IMPACT OF CODE SMELLS ON SOFTWARE FAULT PREDICTION AT CLASS LEVEL AND METHOD LEVEL

Supervisor

Dr. Tamim Ahmed Khan

Submitted by

Um-E-Safia

01-241191-019

MS (Software Engineering)

A thesis submitted to the Department of Software Engineering, Bahria University, Islamabad in the partial fulfillment for the requirements of a Master's degree in Software Engineering.

April 2021

Approval Sheet

Thesis Completion Certificate

Scholar's name: Um-e-Safia   Registration no: 01-241191-019

Programme of study: MS Software Engineering

Thesis Title: Impact of code smells on software fault prediction at class level and method level

It is to certify that the above student's thesis has been completed to my satisfaction and, to my belief, its standard is appropriate for submission for Evaluation. I have also conducted plagiarism test of this thesis using HEC prescribed software and found similarity index at _12%_ that is within the permissible limit set by the HEC for the MS/MPhil degree thesis. I have also found the thesis in a format recognized by the BU for the MS/MPhil thesis.

Principal Supervisor's Signature: ___________________________

Date: __25-april-2021_____   Name: __Um-e-Safia__________

Certificate of Originality

This is certify that the intellectual contents of the thesis

Impact of code smells on software fault prediction at class level and method level.

are the product of my own research work except, as cited property and accurately in the acknowledgements and references, the material taken from such sources as research journals, books, internet, etc. solely to support, elaborate, compare and extend the earlier work. Further, this work has not been submitted by me previously for any degree, nor it shall be submitted by me in the future for obtaining any degree from this University, or any other university or institution. The incorrectness of this information, if proved at any stage, shall authorities the University to cancel my degree.

Signature: _____ Safia _____ Date: 25-april-2021

Name of the Research Student: Um-e-Safia

// Abstract

The main aim of software fault prediction is the identification of such classes and methods where faults are expected. Fault prediction used properties of the software project to predict faults at the early stage of SDLC. Early-stage prediction of software faults supports software quality assurance activities.

Evaluation of code smells for anticipating software faults is basic to ensure its importance in the field of software quality. In this thesis, we will investigate that how code smells help in software fault prediction at the class level and method level. Previous studies show the impact of code smells on fault prediction. However, using code smells for class level faults prediction and method level fault prediction needs more concern.

We make use of the defects4j repository to create the dataset that we use for training and testing of the software fault prediction model. We use pseudo labeling for class level prediction and bagging for method level prediction. We use accuracy, precision, recall, f1 score, and 10-fold cross-validation method for the evaluation of models.

To do validation, we use a case study. We extract code smells from different classes and methods, and we then make use of these code smells for fault prediction. We compare our prediction results with actual results and see if our prediction is correct.

Dedication

This dissertation is dedicated to my beloved parents and siblings who always supported me and stood by my side.

Acknowledgments

Gratitude to Almighty ALLAH, the Most Merciful and Beneficent, who provided us thinking power and gifted us the most signified and distinctive place among all his creatures. I thank ALLAH, the almighty for giving me the ability and strength to complete this dissertation and for blessing me with many great people who have been my greatest support. I would like to extend my deep gratitude to my supervisor, Dr. Tamim Ahmed khan for his kind supervision, enthusiasm, immense knowledge, and guidance. I am indebted to my parents and my siblings for supporting me spiritually.

Table of Contents

Approval Sheet ii
Certificate of Originality iii
Abstract iv
Dedication v
Acknowledgement vi
Table of content vii
List of Tables x
List of Figures xi
Chapter 1: Introduction 1
1.1 Motivation 1
1.2 Research Gap 2
1.3 Problem statement 3
1.4 Proposed solution 3
1.5 Objectivs 3
1.6 Contribution 3
1.7 Thesis Organization 4
Chapter 2: Literature Review 5
2.1 Code smells 5
2.2 Software fault prediction using software matrics 8
2.3 Software fault prediction using code smells 11
2.4 Software fault prediction at method level 16
2.5 Summary 17
Chapter 3: Research Methodology and Our Approach 18
3.1 Introduction 18
3.2 Our Methodology 18
3.2.1 Preprocessing phase 19
3.2.1.1. Selection 19
3.2.1.2. Preprocess/cleaning……. 20
3.2.1.3. Merge 20
3.2.1.4. Normalization 20
3.2.2 Model development phase 21
3.2.2.1. Test and train split 21

3.2.2.2. ML Model ... 21
3.2.2.2.1 Class level prediction... 21
3.2.2.2.1.1 Pseudo labeling ... 21
3.2.2.2.1.2 Random forest ... 22
3.2.2.2.2 Method level prediction ... 22
3.2.2.2.2.1 Bgging ... 22
3.2.2.3 Evaluation ... 23
3.2.3 Post Processing phase ... 23
3.2.3.1 Code smell extraction ... 23
3.2.3.2 Prediction ... 23
3.2.3.3 Validation... 23
3.3 Summary ... 24
Chapter 4: Implementation ... 25
4.1 Fault prediction at class level ... 25
4.1.1 Preprocessing phase ... 25
4.1.1.1 Selection ... 25
4.1.1.1.1 Primary dataset ... 25
4.1.1.1.2 Secondary dataset ... 25
4.1.1.2 preprocess/cleaning ... 25
4.1.2 Model development phase ... 26
4.1.2.1 Train and test split ... 26
4.1.2.2 ML Model ... 26
4.1.2.2.1 Algorithm ... 26
4.1.2.2.2 Software specification ... 27
4.1.2.2.3 Training classifier ... 27
4.1.2.2.3.1 Pseudo labeling ... 27
4.1.2.2.3.2 Random forest ... 28
4.1.2.3. Evaluation ... 28
4.2 fault prediction at method level ... 28
4.2.1 Preprocessing Phase ... 28
4.2.1.1. Selection ... 28
4.2.1.2. Preprocess/cleaning ... 29
4.2.2 Model development phase ... 29

4.2.2.1. Train and test split 29
4.2.2.2. ML Model 29
4.2.2.2.1 Algorithm 29
4.2.2.2.2 Software specification 30
4.2.2.2.3 Training classifier 30
4.2.2.2.3.1 Bagging 31
4.2.2.2.3.2 Bagging with random forest classifier 31
4.2.3. Evaluation 32
Chapter 5: Results and discussion 33
5.1 Fault prediction at class level 33
5.1.1 Training of model 33
5.1.2 Model evaluation 33
5.1.3 Model Validation 34
5.2 Fault prediction at method level 37
5.2.1 Training of model 37
5.2.2 Model evaluation 39
5.2.3 Model Validation 40
5.3 Comparison 43
Chapter 6 45
Conclusion 45
References 47

List of Tables

Table 1 List of first introduced code smells and definitions(Class level) **5**
Table 2 List of first introduced code smells and definitions(Method level) **5**
Table 3 List of additional code smells and definitions(Class level) **7**
Table 4 List of additional code smells and definitions(Method level) **7**
Table 5 Fault prediction using software metrics 9
Table 6 Fault prediction using code smells 13
Table 7 List of selected code smells 18
Table 8 Labeled dataset (Class) 24
Table 9 Dataset (Method) 28
Table 10 10-folds cross-validation(Class) 34
Table 11 Performance evaluation table (Class) 35
Table 12 Predicted and actual results(class) 36
Table 13 10-folds cross-validation (Method) 39
Table 14 Performance evaluation table (Method) 39
Table 15 Predicted and actual results (Method) 41
Table 16 Comparison 45

List of Figures

Figure 1 Thesis organization 4
Figure 2 Research Process 18
Figure 3 Semi-supervised learning method 20
Figure 4 Pseudo-labeling learning method 21
Figure 5 Fault prediction 22
Figure 6 Pseudo labeling code 27
Figure 7 Random forest algorithm code 27
Figure 8 Values without Bagging code 30
Figure 9 Values with Bagging code 30
Figure 10 Bagging classifier and random classifier code 31
Figure 11 Accuracy Graph (Class) 33
Figure 12 Class prediction error graph (Class) 34
Figure 13 Classification report (Class) 34
Figure 14 Actual and predicted Results (Class) 36
Figure 15 Accuracy Graph (Method) 38
Figure 16 Class prediction classification report (method) 38
Figure 17 Classification report (method) 39
Figure 18 Actual and predicted Results (Class) 41

Chapter 1: Introduction:

Software engineering field have many prediction approaches for example fault prediction, test effort prediction, cost prediction, usability prediction and many others. Every approach has their own importance but, among all these predictions approaches many are in the preliminary phase and required more research to reach robust models. Among all these prediction approaches, Software fault prediction is the most popular research area.

Fault prediction models are used to improve software quality and to assist software inspection by locating possible faults. Software fault is a condition that makes a system come up short in performing out its necessary function.[1] Fault is a basic explanation behind system breakdown and is equivalent to the generally used term bug. Efforts are necessary to minimize software faults. However, all these efforts cost time and resources. Early fault prediction strategy is required so that it helps in the reduction of faults and improves the overall quality of software. It is verified that the sooner a fault is detected lesser it costs [2]. So, early-stage software prediction can save many resources (time, money, human).

Code smells are defined as properties of source code that indicate expected faults or deeper problems [3]. At first 21 different types of code, smells were introduced [4] shown in table 1 and 2. Code smells are now an accepted concept that is used to refer to such design aspects and patterns, which may cause problems at the later stage of software systems like development and maintenance. [4, 5]. Regardless, code smells are not incorrect but instead, their essence point towards instability in design, which fails in the system and expected bugs in the future. The primary focus of empirical evaluation of code smell is to see whether the code with smells is more expected to have faults than the code without smells.

Despite the accepted importance of code smells, Fault prediction using code smells is a under rated field as compared to object-oriented metrics. Different researchers use only a few kinds of code smells and use different methods. Some used regression and some used machine leaning techniques. Some fault prediction models have proposed, but mostly give inadequate information. There is a need of fault prediction model which uses class and method-level code smells and predicts faults not only at class level but also at method level.

