandable rules.

To improve the performance, attribute selection algorithm is used in ML. By default, an attribute selection is done by random forest algorithm and it is noted that it has performed better than KNN and naive bayes algorithm. In future, an attempt can be made to explore the performances of KNN and naive bayes with the help of feature selection.

REFERENCES

Ah, D. W. (1991). Instance-based learning algorithms. *Machine Learning*, *6*(1), 37–66. doi:10.1007/BF00153759

Balmas, Francoise and Bergel, Alexandre and Denier, Simon and Ducasse, Stephane and Laval, Jannik and Mordal-Manet, Karine and Abdeen, Hani and Bellingard, Fabrice. (2010). oftware metric for Java and C++ practices.

Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, *28*(1), 4–17. doi:10.1109/32.979986

Booch, G. (2006). *Object oriented analysis \& design*. Pearson Education India.

Bowes, D. a. (2013). The inconsistent measurement of message chains. In *Proceedings of the 2013 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)* (pp. 62--68). IEEE.

Breiman, L. (2001). Random Forest. *Machine Learning*, *45*(1), 5–32. doi:10.1023/A:1010933404324

Capra, E. L., Francalanci, C., Merlo, F., & Rossi-Lamastra, C. (2011). Firms' involvement in Open Source projects: A trade-off between software structural quality and popularity. *Journal of Systems and Software*, *84*(1), 144–161. doi:10.1016/j.jss.2010.09.004

Cohen, W. W. (1995). Fast effective rule induction. In *Machine Learning Proceedings 1995* (pp. 115–123). Elsevier. doi:10.1016/B978-1-55860-377-6.50023-2

Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, *27*(8), 861–874. doi:10.1016/j.patrec.2005.10.010

Ferme, V. (2013). JCodeOdor: A software quality advisor through design flaws detection [Master's thesis]. University of Milano-Bicocca, Milano, Italy.

Fontana, F. A. (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, *11*, 5–1.

Fontana, F. A. (2015). Automatic metric thresholds derivation for code smell detection. In *Proceedings of the 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics (WETSoM)* (pp. 44-53). IEEE.

Fontana, F. A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, *21*(3), 1143–1191. doi:10.1007/s10664-015-9378-4

Fowler, M. a. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Hall, M. a. (2009). The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter, 11*, 10-18.

He, H. (2009). Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, *21*(9), 1263–1284. doi:10.1109/TKDE.2008.239

John, G. H. (1995). *Estimating continuous distributions in Bayesian classifiers*. Morgan Kaufmann Publishers Inc.

Kessentini, W., Kessentini, M., Sahraoui, H., Bechikh, S., & Ouni, A. (2014). A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Transactions on Software Engineering*, *40*(9), 841–861. doi:10.1109/TSE.2014.2331057

Machine learning for code smell detection. (n.d.). ESSeRE Lab. Retrieved from http://essere.disco.unimib.it/wiki/research/mlcsd

Opdyke, W. F. (1992). Refactoring object-oriented frameworks. University of Illinois at Urbana-Champaign, IL.

Platt, J. (1998). Sequential minimal optimization: A fast algorithm for training support vector machine.

Quinlan, J. R. (1993). *C4.5: Programs for machine leaning*. San Francisco, CA: Morgan Kaufmann Publisher Inc.

Spinellis, D. (2008). A tale of four kernels. In *Proceedings of the 30th international conference on Software engineering* (pp. 381--390). ACM.

Stamelos, I., Angelis, L., Oikonomou, A., & Bleris, G. L. (2002). Code quality analysis in open source software development. *Information Systems Journal*, *12*(1), 43–60. doi:10.1046/j.1365-2575.2002.00117.x

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., ... & Noble, J. (2010). The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Proceedings of the 2010 17th Asia Pacific Software Engineering Conference (APSEC)* (pp. 336-345). IEEE. doi:10.1109/APSEC.2010.46

APPENDIX : SOFTWARE METRICS CATEGORIZED INTO QUALITY DIMENSIONS

Figure 7. Software metrics categorized into quality dimensions

Quality dimension	Metric Name	Metric Label	Level or Granularity
Coupling	FANOUT	-	Class, Method
	FANIN	-	Class
	ATFD	Access to foreign data	Method
	FDP	Foreign Data Providers	Method
	RFC	Response For Class	Class
	CBO	Coupling Between Objects Class	Class
	CFNAMM	Called Foreign Not Accessor or Mutator Methods	Class, Method
	CINT	Coupling Intensity	Method
	MaMCL	Maximum Message Chain Length	Method
	MeMCL	Mean Message Chain Length	Method
	NMCS	Number of Message Chain Statements	Method
	CC	Changing Classes	Method
	CM	Changing Methods	Method
Size	LOC	Lines of Code	Project, Package, Class, Method
	LOCNAMM	Lines of Code Without Accessor or Mutator Methods	Class
	NOPK	Number of Packages	Project
	NOCS	Number of Classes	Project, Package
	NOM	Number of Methods	Project, Package, Class
	NOMNAMM	Number of Not Accessor or Mutator Methods	Project, Package, Class
	NOA	Number of Attributes	Class
Inheritance	DIT	Depth of Inheritance Tree	Class
	NOI	Number of Interfaces	Project, Package
	NOC	Number of Children	Class
	NMO	Number of Methods Overridden	Class
	NIM	Number of Inherited Methods	Class
	NOII	Number of Implemented Interfaces	Class
Encapsulation	NOAM	Number of Accessor Methods	Class
	NOPA	Number of Public Attribute	Class
	LAA	Locality of Attribute Accesses	Method
Complexity	CYCLO	Cyclomatic Complexity	Method
	WMC	Weighted Methods Count	Class
	WMCNAMM	Weighted Methods Count of Not Accessor or Mutator Methods	Class
	AMW	Average Methods Weight	Class
	AMWNAMM	Average Methods Weight of Not Accessor or Mutator Methods	Class
	MAXNESTING	Maximum Nesting Level	Method
	CLNAMM	Called Local Not Accessor or Mutator Methods	Method
	NOP	Number of Parameters	Method
	NOAV	Number of Accessed Variables	Method
	ATLD	Access to Local Data	Method
	NOLV	Number of Local Variable	Method

Thirupathi Guggulothu is working as a Research Scholar in the area of software engineering, University of Hyderabad and has done his post-graduation in the stream of computer science at the University of Hyderabad. He has worked as a research associate on the project titled "implementation of A5/1 attack" sponsored by DLRL in the University of Hyderabad. His research interests include software evolution, software maintenance, and machine learning. He qualified for UGC NET and TS SET in 2017.

Salman Abdul Moiz's is working as an Associate Professor in the School of Computer & Information Sciences at the University of Hyderabad. He worked as a Professor & Head of CSE at GITAM University, Hyderabad Campus. He has previously worked as research scientist at the Centre for Development of Advanced Computing, Bangalore. He is a member of the IEEE, ACM, IE, and EWB. His research interests include software engineering, software re-usability, mobile databases, and e-learning.