MACHINE LEARNING BASED AN AUTOMATIC ADVISOR FOR REFACTORING SOFTWARE CLONES

1Mr. RAMBABU ATMAKURI - Head, Department of CSE, Anurag College of Engineering (Aushapur, Ghatkesar, Telangana 501301)
2Mr.BATTU BALA KRISHNA - Student, Department of CSE, Anurag College of Engineering (Aushapur, Ghatkesar, Telangana 501301)
3Ms.HEREMAT SUMASREE - Student, Department of CSE, Anurag College of Engineering (Aushapur, Ghatkesar, Telangana 501301)
4Mr. KARRI SANTHOSH KUMAR - Student, Department of CSE, Anurag College of Engineering (Aushapur, Ghatkesar, Telangana 501301)
5Mr.KOLLOJU SANJAY - Student, Department of CSE, Anurag College of Engineering (Aushapur, Ghatkesar, Telangana 501301)

ABSTRACT:
To assist developers refactored code and to enable improvements to software quality when numbers of clones are found in software programs, we require an approach to advise developers on what a clone needs to refactor and what type of refactoring is needed. This paper suggests a unique learning method that automatically extracts features from the detected code clones and trains models to advise developers on what type needs to be refactored. Our approach differs from others which specify types of refactored clones as classes and creates a model for detecting the types of refactored clones and the clones which are anonymous. We introduce a new method by which to convert refactoring clone type outliers into Unknown clone set to improve classification results. We present an extensive comparative study and an evaluation of the efficacy of our suggested idea by using state-of-the-art classification models

I. INTRODUCTION:

CODE clones are pairs of code fragments which have a high degree of similarity or which are identical. Code clones might cause software maintenance to be more difficult and a system's source codes more difficult to understand. Code cloning is a popular practice in the software development process for a number of reasons, such as reusing code by "copy-and-paste" to increasing the speed of writing the code . There are various clone detector techniques which attempt to find code fragments which have a high number of similarities in the system's source code. Additionally, there have been various refactoring clone tools developed which change the structure of detected code clones without altering code fragment behaviour. The refactoring code clones are a method by which to minimize the chances of introducing a bug. Refactoring, or removing, is utilized for improving software comprehensibility and maintainability. Although have shown that clone refactoring cannot solve software quality improvements for two reasons. Firstly, clones often have a short lifespan. Refactoring is less effective if there are block branches in a short distance. Secondly, longer living clones which have been altered with another element in the same class are difficult to remove or refactor. Additionally, it is a bug which can be simply corrected as the source code can be easily understood, which allows improvement of malleability resulting in code extensibility. Our approach provides different types of refactoring recommendation to a developer for preventing to remove the positive side of code clones and builds a training model after removing outliers to improve the results. Our tool can be built and used to minimize bugs in a system. Our study can improve clone maintenance by removing duplication code by identifying refactoring clones. Also, the possibility of bad design for a system, difficulty in a system improvement or modification, introducing a new bug, can be

decreased by identifying and refactoring clones. In addition, our study can be utilized by various applications such as source code or text plagiarism, malware detection, obfuscated code detection. In summary, the main contributions of this paper are: • A presentation of a new machine learning framework that automatically extracts features from the

1.1**Objective of the Project:**

To assist developers refactored code and to enable improvements to software quality when numbers of clones are found in software programs, we require an approach to advise developers on what a clone needs to refactor and what type of refactoring is needed. This paper suggests a unique learning method that automatically extracts features from the detected code clones and trains models to advise developers on what type needs to be refactored. Our approach differs from others which specify types of refactored clones as classes and creates a model for detecting the types of refactored clones and the clones which are anonymous. We introduce a new method by which to convert refactoring clone type outliers into Unknown clone set to improve classification results. We present an extensive comparative study and an evaluation of the efficacy of our suggested idea by using state-of-the-art classification models

II. LITERATURE SURVEY

‘‘Code clone detection experience at Microsoft,’’

Cloning source code is a common practice in the software development process. In general, the number of code clones increases in proportion to the growth of the code base. It is challenging to proactively keep clones consistent and remove unnecessary clones during the entire software development process of large-scale commercial software. In this position paper, we briefly share some typical usage scenarios of code clone detection that we collected from Microsoft engineers. We also discuss our experience on building XIAO, a code clone detection tool, and the feedback we have received from Microsoft engineers on using XIAO in real development settings.

