# Application of Naïve Bayes classifiers for refactoring Prediction at the method level

Rasmita Panigrahi[1]
*rasmita@giet.edu*
*Department of Computer Science and Engineering*
*GIET University, Gunupur, Odisha*

Sanjay Kumar kuanar[2]
*sanjay.kuanar@giet.edu*
*Department of Computer Science and Engineering*
*GIET University, Gunupur, Odisha*

Lov Kumar[3]
Department of Computer Science and *Information Systems,*
*BITS-PILANI, Hyderabad*
*Jawahar Nagar , Shameerpet Mandal, Hyderabad*
*lovkumar@hyderabad.bits-pilani.ac.in*

***Abstract*—Software refactoring is a technique of redesigning the existing code without changing its functionality in order to improve on code readability, code adherence, maintainability and portability. Recent years have witnessed the advancement of research in the field of improvement in code quality. The challenges involved in the field has attracted many software practitioners to identify methods or classes that need refactoring. We propose a model to predict refactoring candidates by Naïve Bayes classifiers (Gaussian, Multinomial and Bernoulli (GNB, MNB, BNB)) at method level refactoring in terms of AUC and Accuracy. Method level refactoring is carried out on data set from the Tera-Promise repository and then validated. Min-max normalization and Imbalancing techniques are then applied. Then using the Wilcoxon rank test, 8 sets of significant features are drawn out of 103 sets of input features.The experimental results on the performance of 3 Naïve Bayes classifiers shows that the Bernoulli Naïve Bayes classifier gives more accuracy as compared to the other two classifiers. Statistical tests applied on all features (AF) and significant features (SF), shows that significant features gives more accurate prediction than all features.**

***Keywords—Software refactoring, code smell, naïve Bayes classifier***

## I. INTRODUCTION

Refactoring as technology has emerged as the method of altering a software system in such a manner that it should not make any significant amend in the exterior functional characteristics of code but augments the internal structure to achieve higher productivity and QoS assurance. In fact, it intends to brush up code by reducing the probabilities of introducing bugs and defects. In other words, it signifies the process of enhancing the design once it has been prepared and ready to use or even underuse. Practically, the code goes on modifications while retaining the integrity of the system with a change in structure as per the design requirements. Undeniably, refactoring can be taken for poor design and rewrite it into well-formatted code, and therefore assessing a code for its refactoring probability can be of utmost significance to ensure QoS of the software systems. Refactoring is an important step in the development phase. In order to overcome the problem code ROT refactoring is used. It is a technique to control and improve the cleanliness of messy code. It can also be called as stepwise modification in the existing code. It is the method of reforming the software code without changing its behavior. It helps for reducing the complexity of semantics of code. It is also used for removing code smell from our data set. Code smells are not the bugs rather signifies the poor or weak design of the code, by which the development phase will be slow and there will be more chance of getting a failure. Machine learning algorithms can detect the code smell. Code smells are the warning for the existence of an antipattern. Code smells and antipatterns are used interchangeably. A code smell is a reference for antipatterns. Even if code smells are not incorrect but the delicacy of design they signify. Martin fowler specified 22 code smells as well as their proposed refactoring techniques.

TABLE I. LIST OF CODE SMELLS

| | |
|---|---|
| Long Method | Comments |
| Parallel inheritance hierarchies | Refused request |
| Data clumps | Data class |
| Shotgun surgery | Incomplete library class |
| Divergent change | Long Method |
| Feature Envy | Inappropriate intimacy |
| Primitive Obsession | Middle Man |
| Duplicate code | Message Chains |
| Large class | Temporary Fields |
| Switch statements | Duplicate code |
| Long parameter list | Alternative classes with different interfaces |

Table 1 shows the list of code smells. Out of all the code smells, we have considered only two code smells (Duplicate code, Long Method). Duplicate code means similarity of code in different places in the same program. consecutive duplicate codes are known as code clones. Duplicate code finding process is also known as code clone detection. The code smell of Duplicate code can be solved by Extract Method, Pullup Method and substitution Algorithm refactoring techniques. A method containing many lines code is known as a Long method ( more than 10 lines), which is sometimes very difficult to understand.