1.1 Motivation:

Software defect prediction is a key process in software engineering to improve the quality and assurance of software in less time and minimum cost. It is implemented before the testing phase of the software development life cycle. Software fault prediction models provide defects. Software development is becoming an emerging field with the enhancement of applications used in day-to-day life and it

is increasing interaction with technology at a rapid pace. This vast usage of applications enhances the importance of Quality Assurance. Bugs found after production release can be very costly. Limiting the number of bugs in software is an exertion key to software engineering, faulty code neglects to satisfy the reason it was composed for, lastly fixing it costs time and money. resources in a software development life cycle are quite often restricted and thusly ought to be allocated to where they required most to avoid bugs, they ought to center around the most fault-prone areas of the project. Having the option to predict where such areas may be would permit a greater turn of events and testing endeavors to be allocated to the correct places.

1.2 Research Gap:

Most of the existing work focused on fault prediction using Object-oriented or process metrics. Metrics that are used in these models are commonly "Halstead's software metrics"[6], "McCabe Cyclomatic complexity metrics" [7], and "object-oriented metrics" [8-11]. And different modeling techniques used to predict faults. Statistical modeling technique (univariate or multivariate logistic regression) [12]. And machine learning techniques [13-15] are used for software fault prediction.

Conventional bug prediction approaches that use above mention metrics for prediction have certain issues. For instance, it is more than obvious that if a class or codebase has an enormous line of codes, it is more error prone. Yet this is not proved that a class or method having less line of codes has a smaller number of bugs. In this manner, some other metrics for fault prediction should use in research.

Some previous studies showed a significant effect of code smells on fault prediction. In literature, different types of code smell metrics have been proposed and used for fault prediction models. These metrics help in the construction of the prediction model. Fault prediction models used data gathered from such projects where faults have been identified previously [16].

However, most of those approaches predict bugs on class level or file level. This approach often put a considerable amount of effort on tester's shoulder to examine all classes or file until the bug is located. This specific problem is reinforced by the fact that large files are typically predicted as the most bug prone.

In previous studies, researchers predict faults at a class level, no work is done for faults prediction at the method level using code smells. Using fault prediction models at the level of individual methods rather than at class-level/file-level can save effort in term of time and cost. Moreover, this approach increases the granularity of the prediction and thus reduce manual inspection efforts for developers.

In previous work where code smells were used for software fault prediction, the researcher creates their datasets by using CK and some other object-oriented and process metrics for code smells presence. Moreover, researchers have worked only

with 4-8 types of code smells for fault prediction. However, fault prediction using code smells considering more types, needs more concern and we intend to find out if code smell would be beneficial or not.

1.3 Problem statement:

Code smells for fault prediction is a under rated field as compare to other object-oriented metrics. We note that not all code smells found are faults but some of the code smells can become a fault at the later stage of development. Code smells in software fault prediction require an exclusive evaluation and validation which assist with analyzing the effect of code smells on software fault prediction. In addition to this using method level code smells to predict faults at the method level is also requires.

1.4 Research question:

Q1- How can we make use of code smells to detect faults at class level and method level?

Q2- How can we make use of machine learning algorithm to detect faults?

Q3. What is the prediction model performance at the class and method level using code smells?

1.5 Objective:

Objective of this research is to propose a fault prediction model using code smells that help in predict faulty classes and faulty methods to make testing efficient. With the help of predicted faulty modules using code smells at the early stage of SDLC, we reduce the prior mentioned cost of the project and we can schedule our projects more accurately.

The proposed model:

- Identify the faulty classes present in different applications.
- Identify the faulty methods present in faulty classes of applications.

1.6 Contribution:

The overall contribution of work in this dissertation is to propose a methodology to predict faults at the class level and method level. While working on supervised machine learning we tried random forest model to find the solution to our problem. Also, evaluate the solution that is proposed. To validate the proposed solution with the help of a case study of around 30 classes and methods of the open-source project. We have

labeled and unlabeled dataset of code smells at class level and labeled dataset of code smells at method level. We have trained our model on that dataset and tested it to give predictions. We have trained our model via Random Forest classifier. During the training phase of the model label data set used. During the validation phase of the model, we have passed only code smells and the model automatically predicts that which class and method is faulty.

1.7 Thesis Organization:

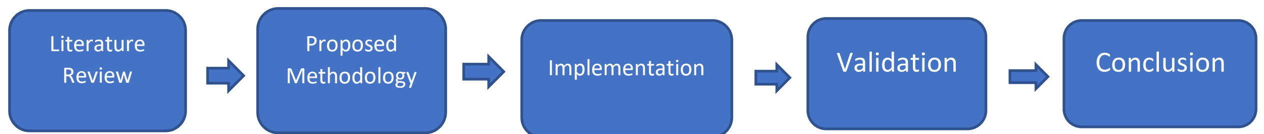

Figure 1: Thesis Organization

Chapter 2: This chapter presents a referential literature review that has already been done by researchers.

The first section, explains what code smells are and provide a definition and description of all known code smells. The second section, explains the work that has been done by authors in the field of software fault prediction. The third section describes software fault prediction using code smells. The last section is about the work done in the field of method-level fault prediction.

Chapter 3: This chapter gives an overall description of the proposed methodology, its validation, and the whole research process. Each step is explained in detail.

Chapter 4: This chapter of the thesis describes the complete implementation details. In this chapter, we explained the implementation of the proposed methodology with the help of a case study. We also present the results of our experiment in this chapter.

Chapter 5: This chapter describes the results and discussion of our work. We also present a comparison with existing studies in this chapter.

Chapter 6: The last chapter concludes our research and represents future work on this research.

Chapter 2: Literature Review

In recent years, software fault prediction is the most arising research field. With the increase in the field of software development and demand for software products, quality concerns also increasing day by day. Early-stage prediction of faults is one the most important quality aspect. With the help of this, we not only improve the product quality, but we also save time and cost. Many researchers have work in this field and use many different techniques, most common techniques are following: Logistic Regression [17-19], Naïve Bayes [20, 21], Support Vector Machines [22], K-Nearest Neighbors [23], Decision Tree [24], Random Forest [25], Linear or Multiple Regression[26, 27], Neural Networks[28, 29], HySOM [30].

2.1 Code smells:

Code smells are properties of source code. They are not bugs but they may cause bugs at later stage of integration. Code smells are an infraction of basic coding standards that decrease the quality of code [31]. A software having code smells still works, it would still give output, but it increases the chances of faults in future. Moreover, it may increase the processing time of software.

From above it is indicating that code smells may cause deeper problems, but they are quick to spot. The best smell is something easy to find but will lead to an interesting problem, like classes with data and no behavior. Code smells can be easily detected with the help of tools.

At first, 21 different types of code smells were introduced shown in Table 1 and 2. Later on, different researchers introduced different new code smells shown in Table 3 and 4.

Table 1: List of first introduced code smells and definitions (Class level)

Sr no.	Code smell	Definition
1	Duplicated code	A similar code structure is in the program in more than one spot.[4]
2	Large class	A class with a lot of functionality and having many instance variables. [4]
3	Divergent change	One class is regularly changed in various manners for various reasons.[4]

4	Shotgun surgery	It is code duplication and refers to when a single change is made in multiple classes.[4]
5	Data clumps	Same data items always together in multiple classes. [4]
6	Primitive obsession	When the code depends a lot on primitives that it starts controlling the logic in a class.[4]
7	Switch Statements	Using switch statements with a type of code to get different behavior or data instead of using subclasses and polymorphism.[4]
8	Parallel Inheritance Hierarchies	Parallel creation of subclasses of super classes.[4]
9	Lazy Class	A class that isn't doing what's necessary.[4]
10	Speculative Generality	Code that writes to handle special cases that are not required.[4]
11	Temporary Field	When many inputs are required by an algorithm temporary field came into use.[4]
12	Message Chains	To fulfill the client's request each object start calling another object. [4]
13	Middleman	A class that starts behaving like a delegate and doing nothing. [4]
14	Inappropriate Intimacy	One unique class utilizes the inner fields and functions for another unique class.[4]
15	Alternative Classes with Different Interfaces	The same functionality is performed by 2 classes with different method names. [4]
16	Incomplete Library Class	When libraries stop meeting user needs.[4]
17	Data Class	A class that only contains fields, getters, and setters. They are just dumb data holders.[4]

18	Refused Bequest	Refer to when child class uses only a few methods inhered from the parent class.[4]

Table 2: List of first introduced code smells and definitions (Method level)

Sr no.	Code smell	Definition
1	Feature envy	It refers to a method that accesses data of other objects more than its data.[4]
2	Long Parameter List	Any method in a class having more than 3 parameters.[4]
3	Long method	Method having many parameters and temporary variables.[4]

Table 3: List of additional code smells and definition (Class level)

Sr no.	Code smell	Definition
1	God class	It refers to such a class that performs too much functionality and has a huge number of lines of code.[32]
2	Tradition breaker	Subclass should provide methods and functionality that are not related to its superclass. [32]
3	Schizophrenic Class	When 2 or more key abstractions are captured in a single class.[32]
4	Brain class	A class that performs too much but have strong cohesion. [32]
5	Anti-singleton	A class that gives mutable variables, which thusly could be utilized as global variables.[16]
6	Blob class	When from 2 coupled classes one class is doing too much and more than the other class.[16]

7	Complex Class	One method of a class having a high value of cyclomatic complexity and LOCs. [16]
8	Swiss Army Knife	It refers to such a class that provides a huge number of interfaces and uses it. [16]
9	Interface Segregation Principal Violation	Interfaces and abstraction should not be forced. [33]
10	Cyclic Dependencies	When 2 or more modules required each other to perform the proper function. [33]
11	Distorted Hierarchy	Such inheritance hierarchy is restricted and deep. [33]

Table 4: List of additional code smells and definition (Method level)

Sr no.	Code smell	Definition
1	Blob Operation	It is a complex and huge operation that centralizes class functionality.[33]
2	God method	A method that performs too much functionality and provides full class functionality in a single method.[33]
3	Extensive coupling	A method that communicates too much with other methods, but provider method dispersed in many classes.[32]
4	Intensive coupling	A method of a class binds with other methods, but the provider method is only dispersed in few classes. [32]
5	Brain method	A method that performs too much but has strong cohesion.[32]

2.2 Software fault prediction using software metrics:

Researchers explored that Software fault prediction utilizing the software metrics is the most usually utilized and most ideal approach to foresee faults. Software metrics

significant to quantify the nature of the product item as far as different factors, for example, coupling, cohesion, reliability, accuracy, completeness, complexity, inheritance etc.[34]. Commonly used metric set are CK metrics[8], McCabe metrics [7], Halstead metrics [6]. Data sets from PROMISE repository [35] and NASA are mostly used in this research area.