‘‘Automatic clone recommendation for refactoring based on the present and the past,’’

When many clones are detected in software programs, not all clones are equally important to developers. To help developers refactor code and improve software quality, various tools were built to recommend clone-removal refactoring based on the past and the present information, such as the cohesion degree of individual clones or the co-evolution relations of clone peers. The existence of these tools inspired us to build an approach that considers as many factors as possible to more accurately recommend clones. This paper introduces CREC, a learning-based approach that recommends clones by extracting features from the current status and past history of software projects. Given a set of software repositories, CREC first automatically extracts the clone groups historically refactored (R-clones) and those not refactored (NR-clones) to construct the training set. CREC extracts 34 features to characterize the content and evolution behaviors of individual clones, as well as the spatial, syntactical, and co-change relations of clone peers. With these features, CREC trains a classifier that recommends clones for refactoring. We designed the largest feature set thus far for clone recommendation, and performed an evaluation on six large projects. The results show that our approach suggested refactorings with 83% and 76% F-scores in the within-project and cross-project settings. CREC significantly outperforms a state-of-the-art similar approach on our data set, with the latter one achieving 70% and 50% F-scores. We also compared the effectiveness of different factors and different learning algorithms.

‘‘Method-level code clone modification using refactoring techniques for clone maintenance,’’

Researchers focused on activities such as clone maintenance to assist the programmers. Refactoring is a well-known process to improve the maintainability of the software. Program refactoring is a technique to improve readability, structure, performance, abstraction, maintainability, or other characteristics by transforming a program. This paper contributes to a more unified approach for the phases of clone maintenance with a focus on clone modification. This approach uses the refactoring technique for clone modification. To detect the clones 'CloneManager' tool has been used. This approach is implemented as an enhancement to the existing tool CloneManager. The enhanced tool is tested with the open source projects and the results are compared with the performance of other three existing tools.

''An empirical study of code clone genealogies,''

It has been broadly assumed that code clones are inherently bad and that eliminating clones by refactoring would solve the problems of code clones. To investigate the validity of this assumption, we developed a formal denition of clone evolution and built a clone genealogy tool that automatically extracts the history of code clones from a source code repository. Using our tool we extracted clone genealogy information for two Java open source projects and analyzed their evolution. Our study contradicts some conventional wisdom about clones. In particular, refactoring may not always improve software with respect to clones for two reasons. First, many code clones exist in the system for only a short time; extensive refactoring of such short-lived clones may not be worthwhile if they are likely diverge from one another very soon. Second, many clones, especially long-lived clones that have changed consistently with other elements in the same group, are not easily refactorable due to programming language limitations. These insights show that refactoring will not help in dealing with some types of clones and open up opportunities for complementary clone maintenance tools that target these other classes of clones.

''Frequency and risks of changes to clones,"

Code Clones - duplicated source fragments - are said to increase maintenance effort and to facilitate problems caused by inconsistent changes to identical parts. While this is certainly true for some clones and certainly not true for others, it is unclear how many clones are real threats to the system's quality and need to be taken care of. Our analysis of clone evolution in mature software projects shows that most clones are rarely changed and the number of unintentional inconsistent changes to clones is small. We thus have to carefully select the clones to be managed to avoid unnecessary effort managing clones with no risk potential.

''Recommending clones for refactoring using design, context, and history,''

Developers know that copy-pasting code (aka code cloning) is often a convenient shortcut to achieving a design goal, albeit one that carries risks to the code quality over time. However, deciding which, if any, clones should be eliminated within an existing system is a daunting task. Fixing a clone usually means performing an invasive refactoring, and not all clones may be worth the effort, cost, and risk that such a change entails. Furthermore, sometimes cloning fulfils a useful design role, and should not be refactored at al. And clone detection tools often return very large result sets, making it hard to choose which clones should be investigated and possibly removed. In this paper, we propose an automated approach to recommend clones for refactoring by training a decision tree-based classifier. We analyze more than 600 clone instances in three medium-to large-sized open source projects, and we collect features that are associated with the source code, the context, and the history of clone instances. Our approach achieves a precision of around 80% in recommending clone refactoring

instances for each target system, and similarly good precision is achieved in cross-project evaluation. By recommending which clones are appropriate for refactoring, our approach allows for better resource allocation for refactoring itself after obtaining clone detection results, and can thus lead to improved clone management in practice.