The code smell of long Method can be solved by Extract Method, Replace Temp with Query, Introduce parameter Object, Preserve the whole object, Replace method with method object, decomposition objects. Refactoring can be done through a set of techniques, which include composing methods, moving features between objects, organizing data, simplifying conditional expressions, making method calls center, dealing with generalization, simplifying method calls. Generally, software refactoring includes the following steps:-

TABLE II. REFACTORING STEPS

| |
|---|
| **Step 1:** specify the code segment where refactoring is needed. |
| **Step2:** Analysis of each refactoring cost/benefit. |
| **Step 3**: Application of refactoring technique |

In Table 2 identification of code smell presents step one which means to find the area where refactoring is needed. Step two shows the cost and benefit analysis of each possible refactoring technique for the identified code smell and Step three applies the best refactoring techniques, which includes less cost and more benefit.

## II. RELATED WORK

Refactoring is very important for the betterment of software maintenance and software quality**.** Refactoring can be done either automated or by using some tools**.** Software quality attributes are reusability, supportability, maintainability, security, testability. Authors studied 238 research papers from different sources (conferences, workshops, journals) about the code smells and anti-patterns as well as different datasets and tools for refactoring[1]. They have presented 22 number of code smells as well as their solutions. Bevota et al.[2] performed an experiential evaluation showing the association between refactoring activities and code smells by applying 12000 refactoring techniques. Martin Fowler has presented a catalog of refactoring on the website as well as published a book containing refactoring activities. Neha Kumari and Satwinder Singh [3] constructed a model of smell prediction to predict the code smell using supervised machine learning techniques. The model has been constructed by taking two tools (i.e. iPLASMA, PMD). Authors have shown a set of code smells as well as their corresponding detectors. From the previous studies, we can conclude that maximum code smells can be detected by iplasma detector, which detects the code smells like feature envy, long method, etc. In this work, we have considered two code smells which are specified in Table 3 (i.e. long method and duplicate code) and their corresponding refactoring activities.

TABLE III. CODE SMELL AND ITS SOLUTION

| **Code smell** | **Refactoring activities solution** |
|---|---|
| Long method | Decompositional objects, Replace method with method object, Replace Temp with Query, Introduce parameter object and preserve the whole object, Extract Method. |
| Duplicate code | Extract Method, Substitute Algorithm, pull Up Method, |

An approach is proposed by **Tarwani et al [4]** to evaluate the refactoring sequence through a greedy algorithm. An optimal solution is selected by this algorithm to generate a globally fined solution at each phase. The amount of software maintainability values is calculated by generating different sequences and applying source code. Their approach was able to identify the refactoring sequence as well as the finest refactoring to improve software maintainability and enhance software quality [5][6]. code smells can be removed by refactoring activities. Refactoring techniques can be implemented either class level, method level, package level or system level. Lov Kumar and Ashish Surekha[7] proposed a model for refactoring prediction at the class level. Authors have adapted LS-SVM for training the machine and PCA for the extraction of features and SMOTE to handle the imbalanced data. They have implemented three kernel functions (RBF, polynomial, Linear) for LS-SVM and computed the average value of AUC for LSS-VM (RBF Kernel) is 96%. Later on, Lov Kumar et al. [8] have implemented SMOTE and LS_SVM techniques for refactoring prediction at method level by various kernels. They have proved that LS-SVM by RBF kernel with SMOTE presents the highest result. Code smells can be generated through cohesion and coupling. Authors have proposed a refactoring algorithm based on complex network theory automated to remove the bad smells introduced through high cohesion and low coupling[9]. Machine learning algorithms can be used for predicting software refactoring. Authors have investigated the performance of machine learning algorithms (Neural Network, Decision tree.random forest, regression, support vector machine, and Naïve Bayes) for software refactoring prediction with 11,419 real-life projects from different sources[10]. As we know that code smell must be detected for refactoring prediction as well as they should be ordered first.so Authors have attempted to find the software metrics related to code smells and found the internal relationship between the code smells[11].