In a study, the author picked promise repository and use machine learning techniques for prediction. He used 5 different machine learning models. The performance of all these 5 models was evaluated using accuracy and F-mean.[32].

In another study author used 3 different techniques he first trained data using artificial neural network ANN. Aftereffects of this methodology are contrasted and ANN without pre-training and support vector machines. The results of these experiments showed that all 3 methods can be used for different datasets. [37]. Another proposed a novel way to deal with anticipate faulty classes. In this examination, HyGRAR strategy is carried out. HyGRAR technique depends on supervised learning. In his experiment, he joins artificial neural network and rule mining techniques that help in the classification of faulty and non -faulty data.

In a study author used ensemble methods for prediction. The author claimed that because of using different methods, the results provided by ensemble methods are better than other methods. Linear and non-linear methods are used. As base learning techniques he used genetic programming and linear regression. For performance evaluation Relative error and accuracy are used [36].

The author proposed fuzzy inference system-based approach for fault prediction. KC1, KC2, KC3 datasets are used. Preprocessing and feature selection in done. McCabe metrics are used in this study. Author claim that prediction that is performed by using expert knowledge is better than simple supervised learning approaches. He compared his method results with naïve based and random forest [37].

In this study, the author has 2 types of data, supervised and unsupervised. He used a semi-supervised technique. he performed prediction on both datasets. After that DFCM clustering approach applied. It works by creating or updating the DFCM membership and finding the cluster center. Cluster centers computed for both subsets of data set (labeled or unlabeled). Cluster centers computed by DFCM clustering. sampling approach random under used to balance the features from both the subsets of dataset. results are evaluated in terms of F-mean and area under the curve and results showed that the DFCM approach provides acceptable results for labeled and unlabeled data [38].

Another study validated the impact of different inheritance metrics on fault prediction. 65 public datasets used. The authors divide the dataset into 2 groups inheritance with CK and inheritance without CK. Artificial neural network were used to build the model. Performance of the model is evaluated by accuracy, precision and

recall. Results of experiments showed that inheritance metrics are effective in predicting software faults [39].

Table 5: Software fault prediction using software metrics

Paper Id	Contribution	Model
[40]	the author used 3 different techniques he first trained data using artificial neural network ANN. Aftereffects of this methodology are contrasted and ANN without pre-training and support vector machines. The results of these experiments showed that all 3 methods can be used for different datasets.	ANN
[36]	In a study author used ensemble methods for prediction. The author claimed that because of using different methods, the results provided by ensemble methods are better than other methods. Linear and non-linear methods are used. As base learning techniques he used genetic programming and linear regression. Relative error and accuracy are used for the performance evaluation of the model.	Ensemble method
[41]	In another study, the author proposed a model for SFP. Boehm's model-based classification acted in this study. The COCOMO model is utilized to classify the projects into various classifications. projects accumulated into 3 classes i.e., embedded, semidetached and organic datasets. KLOC metric used in this investigation. prediction are done in two distinct manners i.e., inside dataset or cross dataset. As assessment measures, TPR,	NNge, DTNB, PART, Conjunctive rules, regression tree, oneR,

	FPR, F-measure, precision, AUC, and accuracy are utilized	C4.5, ripper down rules and JRip classifiers
[37]	The author proposed fuzzy inference system-based approach for fault prediction. McCabe metrics are used in this study. Author claim that prediction that is performed by using expert knowledge is better than simple supervised learning approaches. He compared his method results with naïve based and random forest	Fuzzy inference system (FIS)
[39]	In this study, 65 public datasets were used. The authors divide the dataset into 2 groups inheritance with CK and inheritance without CK. Performance of the model is evaluated by accuracy, precision, and recall. Results of experiments showed that inheritance metrics are effective in predicting software faults.	Artificial neural network (ANN)

2.3 Software fault prediction using code smells:

The work done previously using code smells for faulty prediction showed that code smells affect faults. In literature, many code smells metrics have been proposed and used for the fault prediction model. These metrics help in the construction of the prediction model. Fault prediction models used data gathered from such projects where faults have been identified previously [16]. From some time, fault prediction is an important aspect. In literature, many different types of code smell metrics have been proposed and used for the fault prediction model. These metrics help in the construction of the prediction model. Fault prediction models used data gathered from such projects where faults have been identified previously [16].

Smelly classes changed more frequently as compared to non-smelly classes. This point is evaluated in a study where it explained that in 2 open source projects ("Azureus and Eclipse") smelly classes change frequently as compared to non-smelly classes [42]. In

another study, it is found by investigating open source projects that those methods which had been cloned, change frequently as compared to other methods [43].

In another study, author claim that some code like shotgun surgery, God class, and God method is directly connected with faults. To support his claim, he experimented on 3 releases of the eclipse project [44].

Initially, 22 different types of code smells were introduced [4]. An author study, what results in several code smells can cause together. He explained the relationship among different code smells [45]. Another study describes the Domain Specific tailoring of Code Smell. They think the heuristics of code smell [4] is quite wide, they tailor the heuristics of domain-specific of code smell to make the heuristics to fit the specific domain [46].

Two initial taxonomies were proposed for code smell [43, 44]. 40 different anti-patterns described for the object-oriented system by focusing on implementation and design. 2 famous patterns "blob and spaghetti code" included in these 40 different anti-patterns. This study has an in-depth, Wide view of heuristic, code smells, and anti-patterns, for the academic audience [12].

Recently, more considerations pulled into exploring that how faults and code smells are related to each other. This study proved by giving empirical evidence that code smells are helpful in fault prediction. They used source code metrics and codesmells metrics in their study and used Naïve Bayes, Random Forest, and Logistic Regression techniques. [53].

In a study, the author used code smells and community smells and compare their results for fault prediction. The results showed that community smells improve prediction model performance up to 3% in terms of AUC. While code smells improve prediction model performance up to 17% in terms of AUC. [47]

In another study, the author assesses the benefaction of a proportion of the intensity of code smells. for this, they add code smells to the existing prediction model which used process and product metrics. they compare both existing and new models. As of result of this experiment by adding code smells a predictor, the accuracy of the prediction model increases. [54].

In another study, the author study code smells in web applications. He extracts PHP code smells and uses them in his study. The results of this study show that code smells can help in fault prediction and it helps developers to identify faults and plan projects accordingly. [48]

In another study, the author selects 3 projects BIRT, Aspect J, and SWT. Extract code smells from them and studies how code smells are associated with bugs. His study shows that code smells and fault have a strong correlation. Lazy class, complex class, message chain, and long method have a strong correlation with faults. [49]

The author in another study evaluates class level change-prone prediction power using code smells. they used many machine learning approaches. The result of this experiment indicates positive relation between code smells and class change proneness with a probability superior to 70%. [55].

In this study, the author proposed a model in which he used code smells from literature and designate smells. the author used 97 different real projects. The results of this study showed that the model improves 5% in terms of AUC. He concludes that designated smells are a good addition and they help with code smells in prediction.[50].

Another study looks into that that how code smells and fault prediction are related. To study this relation, they used many techniques like ADTree, Naïve Bayes, Logistic regression, and Multilayer perceptron. Results of this study showed that the combined model improves F-measure up to 20%. [56].

Another author in his study confirms that code smells are directly related to software faults and the performance of fault prediction models. They used naïve Bayes and logistic regression [57].

Another study focused on inspecting tool design for software code smell detection. In their work, they explained that source code split is used to automatically detect code smells. Besides, they portrayed how the code smell idea might be extended to incorporate coding standard conformance. To investigate the feasibility of the given approach they used a case study, developed a prototype tool, and test it on the software system. [51]

In another study, the author picked one metric of code smells and investigate its impact. He used the God class metric and see how God classes can help to improve the quality of software. The result of their study showed that in some cases God class are more susceptible to faults [52]

Industrial system and 6 different code smell used to study the relationship between faults and code smells. It is found because of this study that "Shotgun Surgery" presence identifies with a factually critical higher likelihood of faults [44].

But, another author found in his study that there is no relation between faults and code smells and code smell does not affect the presence of faults [53]. So, in literature, the relationship between faults and code smells has not come to an agreement.

A tool was built that is used to rank code smells according to severity based on 3 standards: "past component modification, important modifiability situations for the system, and importance of the sort of smell" [54]. Another author studied the importance of bad smells, and found bad smell resolution' importance and present ordinarily occurring bad smells [55].

A study surveyed that how bad smells affect whole software and especially software maintainability. It proposed that by examining historical information it is possible to

see how bad smells affect software maintainability. It is presumed that; the quality can assess by code smells. By detecting and visualizing code smell quality can be improved [56]

Another study examines experimentally the connection between class error probability and code smells in the three error severity levels and finishes up his investigation as that anticipated models can work agreeably for predicting the errors when all is said in done. Bad code smells could anticipate the class error likelihood and found that some bad code smells could at present predict class error probability. The result of experiment also proposes that when refactoring a class, not only it helps in improving the quality of architecture, but it also helps in reduces the probability of the cl ass having blunders later [57].