III. SYSTEM ANALYSIS

3.1 EXISTING SYSTEM:

Several studies are related to code clone refactoring. Higo et al. [7] suggest a method that refactors code clones using existing refactoring patterns such as the Extract and Pull Up Method. This research performed fully automated refactoring without developer intervention. The developer should evaluate refactoring based on their preference and indicate any clone which is a probable candidate for refactoring. Conversely, our work extracts features and relies on machine learning to build our model and classify clones according to the type of refactored clone and those which are not refactored. Next, the developer evaluated the refactoring clones. Higo et al. [8] suggested a method that detects refactoring-oriented code clone to improve the usefulness and applicability of the software maintenance method. Higo et al. [9] proposed a refactoring method for merging software clones. Their technique can detect a refactoring-oriented code clone in a general clone detected by token-based or text-based clone detection tools. We refactor clones using AST-based and PDG-based clone detection tools.

Disadvantage:

- Less Accuracy

3.2 PROPOSED SYSTEM:

In this paper for the Unknown set classification, our adopted work model combines supervised learning classifiers and outlier detection for unknown classes model. This paper discusses the common and recent classification algorithms used for refactoring code clone classification and an outlier detection model combined for classifying the test examples as belonging to known or unknown class sizes. The improved performances of our classifier model are reliant upon its closed set validation. Model validation in machine learning is the process whereby trained models are evaluated with testing datasets. The testing dataset in closed set validation contains examples which belong to known classes. We ran an outlier algorithm for datasets to find the data points which have considerably dissimilarity or inconsistency with the other given data points. Then, the data point classes are changed into unknown classes. After detecting outlier data points, we build our model for closed-set classification and perform analysis of their performance after training. We train and test our classifier with vectors of datasets.

Advantage:

- High Accuracy

IV. MODULES

1. User: This module is responsible for handling user interactions and providing recommendations, guidance or suggestions users. Typically software developers to maintainers, to help them refactor or eliminate code clones in their codebase**.**

2. System: Features will be converted into trainand test records andthen based on similarity between code modules class label will be assigned as 0 or 1. If code contains so many similar words then 1 will be assigned other wise 0 will be assigned. Using this module various machine learning algorithms will be applied such as SVM, KNN, Bagging classifier and Random Forest.

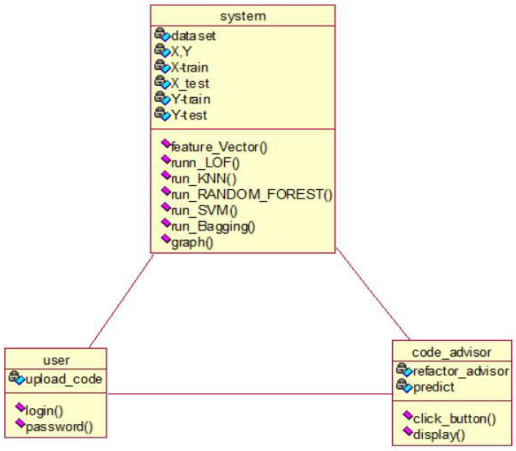
V. SYSTEM DESIGN

Class Diagram:

The class diagram is the main building block of object oriented modeling. It is used both for general conceptual modeling of the systematic of the application, and for detailed modeling translating the models into programming code.

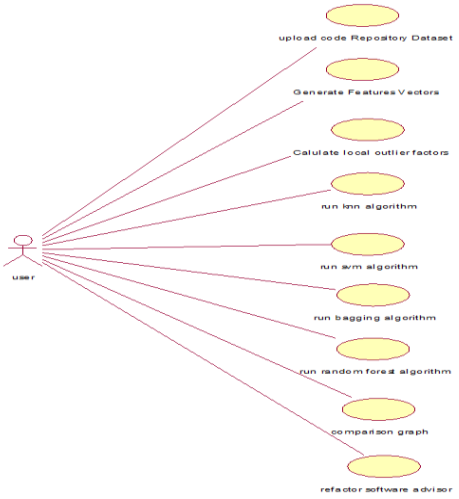
Class diagrams can also be used for data modeling. The classes in a class diagram represent both the main objects, interactions in the application and the classes to be programmed. In the diagram, classes are represented with boxes which contain three parts:

- The upper part holds the name of the class
- The middle part contains the attributes of the class
- The bottom part gives the methods or operations the class can take or undertake



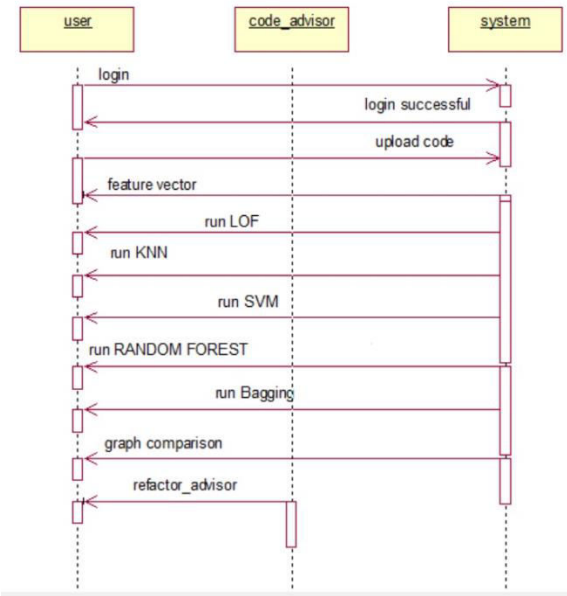
Use case Diagram:

A **use case diagram** at its simplest is a representation of a user's interaction with the system and depicting the specifications of a use case. A use case diagram can portray the different types of users of a system and the various ways that they interact with the system. This type of diagram is typically used in conjunction with the textual use case and will often be accompanied by other types of diagrams as well.



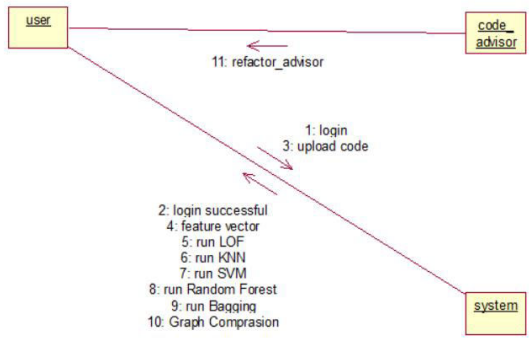
Sequence diagram:

A sequence diagram is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called event diagrams, event scenarios, and timing diagrams.



Collaboration diagram:

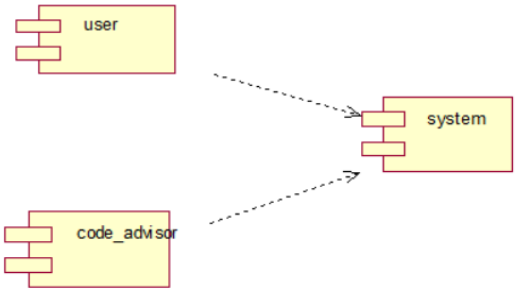
A collaboration diagram describes interactions among objects in terms of sequenced messages. Collaboration diagrams represent a combination of information taken from class, sequence, and use case diagrams describing both the static structure and dynamic behaviour of a system.



Component Diagram:

In the Unified Modelling Language, a component diagram depicts how components are wired together to form larger components and or software systems. They are used to illustrate the structure of arbitrarily complex systems.

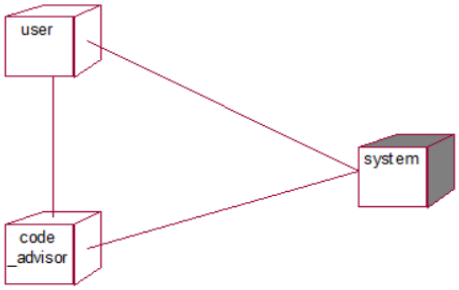
Components are wired together by using an assembly connector to connect the required interface of one component with the provided interface of another component. This illustrates the service consumer - service provider relationship between the two components.