**RQ1**:- Can it be possible to predict the refactoring through the Naïve Bayes classifier.
**RQ2:** Which naïve Bayes classifier gives the high accuracy in refactoring prediction?
**RQ3:** Whether all features accuracy prediction is equivalent to the significant features of accuracy prediction?

## III. THE MOTIVATION OF RESEARCH AND AIM

Refactoring is used for cleansing and simplification of code. The code semantic is the same before and after code refactoring. The catalog of refactoring is presented by Martin Fowler. For example Extract method refactoring is the technique to create a new method from the existing code block. The refactoring process includes the modification of classes, methods, variables or fields in different applications. The developers are having the challenging task to predict, which part of code needs to refactor. The prior task for the developers to identify the method or classes or variables need to be refactored in a large scale software system by using either supervised or unsupervised learning techniques and object-oriented metrics as a feature or predictor.

This work depicts the novel and remarkable contributions, which can be the extension work of class level refactoring. The proposed work is based upon method level refactoring prediction. This work focuses on the requirement to predict refactoring using software metrics at the method level. This work emphasizes on five publicly assessable data sets (java projects antlr4, JUnit, mct, oryx, titan) and predicts the refactoring at the method level.

Naïve Bayes classifier is a technique of classification basing upon the Bayes theorem, so it is called as Probabilistic classifier. It gives better performance as compared to other supervised machine learning classifiers (i.e. logistic regression) with less training data. It performs fast and well in multi-class prediction. Specially MNB helps to arrange a new document according to the existing category. In other words, we can say that it is used for text classification due to its assumptions. Naïve base classifier is used to handle the real-time data with continuous distribution with the help of GNB.

## IV. Research Contribution

In our paper, we are considering the method level refactoring prediction by using Naïve Bayes Classifiers. A previous study reveals that nowhere only naïve Bayes classifier is used for refactoring prediction at method level as well as a comparison study on that in terms of ACCURACY and AUC. So, first of all, we have to calculate software metrics value at the method level. Out of all the features from every project we have extracted 8 significant features and their performance has been shown in terms of ACCURACY and AUC values. According to my knowledge, this is the first work that combines statistical analysis and performance of Naïve Bayes classifiers and feature descriptions ( AF and SF) in terms of ACCURACY and AUC.

## V. Work Experiment

The experimental data set has been drawn from the tera-PROMISE repository, which contains a set of experimental data sets on engineering topics (i.e source code analysis, code smell fault detection, refactoring, and source code metrics).

The tera-PROMISE is a renowned storehouse, which stores the publicly assessable open source projects. The reason for