Another study gives observational proof about how code smell metrics (Brain class and God class) contributes towards the quality of software system. They used 3 very common open-source software systems and study the effect of brain class and God class on these systems without standardization regarding the size. The investigation shows that God and Brain Classes have a negative impact estimated as far as change frequency, change size, and number of weighted defects. [58].

Table 6: Software fault prediction using code smells

Paper	Contribution	Model
[59]	This study proved by giving empirical evidence that code smells are helpful in fault prediction. They used source code metrics and code smells metrics in their study	Naive Bayes, Random Forest, Logistic Regression
[60]	Author in another study evaluate class level change prone prediction power using code smells. they used many machine learning approaches. The result of this experiment indicates positive relation between code smells and faults.	Naive Bayes, Logistic Regression, Decision tree
[61]	Author in another study evaluate class level change prone prediction power	Naïve bayes,

	using code smells. The result of this experiment indicates that Code smell can predict class change proneness with a probability superior to 70%.	Multilayer perceptron, LogitBoost, Decision tree
[62]	Another study investigates the effect of code smells of predictions of faults. Results of this study showed that combined model improve F-measure up to 20%	ADTree, Naïve bayes, Logistic regression, Multilayer perceptron
[63]	Another author in his study confirms that code smells are directly related to software faults and performance of fault prediction models.	Naive Bayes Logistic Regression
[16]	The creator discovered devotion linking code smell location and the consequences of the fault prediction. In any case, the creator noticed that utilizing code smell identification results can improve the review of bug prediction.	Multivariable logistic regression model
[64]	Author researched the commitment of code smell force with regards to bug prediction. Results showed that the power in every case decidedly adds to best-in-class prediction models, in any event, when they as of now have superior exhibition	Naïve Bayes Logistic regression Decision tree

2.4 Software fault prediction at method level:

Many researchers work in field of fault prediction, all use different approaches, different techniques and cover different aspects of software development. Such models achieved good prediction performance, guiding developers towards those parts of their system where a large share of bugs can expect. However, most of those approaches predict bugs on file-level. This often leaves developers with a considerable amount of effort to examine all methods of a file until a bug is located. This problem reinforced by the fact that large files are typically predicted as the most bug prone.

In a study author proposed bug prediction models at the level of individual methods. This increases the granularity of the prediction and thus reduces manual inspection efforts for developers. The models are based on change metrics and source code metrics that are typically used in bug prediction. Experiment performed on 21 Java open source (sub) systems. Experiment show that prediction mode ls reach a precision and recall of 84% and 88%, respectively [65].

In this study, author replicate previous research on method level bug prediction on different systems/timespans. Afterwards, they reflect on the evaluation strategy and propose a more realistic one. Key results of this study show that the performance of the method-level bug prediction model is like what previously reported. Even so, when same strategy is applied with more realistic parameters all models show a dramatic drop in performance exhibiting results close to that of a random classifier [66].

To the best of our knowledge, efforts in the field of software fault prediction at the method level used process metrics, change metrics, and object-oriented metrics. No work is done for fault prediction at the method level using code smells. This is a fact that faults predicted at the method level rather than at class level or file level can save time and cost. Moreover, predicting faults at method level reduce inspection effort for developer and testers. Thus, a method that can predict software faults at the method level using code smells is highly desired.

2.5 Summary:

This chapter presented a referential literature review that has already been done by researchers. In the first section, it explained the work that has been done by authors in the field of software fault prediction using software metrics. In the second section, it explained the work in the field of software fault prediction using code smells. In third section it describes about software fault prediction at method level. We have studied the literature in detail and identified different prediction models proposed by researchers.

Chapter 3: Research methodology and our approach

3.1 Introduction:

This chapter presents the research methodology and the research process that has been used for software fault prediction using code smells at the class level and at the method level. Various steps and phases are involving in the research process including data gathering, data pre-processing and cleaning, model training, and testing and evaluation.

The applied methodology consists of 3 parts, Preprocessing, Model development phase, and Postprocessing phases. First, is the preprocessing phase which includes code smells selection, dataset pre-processing/cleaning, dataset merge, and dataset normalization and outlier detection. We select dataset with code smells shown in table 5, this dataset will help in the training model that we will use for software fault prediction. We then, in the 2nd phase, do preprocess and cleaning of dataset. we preprocess and clean datasets to remove unused smells and to identify null values in our dataset. we then do Data normalization and outlier analysis.

Second is the model development phase which includes test/train split, training supervised model, and evaluation. First, we split training data into train and test split. Then, we built a software fault prediction classifier and trained the classifier using the data. Later, we evaluated our classifier using performance evaluation metrics. we employ accuracy, precision, recall, and F1-score [25].

Next is postprocessing phase. This phase includes code smells extraction, prediction, and validation. For code smells extraction, there are many tools and techniques available. We used iPlasma tool to extract code smells. after getting the code smell of the class/method, we predict faulty instances. we input code smells to the classifier and our classifier predicts faulty instances. The last phase is of validation phase, in this phase we have fault information of about 30 classes/methods, and we compare our prediction results with actual results and check how accurately we predict faulty instances.

The research strategy uses to conduct this research is Applied Research that is to resolve a specific problem of the software Industry. We know the testing problems faced by industries. To resolve the problem, we have proposed a software fault prediction model using code smells whose validation is done through the open-source application. We describe our research process below with all the steps involve in the process of research.

3.2 Our methodology:

Our research process comprises 3 steps: preprocess phase, Model development phase, and postprocess phase. Figure 2 demonstrates the whole research process with all the steps taken in between the process.

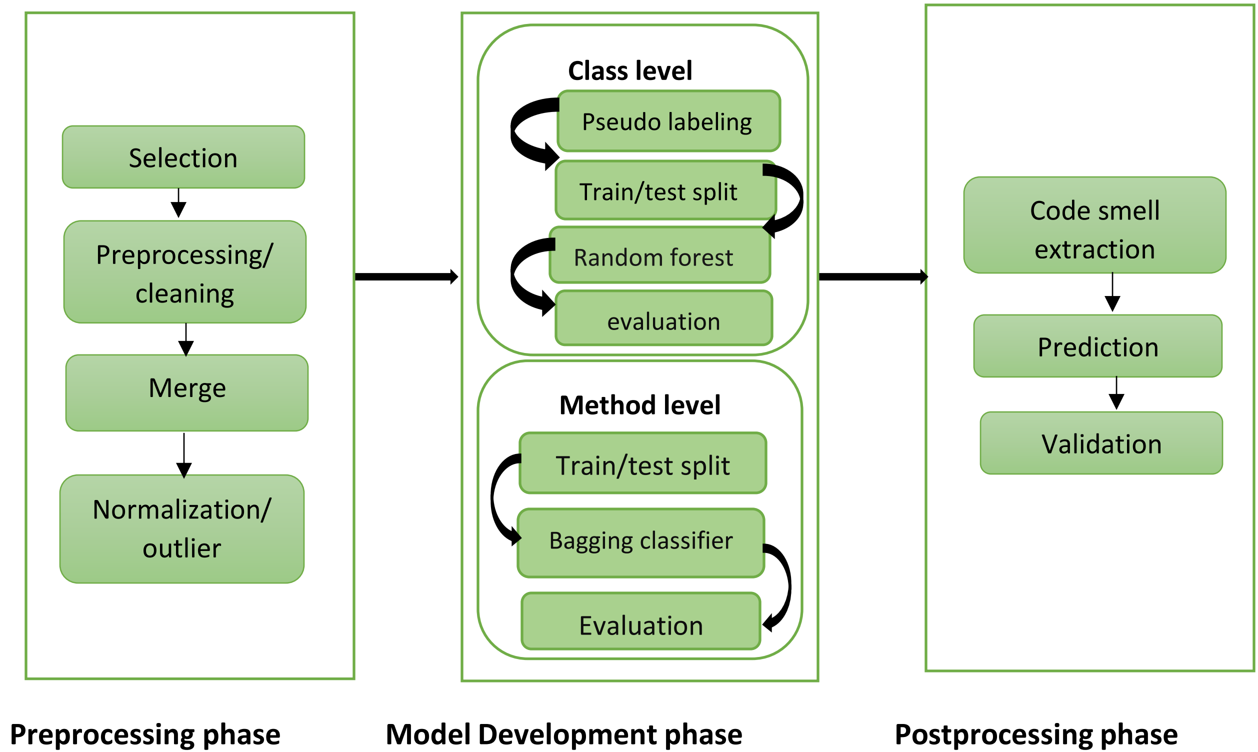

Figure 2: Research process

3.2.1 Preprocessing phase:

3.2.1.1. Selection:

The code smells we are using at class level are God class (GC), Shotgun surgery (SS), Feature envy (FE), Brain class (BC), Tradition breaker (TB), Brain method (BM), Extensive coupling (EC), Parent bequest (PB), Intensive coupling (IC), Long parameter list (LPL), Schizophrenic Class (SC), Data class (DC) and code smells we are using at method level are Feature envy (FE), Brain method (BM), Extensive coupling (EC), Intensive coupling (IC), Shotgun surgery (SS) and Long parameter list (LPL). The criteria of selecting code smells are, these are important and most used basic code smells [59-63], most of the published literature used these code smells [32], tools are available to extract these code smells from the source code of the application and unlabeled dataset of these code smells are publicly available.

3.2.1.2. Pre-processing/cleaning phase:

We do clean operation in our dataset to clean dataset. from dataset we remove unused code smells. We check for any null value in our dataset and null value percentage for each column of dataset.