Deployment Diagram:

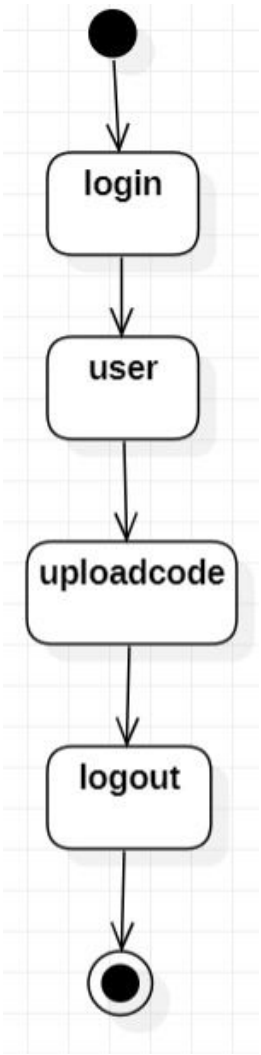
A **deployment diagram** in the Unified Modeling Language models the *physical* deployment of artifacts on nodes. To describe a web site, for example, a deployment diagram would show what hardware components ("nodes") exist (e.g., a web server, an application server, and a database server), what software components ("artifacts") run on each node (e.g., web application, database), and how the different pieces are connected (e.g. JDBC, REST, RMI).

The nodes appear as boxes, and the artifacts allocated to each node appear as rectangles within the boxes. Nodes may have sub nodes, which appear as nested boxes. A single node in a deployment diagram may conceptually represent multiple physical nodes, such as a cluster of database servers.



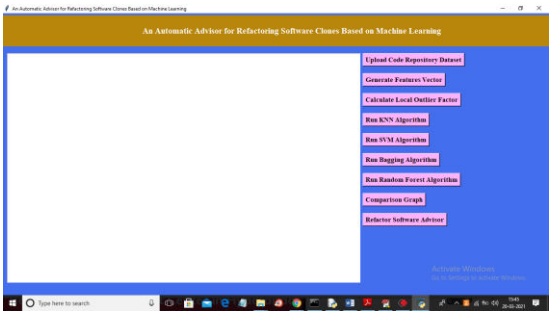
Activity Diagram:

Activity diagram is another important diagram in UML to describe dynamic aspects of the system. It is basically a flow chart to represent the flow form one activity to another activity. The activity can be described as an operation of the system. So the control flow is drawn from one operation to another. This flow can be sequential, branched or concurrent

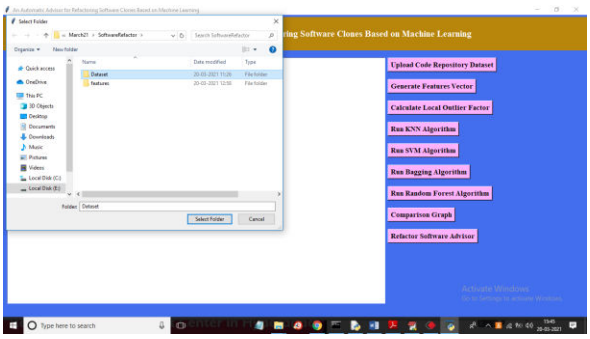


VI. SCREEN SHOTS:

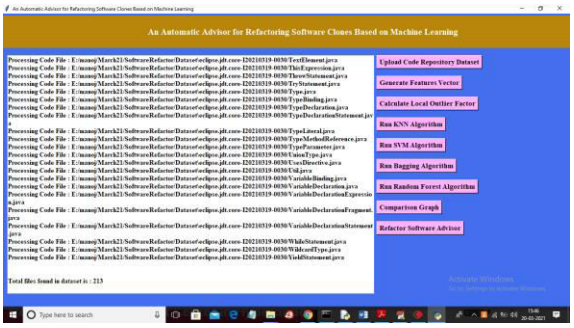
To run project double click on 'run.bat' file to get below screen



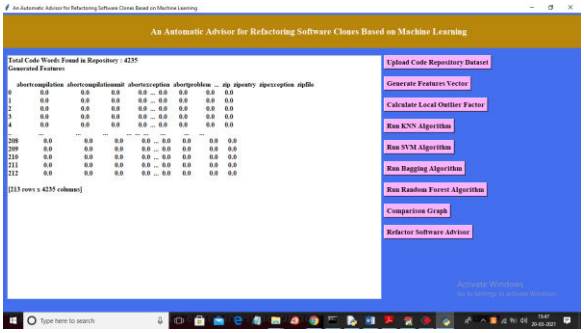
In above screen click on 'Upload Code Repository Dataset' button and upload dataset