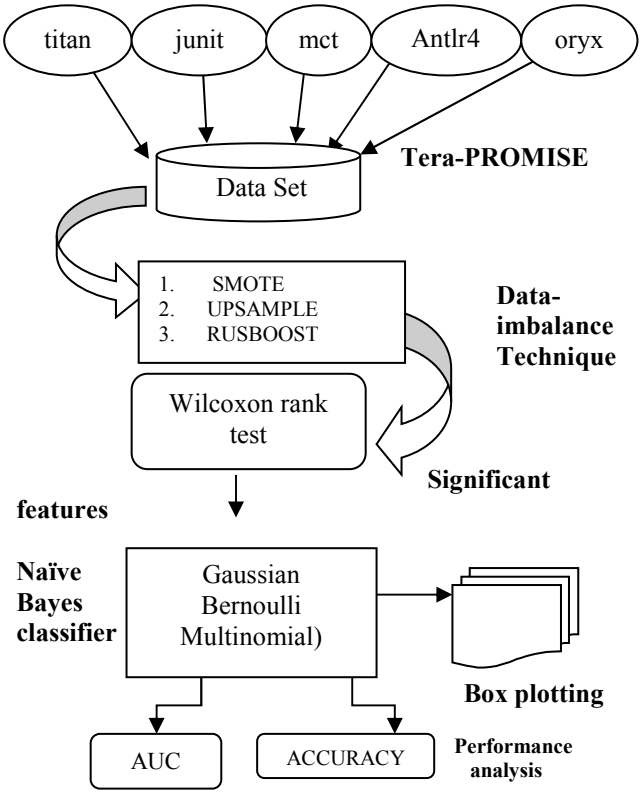


Fig. 1. Research schema and the proposed approach

choosing the tera-PROMISE repository is to make a standard for other researchers for easy replication of experiments.

Kedar et al.[12] validated the experimental data set manually as well as shared with the tera-PROMISE repository, where they calculated the source code metrics using a source meter tool that computes only class-level metrics. The authors have taken two successive releases of 7 open source java projects and extracted refactoring by the RefFinder tool[13]. The high-quality data set is manually validated by removing the false-positive occurrences created by the RefFinder tool.

## VI. Wilcoxon Rank Test

Wilcoxon test means a nonparametric test, which compares two separate groups as well as analyzes the differences between them. It can be either a Wilcoxon rank-sum test or Wilcoxon signed-rank test. Wilcoxon rank-sum test assigns the rank to each observation and then combines the observations with their corresponding ranks of all the groups whereas Wilcoxon signed-rank test computes the difference between each group of matched pairs and then checks whether their median differences are

equal to zero or not. In this paper, we have used the Wilcoxon signed-rank test for measuring the Gaussian Naïve Bayes, Multinomial Naïve Bayes, and Bernoulli Naïve Bayes classifiers performance. During the implementation of the Wilcoxon test, we have to make a parametric assumption for the NULL hypothesis with a particular p-value. In the paper, we have taken the p-value within the range of 0 to 1. If we are assuming some p-value then after the performance we have to test whether the computed p-value is less then the assumed p-value, then the Null hypothesis is accepted otherwise rejected. Being a statistical test it results in the statistical performance of each classifier is zero. In figure 2, GNB, MNB, BNB classifier's performance for refactoring prediction has been shown in the form of the Box-Plot graph. Box-plot is a combination of a box and a whisker plot, which shows a five-number summary of a data set. In a box-plot we are drawing a box from minimum, first quartile, median, third quartile, and maximum as well as it shows a vertical line through the median

## VII. Statistical analysis of Naïve Bayes classifiers in terms of AUC and ACCURACY

In this paper, three classifiers have been implemented on a dataset drawn from the tera-PROMISE repository and ACCURACY and AUC values have been computed in Table 4 and Table 5. Accuracy is the degree of closeness of measured value to its actual value. Sometimes basing up on accuracy prediction we cannot say about the best model for refactoring, so AUC prediction is required for every proposed model because AUC determines the best method of prediction. The mean value of AUC prediction for all the 3 classifiers is different (GNB=70%, MNB=64%, BNB=78%) whereas the median value for GNB and MNB is likely the same. The minimum AUC value for GNB and BNB is likely the same, but the minimum AUC value for MNB is much lower than the other two classifiers. The maximum AUC value for GNB Is 97 % whereas the maximum AUC value for the other two classifiers is 94% and 95%. The statistical analysis of all the 3 classifiers has been represented in the form ACCURACY Box-Plot in Fig 2. In the box plot graph box-plots of GNB is much larger then BNB and MNB, where the box-plot for BNB is very much shorter the GNB and MNB. In the AUC box-plot graph, all the 3 classifiers are likely the same sized box-plots.

**The answer of RQ1**:-In this work we have considered 5 data sets and applied three Naïve Bayes classifiers (GNB, MNB, BNB) for method level refactoring prediction. Here we have tested 25% statistical analysis (min, max, mean, median) from 75% trained data set.