Table 7: list of selected code smells

No.	Name	Definition
1	God class	It refers to such a class that perform too much functionality and have huge number of lines of code.
2	Shotgun surgery	It is code duplication and refer to when single change made in multiple classes.
3	Feature envy	It refers a method that access data of other object more than data of object.
4	Brain class	A class that performs too much but have strong cohesion
5	Tradition breaker	Sub class should provide methods and functionality that is not related to its super class.
6	Brain method	A method that performs too much but have strong cohesion.
7	Extensive coupling	A method that communicates too much with other methods, but provider method dispersed in many classes
8	Parent bequest	Refer to when child class uses only few methods inhered from parent class.
9	Intensive coupling	A method of a class binds with other methods, but provider method is only dispersed in few classes.
10	Long parameter list	Any method in class having more than 3 parameters.
11	Schizophrenic Class	When 2 or more key abstraction are captured in a single class.
12	Data class	Refer to when child class uses only few methods inhered from parent class

3.2.1.3 Merge:

All datasets have the same number of attributes, so we merge all datasets into one. We use the discrete value of label bug, i.e., 1 and 0. 1 label depict faulty instance whereas, 0 label depict non- faulty instance.

3.2.1.4 Normalization:

At last, we analyze our dataset for the existence of outlier and data normalization. Our dataset doesn't have any outlier and it is in binary form, so we don't need data normalization as well.

3.2.2 Model Development phase:

3.2.2.1 Train and test split:

After getting cleaned and combined dataset we perform train and test split. We split our dataset into the ratio of 70:30, 70% training data, and 30% testing data. we use train split to train our classifier and test split is used to test our classifier.

3.2.2.2 ML Model:

We propose a fault prediction model for early-stage fault prediction at the class and method level. We use dataset of code smells for the training classifier. We use supervised and semi supervised machine learning approach to build and train the classifier. After training classifier, we use performance evaluation metrics to evaluate the performance of our classifier.

3.2.2.2.1 Class level prediction:

We have less quantity of labeled and an enormous quantity of unlabeled dataset available. To use both types of datasets in training classifiers we use a semi-supervised machine learning technique named pseudo-labeling [67].

3.2.2.2.1.1 Pseudo labeling:

Pseudo Labeling is an effortless and proficient strategy to do semi-supervised learning.

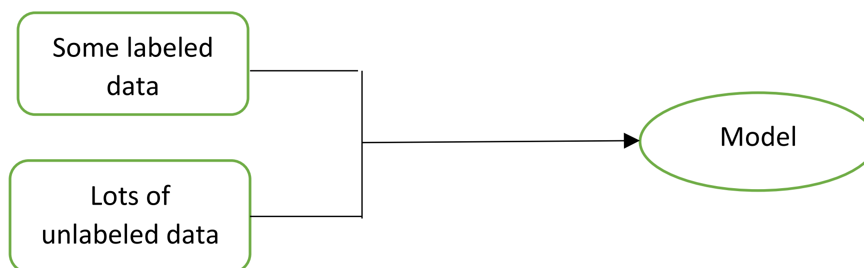

Figure 3: Semi-supervised learning method

Pseudo labeling technique uses less amount of labeled dataset and big amount of unlabeled dataset and improves the model's exhibition. It first uses labeled data to train the model, 2nd it predicts pseudo labels for unlabeled data, third it merges both labeled and pseudo labeled data into one, and 4th it retrains the model with merged data.

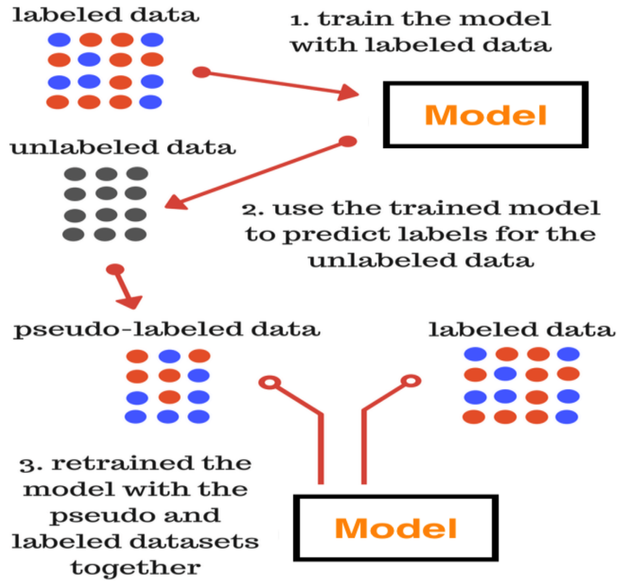

Figure 4: Pseudo-labeling learning method

3.2.2.2.1.2 Random forest:

We use the random forest machine learning technique to train our classifier. We have 2 class problem and it is supposed that SVM performed well with 2 class problem but in our case, we have imbalance data, and SVM performed poorly due to class imbalance [68]. Previous studies showed that the random forest technique is more powerful for fault prediction as compared to SVM or other techniques [69, 70]. Random forest is a sort of supervised machine learning calculation dependent on ensemble learning. In ensemble learning, sometimes we join different algorithms or sometimes we join the same algorithm at different times to improve accuracy. The combined dataset is used for training and testing random forest classifier. Accuracy, and performance evaluation metrics used to evaluate classifier.

3.2.2.2.2 Method level prediction:

We extract method-level dataset from defects4j. this dataset is in a small amount so, to increase the amount of dataset we use a technique named Bagging. Bagging not only increases the amount of data but also improves accuracy, loss, bias, and variance and improves the performance of the classifier.

3.2.2.2.2.1 Bagging:

Bagging classifier is a technique that divides dataset into subsets and then fits the base classifier on each subset of dataset. after that, through voting or the average method it aggregates the results and gives a final prediction. Bagging not only increase the amount of data but also improves accuracy, loss, bias, and variance and improves the performance of classifier.

As a base classifier, we are using random forest tree. The number of bags we are using is 10. each bag has 500 data points and the number of features we are using is 6.

3.2.2.3 Evaluation:

To evaluate classification performed by the model we use performance evaluation metrics. We pick Accuracy, Precision, Recall, and F1-score for the appraisal of the models. In the prediction model, the positive class is the damaged class, and the negative class is the non-flawed class. Precision is characterized as the closeness of estimated values with real worth. Here it addresses the quantity of the all-out right expectations to an absolute number of inaccurate and right predictions. precision estimates the number of positive class expectations that have a place with the positive class. The recall is the extent of accurately grouped flawed occasions to every one of the real cases that are deficiency inclined. Also, F1 score gives a solitary score that adjusts both the worry of precision and recall in one worth. It is the consonant mean of precision and recall.

3.2.3 PostProcessing Phase:

3.2.3.1 Code smells extraction:

We extract our selected code smells from the source code. There are many tools available for code smell extraction. Some tools are limited in the number of code smells, so we select and use such a tool that is easily available and extract all selected code smells.

3.2.3.2 Prediction:

The above phase extracted code smells then input them to the classifier so that our classifier predicts faulty instances, as shown in figure 5. Our classifier predicts which class and method are more expected to have faults.

3.2.3.3. Validation:

We have fault information of 30 classes and methods. In the validation stage, we compare these actual results with our classifier prediction and see how accurately our classifier is predicting faulty instances.

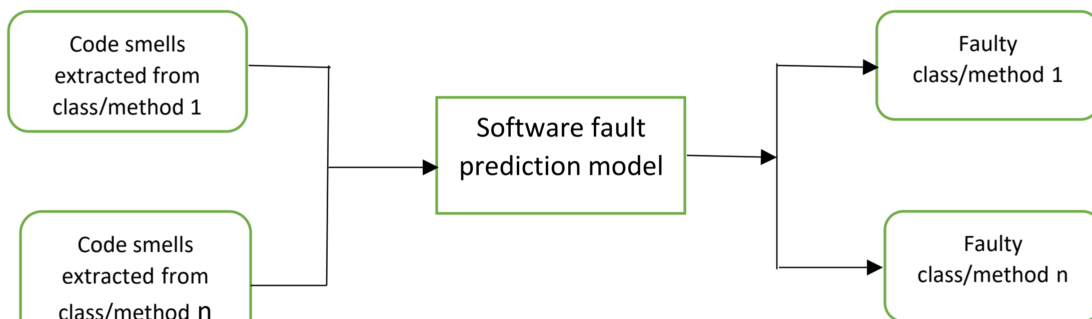

Figure 5: fault prediction

3.3 Summary:

This chapter provided overall description of the proposed methodology. It is divided into 3 main phases which have further phase. The preprocess phase includes selection, preprocessing/cleaning, merge, and normalization. The process phase includes train and test split, supervised model, and evaluation. And the last post process phase includes code smells extraction, prediction, and validation.

Chapter 4: Implementation:

This chapter describes the steps involved in preprocessing, model development and postprocessing phases.

4.1 Fault prediction at class level:

4.1.1 Preprocessing phase:

The preprocessing phase for software fault prediction at class level model comprises various steps including selection, pre-processing/cleaning, and normalization.

4.1.1.1. Selection:

Faulty classes data for the development of fault prediction model collected from two different sources.

4.1.1.1.1 Primary dataset:

The primary dataset is the labeled dataset. This labeled dataset is extracted from projects available at defects4j [3]. defects4j is an open-source repository which has complete detail of active bugs of multiple java projects. It provides complete information of faults which includes the package name, class name, method name, and exact location of fault [71]. Table 8 shows the description of selected datasets. we have complete details of faults now for code smells we use the tool iPlasma [72]. iPlasma is an environment that helps in the extraction of code smells from source code with addition to this, the object-oriented system used it for quality analysis. After getting code smells of all projects as shown in table 8, the reports of code smells are generated in comma separated version (CSV) extension. we add labels e.g., 0 or 1. 0 if no faults and 1 if the fault is present in that class.