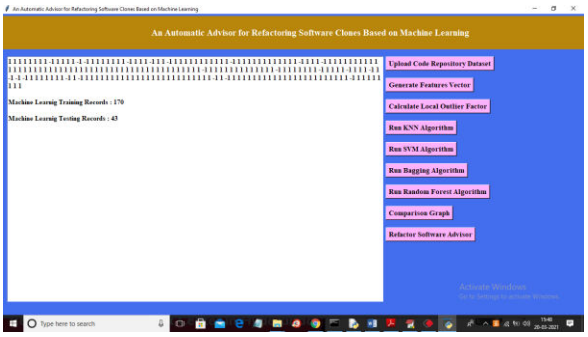
In above screen selecting and uploading 'Dataset' folder and then click on 'Select Folder' button to load dataset and then wait for few second so application read all code file and then will get below screen



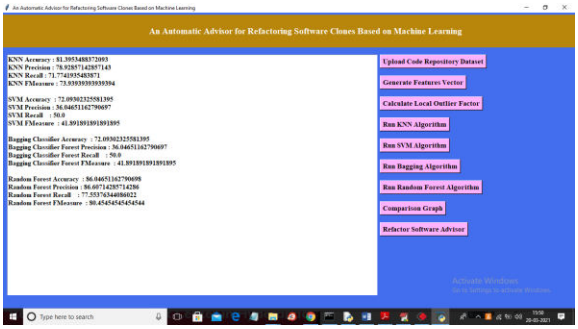
In above screen application read each code file and then process it and total java files found in dataset is 213 and now click on 'Generate Features Vector' button to convert above code into vector



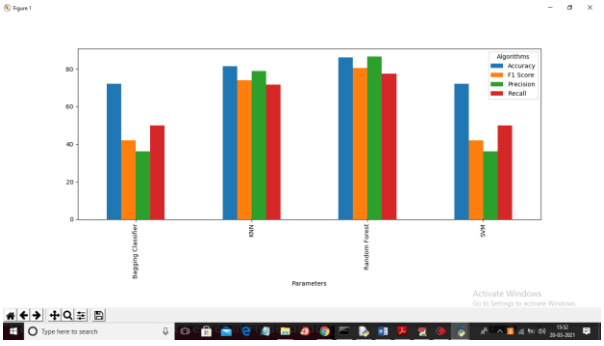
In above screen we can see all codes converted into vector where all words in codes will put as column header and the count of each word and its average values will put in rows and now vector is ready and now click on 'Calculate Local Outlier Factor' button to remove irrelevant columns/attributes



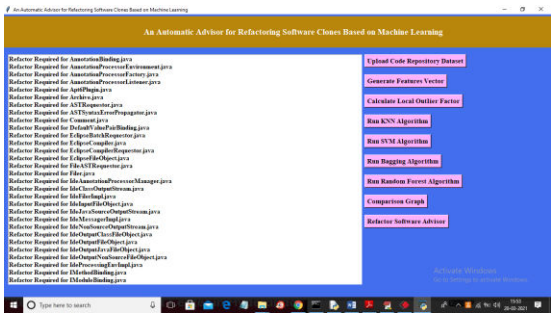
In above screen where ever we are seeing 1 that column in feature vector is important and where we are seeing -1 that column contains irrelevant attributes and now click on 'Run KNN Algorithm' and all other algorithm button to build machine learning model and then will get its prediction accuracy



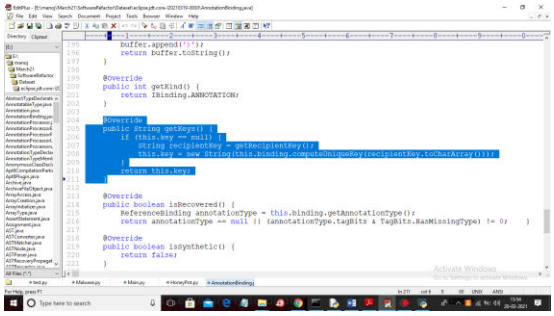
In above screen each algorithm evaluated on dataset and then we got above performance values such as accuracy, precision, recall and FMeasure and now click on 'Comparison Graph' button to get below graph