**The answer of RQ2**:-This work implements three Naïve Bayes classifiers (GNB, MNB, BNB) and the BNB classifier gives the best performance as compare to MNB and GNBbasing up their mean value in terms of AUC and ACCURACY, which is shown in Table 7.

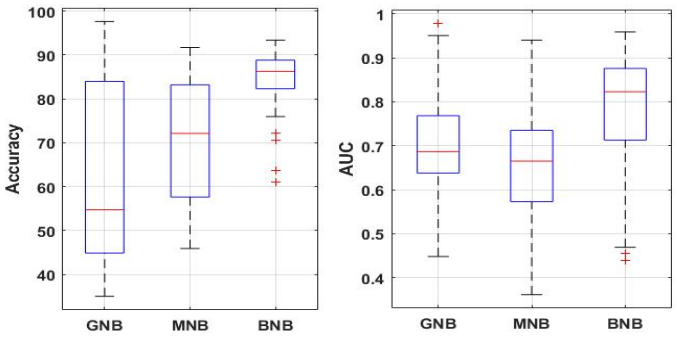


Fig. 2. Box-plot diagrams of AUC and ACCURACY values for 3 Naïve Bayes Classifiers

## VIII. Model accuracy

In this work, we have proposed a model by applying Wilcoxon signed-rank test for normalization and feature extraction and three Naïve Bayes classifiers (Gaussian, Bernoulli, Multinomial) for refactoring prediction in the form of AUC and ACCURACY. The effectiveness of the proposed model can be known from their mapping value or significance test. If the mapping value from one classifier to the same classifier is less than 0.05( p-value), then that model is not acceptable but the following Table shows the mapping value from GNB to GNB is 1 and MNB to MNB is 1 and BNB to BNB is 1. This work also focuses on the performance of the model by considering all features as well as all features. Here also the mapping value of AF to AF is 1 and SF to SF is 1. The result is presented in table 5.

TABLE IV. CLASSIFIERS SIGNIFICANT TEST

| Classifiers | GNB | MNB | BNB |
|---|---|---|---|
| GNB | 1 | 0.040276 | 0.000859 |
| MNB | 0.040276 | 1 | 5.49E-07 |
| BNB | 0.000859 | 5.49E-07 | 1 |

**Feature selection**: Feature refers to the individual assessable property of an observable fact. Feature selection is an important aspect of any classification algorithm.

TABLE V. FEATURES SIGNIFICANT TEST

| Features | AF | SF |
|---|---|---|
| AF | 1 | 0.839157 |
| SF | 0.839157 | 1 |

Every effective classification algorithm needs a set of instructive, selective and self-sufficient features. By selecting good features we can achieve over fitting reduction and accuracy improvement as well as training time reduction. As we are proposing a new model means It must improve the accuracy as well as must reduce the

training time of the model. We have downloaded the data set from the tera-PROMISE Repository and the source meter tool helped us to extract the source code metrics from the dataset which is publicly accessible. Feature selection is also a statistical test that selects the features basing upon their strongest association with the output variable. Best features can be selected by using a set of techniques ( Filter, Wrapper, and Embedded). In this work, we have used a ranking method for the best feature selection for our proposed model. The ranking method assigns the ranks to every feature and selects the best features. In figure3 it has been shown that how to filter method is used for measuring the performance of machine learning techniques by choosing the filter method for feature selection.

## IX. The Statistical Description of AF and SF in Terms of AUC and ACCURACY

We have extracted eight number of significant features from the data set which contains five open source projects taken from the tera-PROMISE repository. From Table-6, the conclusion of statistical test performance on all features shows low performance than computed significant features in terms of their AUC value whereas Table–9 shows the better performance of SF than AF in terms of ACCURACY computed on the 25% tested data from 75% training data set. The minimum AUC value for AF is 35% whereas the minimum value for SF is 40%. The maximum AUC value for AF is likely to be the same as the maximum AUC value of SF.