Table 8: labeled dataset

Sr no.	Identifier	Project name	Instances	Faults
1	Chart	Jfreechart	61	12
2	Mockito	Mockito	90	26
3	Lang	commons-lang	130	43
4	Closure	closure-compiler	184	92
5	Cli	commons-cli	85	18
6	Math	commons-math	124	68
7	Jsoup	Jsoup	81	37

4.1.1.1.2 Secondary dataset:

The secondary dataset is publicly available unlabeled dataset of code smells [73]. This dataset comes from Qualitas Corpus (QC). Code smells of 76 different systems are in the corpus.

4.1.1.2. Preprocessing/cleaning:

We remove unused code smells from our target sets. We check for any null value in our dataset and null value percentage for each column of the dataset.

4.1.2 Model Development phase:

The process phase for software fault prediction at class level model comprises of various steps including train and test split, model, and evaluation.

4.1.2.1 Train and test split:

We split our dataset into the ratio of 70:30, 70% training data, and 30% testing data. we use train split to train our classifier and test split to test our classifier.

4.1.2.2 ML Model:

Both primary and secondary datasets are used to design and train the classifier.

4.1.2.2.1 Algorithm:

We present our algorithm for training classifier in algorithm 1. We first select unlabeled datasets containing code smells ($M= M_1 \ldots M_m$). we merge datasets into one and removed unused code smells. after that, we select a labeled dataset containing code smells. we merge datasets into one. Then we train the algorithm using labeled dataset and predict pseudo labels for unlabeled dataset. after predicting pseudo labels, we concatenate labeled and unlabeled datasets, split combined data into 70% train and 30% test data, and retrain the model. Finally, we use accuracy, precision, recall, and F1 score to evaluate our classifier.

Algorithm 1: Algorithm for Training Classifier

Input: A set of labeled and unlabeled datasets, containing a set of code smells ($M= M_1 \ldots M_m$).

Output: software fault prediction model with overall classifier (C_f) evaluation.

Start:

1. Select unlabeled datasets ($D_{ul1}, D_{ul2}, \ldots, D_{uln}$) //all containing set of selected code smells ($M=M_1 \ldots, M_m$)
2. Combine datasets into one.

$$D_{UL}= \sum_{k=1}^{n}(D_{ulk})$$

3. Remove unused code smells M_1, M_2, ….., $M_m \rightarrow M_c$
4. $D_{UL} \leftarrow$ apply data cleaning (D_{UL})
5. Gather labeled datasets (D_{l1}, D_{l2}, ….., D_{ln}) //all containing set of selected code smells ($M=M_1$….., M_M)
6. Combine datasets into one.

$$D_L = \sum_{k=1}^{n} (D_{lk})$$

7. $D_L \leftarrow$ apply data cleaning (D_L)
8. Train algo using labeled dataset D_L.
9. Predict pseudo labels for unlabeled datasets D_{UL}.
10. Combine both datasets

$$D = D_L + D_{UL}$$

11. Spilt the combined Dataset (D) into train and test spilt (70:30).
12. Train and test the random forest classifier (C_f) on cleaned combined dataset (D) with code smells Mc.
13. Calculate precision, recall, and f1 score.

END

4.1.2.2.2 Software specification:

For the development of the fault prediction model at the class level we have used the Jupyter notebook. Python language provides a great platform for building machine learning algorithm which is very concise and easy to understand. It provides several libraries for data analysis, visualization, text classification, and natural language processing. Python library NumPy is used for data storage at runtime array. We have used Panda library to read data from csv file (code smells dataset). Pandas is a general information control library based on the top of NumPy. Scikit-learn is based on top of two Python libraries NumPy and SciPy and has become the most mainstream Python ML library for creating ML calculations. Scikit-learn has a wide scope of regulated and solo learning calculations that deal with a reliable interface in Python. The library can likewise use for information mining and information investigation. The principal machine learning works that the Scikit-learn library can deal with are classification, relapse, grouping, dimensionality decrease, model choice, and preprocessing.

4.1.2.2.3 Training classifier:

We have small amount of labeled (primary) and huge amount of unlabeled (secondary) dataset is available. To use both types of datasets in the training classifier we use a semi-supervised machine learning technique named pseudo-labeling [67].

4.1.2.2.3.1 Pseudo labeling:

For pseudo labels, we first train the model using labeled dataset, then predict pseudo labels for the unlabeled dataset. after that concatenate both label and pseudo label data.

```
Y_train = train['Fault'].values
X_train = train[list(features)].values
X_test = test[list(features)].values
```

```
model1 = RandomForestClassifier(n_estimators=10)
model1.fit(X_train, Y_train)
pseudoY_test = model1.predict(X_test)
```

```
data = np.vstack((X_train, X_test))
labels = np.concatenate((Y_train, pseudoY_test), axis=0)
```

Figure 6: pseudo labeling code

4.1.2.2.3.2 Random forest:

After concatenating both label and pseudo label data, we split dataset into 70:30 and training model using Random forest algorithm.

```
pseudo_model = RandomForestClassifier(n_estimators=10)
pseudo_model.fit(X_train, Y_train)
```

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=10,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

Figure 7: Random forest algorithm code

4.1.2.3. Evaluation:

For evaluation of our model, we use performance evaluation metrics which include Accuracy, Precision, Recall, and F1-score for the assessment of the models. With performance metrics, for completely unbiased results we have use the 10-folds cross-validation method.

4.2 Fault prediction at method level:

4.2.1 Preprocessing phase:

The preprocessing phase for software fault prediction at the method level model comprises various steps including selection, pre-processing/cleaning, and normalization.

4.2.1.1. Selection:

The labeled dataset is extracted from projects available at defects4j [3]. defects4j is an open-source repository which has complete detail of active bugs of multiple java projects. It provides complete information of faults which includes the package name, class name, method name, and exact location of fault [71]. Table 9 shows the description of selected datasets. we have complete details of faults now for code smells we use the tool iPlasma [72]. iPlasma is an environment that helps in the extraction of code smells from source code with addition to this, the object-oriented system used it for quality analysis. After getting code smells of all projects shown in table 9, the reports of code smells are generated in comma-separated version (CSV) extension. we add labels e.g., 0 or 1. 0 if no faults and 1 if the fault is present in that class.

Table 9: Selected Dataset

Sr no.	Identifier	Project name	Instances	Faults
1	Chart	Jfreechart	61	12
2	Mockito	Mockito	90	26
3	Lang	Commons-lang	130	43
4	Closure	closure-compiler	184	92
5	Cli	Commons-cli	85	18
6	Math	commons-math	124	68
7	Jsoup	Jsoup	81	37
8	Compress	Compress	53	20

4.2.1.2. Preprocessing/cleaning:

We check for any null value in our dataset and null value percentage for each column of dataset.

4.2.2 Model Development phase:

The process phase for software fault prediction at the method level model comprises various steps including train and test split, model, and evaluation.

4.2.2.1 Train and test split:

We split our dataset into the ratio of 70:30, 70% training data, and 30% testing data. we use train split to train our classifier and test split to test our classifier.

4.2.2.2 ML Model:

The labeled dataset is used to design and train classifier.

4.2.2.2.1 Algorithm:

We present our algorithm for training classifier in algorithm 2. We first select labeled datasets containing code smells (M= M_1…. M_m). we merge datasets into one and removed unused code smells. after merge we split combined data into 70% train and 30% test data, and train model using classifier. Finally, we use accuracy, precision, recall and F1 score to evaluate our classifier.

Algorithm 2: Algorithm for Training Classifier

Input: A set of labeled datasets (D_1, D_2, D_3…., D_n), containing code smells (M= M_1…. M_m).

Output: software fault prediction model with overall classifier (C_f) evaluation.

Start:

1. Select labeled datasets (D_1, D_2, D_3….,Dn) // all containing code smells (M= M_1….M_m).
2. Combine datasets into one.
   $D= \sum_{k=1}^{n}(D_k)$
3. D ← apply data cleaning (D)
4. Spilt the combined Dataset (D) into train and test spilt (70:30).
5. Apply Bagging classifier (Cf) with random forest as base classifier on the cleaned dataset.
6. Calculate precision, recall, and f1 score.

END

4.2.2.2.2 Software specification:

For the development of the fault prediction model at the method level we have used the Jupyter notebook. Python language provides a great platform for building machine learning algorithm which is very concise and easy to understand. It provides several libraries for data analysis, visualization, text classification, and natural language processing. Python library NumPy is used for data storage at runtime array. We have used Panda library to read data from csv file (code smells dataset). Pandas is a general information control library based on the top of NumPy. Scikit-learn is based on top of two Python libraries NumPy and SciPy and has become the most mainstream Python ML library for creating ML calculations. Scikit-learn has a wide scope of regulated and solo learning calculations that deal with a reliable interface in Python. The library can likewise use for information mining and information investigation. The principal machine learning works that the Scikit-learn library can deal with are classification, relapse, grouping, dimensionality decrease, model choice, and preprocessing.

4.2.2.2.3 Training classifier:

At the method level, dataset of code smells is not available publicly so we extract the method level dataset. this dataset is in a small amount so, to increase the amount of dataset we use a technique named Bagging. Bagging not only increases the amount of data but also improves accuracy, loss, bias, and variance and improves the performance of the classifier.

4.2.2.2.3.1 Bagging:

1. Value of loss, bias, and variance without bagging:

```
avg_expected_loss, avg_bias, avg_var = bias_variance_decomp(
        rfc_model, X_train, Y_train, X_test, Y_test,
        loss='0-1_loss')
```

```
print('Average expected loss: %.3f' % avg_expected_loss)
print('Average bias: %.3f' % avg_bias)
print('Average variance: %.3f' % avg_var)
```

```
Average expected loss: 0.143
Average bias: 0.148
Average variance: 0.012
```

Figure 8: Values without Bagging code

2. Value of loss, bias, and variance with bagging.

```
avg_expected_loss_bagg, avg_bias_bagg, avg_var_bagg = bias_variance_decomp(
        bagg_model, X_train, Y_train, X_test, Y_test,
        loss='0-1_loss')
```

```
print('Average expected loss: %.3f' % avg_expected_loss_bagg)
print('Average bias: %.3f' % avg_bias_bagg)
print('Average variance: %.3f' % avg_var_bagg)
```

```
Average expected loss: 0.142
Average bias: 0.136
Average variance: 0.012
```

Figure 9: Values with Bagging code

4.2.2.2.3.2 Bagging classifier and Random forest:

We are using random forest as the base classifier for bagging. We create 10 bags of dataset, each with 500 data points.