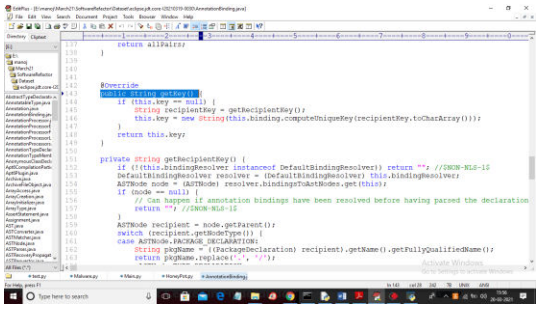
In above graph we can see performance of each algorithm and in all algorithm random forest giving better result and now machine learning models are ready and now click on 'Refactor Software Advisor' button to get all code names which require refactor



In above screen we got all class names which require refactor and now open first file called AnnotationBinding.java and see is there any duplicate code



In above screen we can see in same program two functions are there with same code and different name as getKeys() in selected text and in next screen we have another method as getKey()



getKeys() and getKeys() contains duplicate code so refactor require

VII. CONCLUSION

This paper suggests a learning method which automatically extracts features from the detected code clones and trains the models to advise the developers in regard to what a clone needs to be refactored and what is its type. We introduce a new method of converting clone type outliers into an Unknown clone to improve classification results. We present an extensive comparative study and perform an evaluation of the efficacy

of our suggested idea by using state-of-the- art classification models. • We present a new machine learning framework that automatically extracts features from the detected code clones and trains models to advise the developers on the type of refactored clone code and those which are not refactored. • We explore a new method by which to clone types of outliers into an Unknown clone from the training categories, which significantly improves the classification results. • We present an extensive comparative study and an evaluation of the efficacy of our suggested idea by using stateof-the-art classification models. We used four classification models to obtain their relative performance. The experimental results suggest that our approach has high value in achieving high automated advising refactored clone accuracy. In future, we would like to increase the scope of work to achieve additional improvements, for example, by using set classification and deep learning.

REFERENCES

[1] Y. Dang, S. Ge, R. Huang, and D. Zhang, ‘‘Code clone detection experience at microsoft,’’ in Proc. 5th Int. Workshop Softw. Clones, 2011, pp. 63–64.

[2] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, ‘‘Automatic clone recommendation for refactoring based on the present and the past,’’ in Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME), Sep. 2018, pp. 115–126.

[3] S. Kodhai and S. Kanmani, ‘‘Method-level code clone modification using refactoring techniques for clone maintenance,’’ Adv. Comput. Int. J., vol. 4, no. 2, pp. 7–26, Mar. 2013.

[4] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, ‘‘An empirical study of code clone genealogies,’’ ACM SIGSOFT Softw. Eng. Notes, vol. 30, no. 5, 2005, pp. 187–196.

[5] N. Göde and R. Koschke, ‘‘Frequency and risks of changes to clones,’’ in Proc. 33rd Int. Conf. Softw. Eng., 2011, pp. 311–320.

[6] W. Wang and M. W. Godfrey, ‘‘Recommending clones for refactoring using design, context, and history,’’ in Proc. IEEE Int. Conf. Softw. Maintenance Evol., Sep. 2014, pp. 331–340.

[7] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, ‘‘Refactoring support based on code clone analysis,’’ in Proc. 135Int. Conf. Product Focused Softw. Process Improvement. Cham, Switzerland: Springer, 2004, pp. 220–233.

[8] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, and K. Words, ‘‘ARIES: Refactoring support environment based on code clone analysis,’’ in Proc. IASTED Conf. Softw. Eng. Appl., 2004, pp. 222–229.

[9] Y. Higo, S. Kusumoto, and K. Inoue, ‘‘A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system,’’ J. Softw. Maintenance Evol. Res. Pract., vol. 20, no. 6, pp. 435–461, Nov. 2008.

[10] M. F. Zibran and C. K. Roy, ‘‘A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring,’’ in Proc. IEEE 11th Int. Work. Conf. Source Code Anal. Manipulation, Sep. 2011, pp. 105–114.

[11] K. Hotta, Y. Higo, and S. Kusumoto, ‘‘Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph,’’ in Proc. 16th Eur. Conf. Softw. Maintenance Reeng., Mar. 2012, pp. 53–62.

[12] R. Tairas and J. Gray, ‘‘Increasing clone maintenance support by unifying clone detection and refactoring activities,’’ Inf. Softw. Technol., vol. 54, no. 12, pp. 1297–1307, Dec. 2012.

[13] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, ‘‘Assessing the refactorability of software clones,’’ IEEE Trans. Softw. Eng., vol. 41, no. 11, pp. 1055–1090, Nov. 2015.