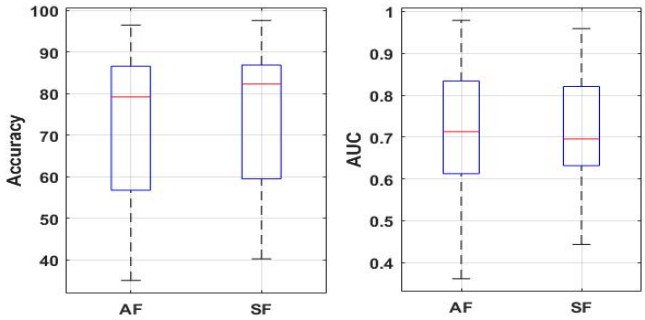


Fig. 3. Box-plot Diagram of AUC and ACCURACY for AF and SF

**The answer of RQ3:** In this paper, we have taken 5 data sets from the tera-PROMISE repository. We have predicted the AUC and ACCURACY basing up on All Features (AF) and Significant Features (SF) and represented in the form of box- plot diagram in Fig. 3. We have done the statistical analysis also and we have got the result better result in form of AUC and ACCURACY by considering the SF as compared to AF. The result has been represented in Tables 8 and 9 as well as in the form of a box plot diagram in Fig.3.

The mean AUC value of SF is 2% more than the AUC mean value of AF. The AUC and ACCURACY prediction has been presented in the form of Box-plot in figure 3. From the box-plot diagram, we can get the conclusion that The AUC box-plots for AF and SF are likely the same whereas the ACCURACY box-plots for AF are a little bit bigger than ACCURACY box-plot of SF.

## X. Conclusion

This work considers 5 different publicly accessible open-source projects which are validated manually to compute source code metrics by a source meter tool. Actually, we have computed 103 source code metrics then selected 8 significant features using Wilcoxon rank test by deleting unwanted features with the data unbalanced technique(SMOTE, UPSAMPLE, RUSBoost) and then applied machine learning algorithm for classifying which methods needs refactoring by 3 Naïve Bayes Classifiers (GNB, MNB, BNB). The statistical performance of all the classifiers depicts the result that AUC and ACCURACY measure value for BNB is better performance than the other two classifiers. The AUC and ACCURACY measure value for SF gives a better result than the AF.

TABLE VI. Statistical Analysis of Naïve Bayes Classifiers in Terms of AUC

| | AUC | | | | | |
|---|---|---|---|---|---|---|
| | **Min** | **Max** | **Mean** | **Median** | **25%** | **75%** |
| GNB | 0.448246 | 0.978889 | 0.709449 | 0.686576 | 0.637911 | 0.768452 |
| MNB | 0.361491 | 0.940397 | 0.648895 | 0.665184 | 0.572851 | 0.735329 |
| BNB | 0.439689 | 0.959144 | 0.78782 | 0.822951 | 0.71306 | 0.875837 |

TABLE VII. STATISTICAL ANALYSIS OF NAÏVE BAYES CLASSIFIERS IN TERMS OF ACCURACY

| Accuracy | | | | | | |
|---|---|---|---|---|---|---|
| | **Min** | **Max** | **Mean** | **Median** | **25%** | **75%** |
| GNB | 35.0649 | 97.5983 | 63.82624 | 54.75415 | 44.898 | 83.9752 |
| MNB | 45.9459 | 91.7031 | 70.66389 | 72.1304 | 57.6531 | 83.1878 |
| BNB | 61.039 | 93.3673 | 84.64932 | 86.2445 | 82.3144 | 88.8199 |