4.2.2.3 Evaluation:

For evaluation of our model, we use performance evaluation metrics which include Accuracy, Precision, Recall, and F1-score for the assessment of the models. With performance metrics, for neutral results, we use 10-fold cross validation method.

```
method_model = BaggingClassifier(RandomForestClassifier(n_estimators=20), n_estimators=10, max_features=6, max_samples=500)
```

```
method_model.fit(X_train, Y_train)
```

```
BaggingClassifier(base_estimator=RandomForestClassifier(bootstrap=True,
                                                        ccp_alpha=0.0,
                                                        class_weight=None,
                                                        criterion='gini',
                                                        max_depth=None,
                                                        max_features='auto',
                                                        max_leaf_nodes=None,
                                                        max_samples=None,
                                                        min_impurity_decrease=0.0,
                                                        min_impurity_split=None,
                                                        min_samples_leaf=1,
                                                        min_samples_split=2,
                                                        min_weight_fraction_leaf=0.0,
                                                        n_estimators=20,
                                                        n_jobs=None,
                                                        oob_score=False,
                                                        random_state=None,
                                                        verbose=0,
                                                        warm_start=False),
                  bootstrap=True, bootstrap_features=False, max_features=6,
                  max_samples=500, n_estimators=10, n_jobs=None,
                  oob_score=False, random_state=None, verbose=0,
                  warm_start=False)
```

Figure 10: Bagging classifier and random classifier code

Chapter 5: Result and Discussion:

5.1 Fault prediction at class level:

To demonstrate the validity of our proposed model, we took a project Joda-time from defect4j. Joda-time provides replacement for java date and time class. It allows multiple calendars systems. Joda-Time solves one critical problem stale time zone data. [74]. There is a total of 145 different classes. We extract code smells from the project using a tool named IPlasma. We extract code smells of 30 classes. From 145 classes we picked only business logic classes. other classes that are platform depended are not considered. As per our proposed model we are interested to predict the faulty class through a trained Random forest classifier.

5.1.1 Training of Model

We have 2 datasets that we used to train our model primary dataset which is labeled and a secondary dataset which is unlabeled. we use a semi-supervised machine learning approach pseudo-labeling to label unlabeled dataset. Our training dataset has 13 columns. first 12 Columns names are "BrainMethod-IPLASMA", "ExtensiveCoupling-IPLASMA", "IntensiveCoupling-IPLASMA", "SchizoClass-IPLASMA", "BrainClass-IPLASMA", "TraditionBreaker-IPLASMA", "GodClass-AGGREGATE", "FeatureEnvy-AGGREGATE", "DataClass-AGGREGATE", "LongParamList-AGGREGATE", "ShotgunSurgery-AGGREGATE", "RefusedParentBequest-AGGREGATE" which are defined as X_train while 13th column is our labeled column named "fault" which is defined as Y_train. Initially, we train our model on both X_train and Y_train. As we have to do pseudo labeling on an unlabeled dataset so that we can increase our training dataset we have X_test on which we predict labels.

After pseudo-labeling, we concatenate data and labels and create one complete labeled dataset. we split our data into 70 and 30. we are using 70% of the dataset for model training while the rest of 30% is used for testing to ensure the test results of the trained model.

5.1.2 Model evaluation:

We are using random forest classifier. To select the most optimal value for the number of trees we calculate accuracy with the different number of trees and select the optimal number of trees. Initially, We have achieved 99.78% of test accuracy with 10 trees, 99.74% of test accuracy with 20 trees, 99.60% of test accuracy with 30 trees, 99.53 with 40 trees, and 99.23 with 100 trees. We observe that accuracy is decreasing with increasing the number of trees so the optimal number of trees with the highest accuracy is 10 as shown in figure 11.

So, we have achieved 99.78% of test accuracy with our dataset.

for neutral results, we have use 10-fold cross-validation method. We present the accuracy, precision, recall, and f1-scores' result of each fold in table 10.

Table 10: 10-folds cross validation (Class)

Folds	Accuracy	Precision	Recall	F1-score
1	0.993	0.968	0.954	0.961
2	0.995	0.976	0.969	0.973
3	0.996	0.977	0.977	0.977
4	0.994	0.991	0.946	0.968
5	0.996	0.992	0.962	0.980
6	0.996	0.969	0.984	0.977
7	0.995	0.962	0.984	0.973
8	0.997	0.970	1	0.984
9	0.996	0.976	0.976	0.976
10	0.998	0.984	0.992	0.988

We can view the class error prediction in figure 12. We assess the classifier utilizing performance evaluation metrics which include precision, recall, and F1 score as shown in figure 13 and table 8. In classification, precision is the negligible part of significant occasions among the recovered cases, while recall is the small portion of applicable cases that were recovered. Both accuracy and recall are hence founded on significance. F1-score is characterized as a proportion of a model's precision on a dataset. It is utilized to assess parallel classification frameworks, which order models into 'positive' or 'negative'.

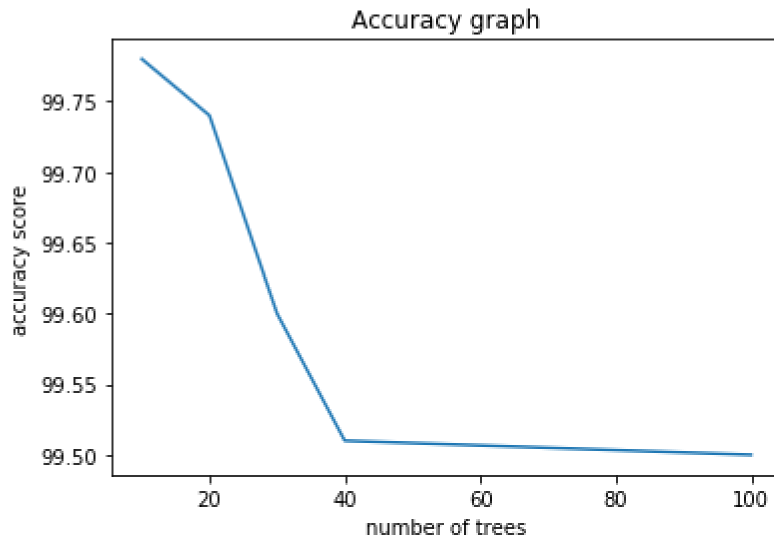

Figure 11: Accuracy graph (Class)

Table 12: performance evaluation table (class)

Accuracy	Precision	Recall	F1 score
99.78%	98%	97%	98%

5.1.3 Model validation:

Our proposed model helps in software fault prediction. We performed a case study on a project named Joda-time from defects4j. We picked business logic classes. platform depended classes are not considered. We have source code and complete information about faulty classes of the project. The model helps the tester to identify where in the application we have high chances of occurrence of faults. It helps the organization to plan their testing activities from the early stages of the software development life cycle because faults become costly when the applications go live.

Figure 12: Class prediction error graph (class)

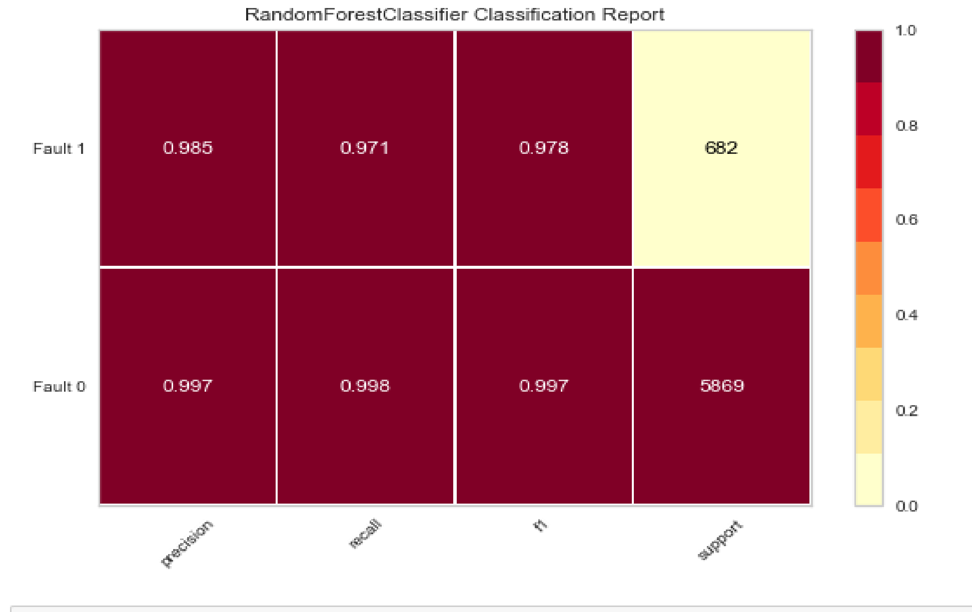

Figure 13: Classification report (class)

As we have developed a model via machine learning in Random forest based on which we predict software faults. For validation of our model, we used a project named CSV from defects4j. we have complete information of source code and faulty classes of the project. We picked 30 different classes of the project and extract smells of these classes using smells extraction tool iPlasma. Among these 30 classes, 15 classes are faulty and 15 classes are non-faulty. we generate csv file of smells. After that, we input this csv file into our model and our model predicts faulty classes. our model predicts 13 faulty classes and 17 non-faulty classes as shown in Table 12 and figure 14.