TABLE VIII. STATISTICAL PERFORMANCE OF AF AND SF IN TERMS OF AUC

| AUC | | | | | | |
|---|---|---|---|---|---|---|
| | **Min** | **Max** | **Mean** | **Median** | **25%** | **75%** |
| AF | 35.0649 | 96.5066 | 72.03121 | 79.2208 | 56.76218 | 86.60715 |
| SF | 40.2597 | 97.5983 | 74.06175 | 82.3144 | 59.5319 | 86.87548 |

TABLE IX. STATISTICAL PERFORMANCE OF AF AND SF IN TERMS OF ACCURACY

| ACCURACY | | | | | | |
|---|---|---|---|---|---|---|
| | **Min** | **Max** | **Mean** | **Median** | **25%** | **75%** |
| AF | 0.361491 | 0.978889 | 0.712722 | 0.71306 | 0.612586 | 0.833943 |
| SF | 0.443306 | 0.959144 | 0.718054 | 0.695489 | 0.631587 | 0.820809 |

## REFERENCE


1. Singh, S., & Kaur, S. (2018). A systematic literature review: Refactoring for disclosing code smells in object-oriented software. Ain Shams Engineering Journal, 9(4), 2129-2151.
2. Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., & Palomba, F. (2015). An experimental investigation on the innate relationship between quality and refactoring. Journal of Systems and Software, 107, 1-14.
3. Neha Kumari and Satwinder Singh, "improving smell prediction: Developing an Improved Model with Supervised Learning Techniques",2017, Indian journal of science and technology, vol 10(24).
4. S. Tarwani and A. Chug, "Sequencing of refactoring techniques by Greedy algorithm for maximizing maintainability," 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Jaipur, 2016, pp. 1397-1403.
5. G. Soares, R. Gheyi and T. Massoni, "Automated Behavioral Testing of Refactoring Engines," in IEEE Transactions on Software Engineering, vol. 39, no. 2, pp. 147-162, Feb. 2013.
6. Y. Wang, H. Yu, Z. Zhu, W. Zhang, and Y. Zhao, "Automatic Software Refactoring via Weighted Clustering in Method-Level Networks," in IEEE Transactions on Software Engineering, vol. 44, no. 3, pp. 202-236, 1 March 2018.
7. Kumar, L., & Sureka, A. (2017, December). Application of LSSVM and SMOTE on seven open-source projects for predicting refactoring at the class level. In the 2017 24th Asia-Pacific Software Engineering Conference (APSEC) (pp. 90-99). IEEE.
8. Kumar, L., Satapathy, S. M., & Krishna, A. (2018, December). Application of SMOTE and LSSVM with Various Kernels for Predicting Refactoring at Method Level. In International Conference on Neural Information Processing (pp. 150-161). Springer, Cham.
9. Wang,Y.,yu,H.,Zhang, W., & Zhao,Y.(2017).Automatic Software Refactoring via Weighted Clustering in method level Networks.IEEE transaction on software engineering,44(3), 202-236.
10. Aniche, m.,maziero,e.,durreli,r.,&durelli,v.(2020) The effectiveness of supervised machine learning algorithms in predicting software refactoring, Arxiv preprint arxiv:2001.03338.
11. ICETCE2019: EMERGING TECHNOLOGIES IN COMPUTER ENGINEERING: MICROSERVICES IN BIG DATA ANALYTICS PP 250-260 **AN APPROACH TO SUGGEST CODE SMELL ORDER FOR REFACTORING**
12. Kedar, I., Hegedus, P., Ferenc, R., Gyimothy, "A code refactoring dataset and its assessment regarding software maintainability. In: 2016 IEEE 23rd International conference on software analysis, Evolution, and Reengineering(SANER), vol. 1, pp. 559-603. IEEE (2016)
13. KIM, M., GEE, M., LOH, A., & RACHATASUMRIT, N. (2010, NOVEMBER). REF-FINDER: A REFACTORING RECONSTRUCTION TOOL BASED ON LOGIC QUERY TEMPLATES. IN PROCEEDINGS OF THE EIGHTEENTH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING (PP. 371-372). ACM.