Table 12: Predicted and actual results (Class)

Sr no.	Classes	Actual	Predicted
1	C1	F	NF
2	C2	NF	NF
3	C3	F	F
4	C4	F	F
5	C5	F	F
6	C6	NF	NF
7	C7	NF	NF
8	C8	F	F

9	C9	F	F
10	C10	NF	NF
11	C11	NF	NF
12	C12	F	F
13	C13	F	F
14	C14	F	F
15	C15	NF	NF
16	C16	F	F
17	C17	NF	NF
18	C18	NF	NF
19	C19	NF	NF
20	C20	NF	NF
21	C21	F	NF
22	C22	F	F
23	C23	NF	NF
24	C24	NF	NF
25	C25	F	F
26	C26	F	F
27	C27	NF	NF
28	C28	NF	NF
29	C29	NF	NF
30	C30	F	F

To check that how efficient our prediction is we use a strategy named Percentage of right prediction to check the percentage of right prediction, as shown in Equation1 and we use the percentage of wrong prediction to check the percentage of our wrong prediction, as shown in Equation 2. This methodology helped in reducing testing effort and time.

% of Right Prediction = classes predicted right/ total number of classes* 100 (1)

= 28/30

= 93.33%

% of Wrong prediction = classes predicted wrong/ total number of classes* 100 (2)

= 2/30

= 6.66%

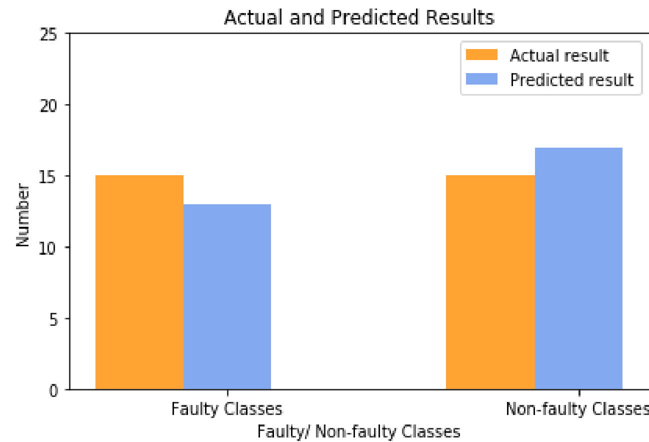

Figure 14: Actual and predicted Results (Class)

5.2 fault prediction at method level:

To demonstrate the validity of our proposed model, we took a project Joda-time from defect4j. Joda-time provides replacement for java date and time class. It allows multiple calendars systems. Joda-Time solves one critical problem stale time zone data. [74]. There is a total of 145 different classes. We extract code smells from the project using a tool named IPlasma. We extract code smells of 30 methods of these 30 classes. from 145 classes we picked only business logic classes. other classes that are platform depended are not considered. As per our proposed model we are interested to predict the faulty method through a trained Bagging classifier.

5.2.1 Training of Model

We have a dataset of method level which we used to train our model. Our training dataset has 7 columns. The first 6 Columns names are “Brain Method”, “Extensive Coupling”, “Feature Envy”, “Intensive Coupling”, “LongParaList”, “Shotgun Surgery” while the 7th column is our labeled column.

we split our data into 70 and 30. we are using 70% of the dataset for model training while the rest of 30% is used for testing to ensure the test results of trained model.

We are using Bagging Classifier with Random forest as base classifier. Bagging uses a basic methodology that appears in measurable examinations over and over to improve the gauge of one by joining the assessments of many. Bagging construct and grouping

classifiers utilizing bootstrap testing of the preparation information and afterward consolidates their prediction to create a final meta-prediction.

5.2.2 Model evaluation:

We are using random forest classifier. To select the most optimal value for the number of trees we calculate accuracy with the different number of trees and select the optimal number of trees. Initially, We have achieved 85.18% of test accuracy with 10 trees, 86.41% of test accuracy with 20 trees, 85.16% of test accuracy with 30 trees, 85.13% of test accuracy with 40 trees. We observe that accuracy is decreasing with increasing the number of trees so the optimal number of trees with the highest accuracy is 20 as shown in figure 15.

So, we have achieved 86.41% of test accuracy with our dataset.

for neutral results, we have use 10-fold cross validation method. We present accuracy, precision, recall, and f1-scores' result of each fold in table 13.

Table 13: 10-fold cross validation (Method)

Folds	Accuracy	Precision	Recall	F1-score
1	0.94	0.836	0.816	0.813
2	0.842	0.829	0.800	0.830
3	0.848	0.830	0.810	0.930
4	0.874	0.834	0.834	0.909
5	0.851	0.843	0.843	0.904
6	0.96	0.837	0.837	0.837
7	0.872	0.831	0.829	0.829
8	0.816	0.810	0.701	0.718
9	0.813	0.838	0.73	0.73
10	0.821	0.812	0.80	0.70

We can view the class error prediction in figure 16. We assess the classifier utilizing performance evaluation metrics which are precision, recall, and F1 score as shown in figure 17 and table 14. In classification, precision is the negligible part of significant occasions among the recovered cases, while recall is the small portion of applicable cases that were recovered. Both accuracy and recall are hence founded on significance. F1-score is characterized as a proportion of a model's precision on a dataset. It is utilized to assess parallel classification frameworks, which order models into 'positive' or 'negative'.

Table 14: performance evaluation table (method)

Accuracy	Precision	Recall	F1 score
86.41%	83%	80%	82%

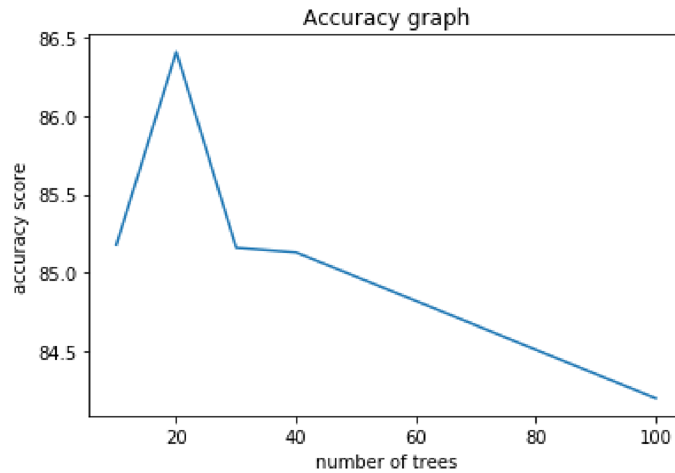

Figure 15: Accuracy graph (Method)

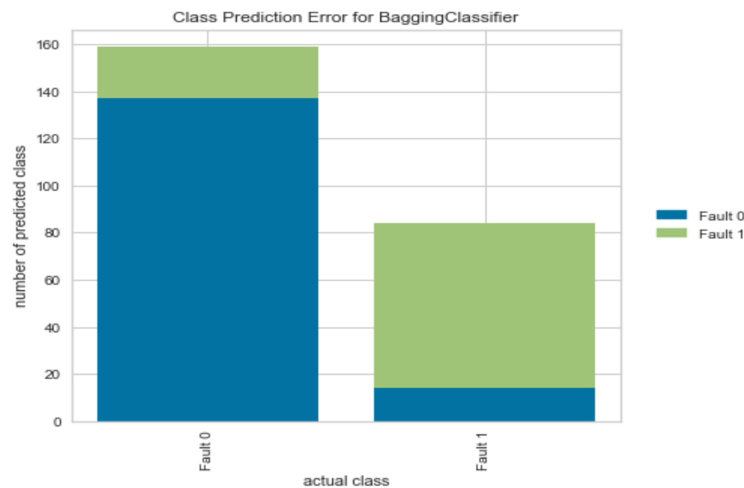

Figure 16: Class prediction error graph (method)

5.2.3 Model validation:

Our proposed model helps in software fault prediction. We performed case study on a Joda-time project from defects4j. We picked business logic classes. platform depended classes are not considered. We have source code and complete information about the faulty methods of the project. The model helps the tester to identify where in the application we have high chances of occurrence of faults. It helps the organization to plan their testing activities from the early stages of the software development life cycle because faults become costly when the applications go live.

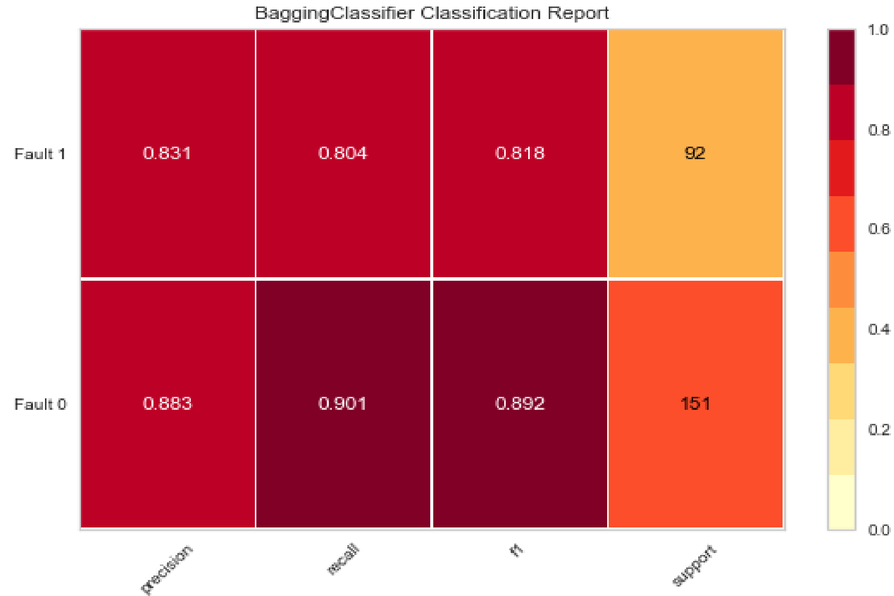

Figure 17: Classification report (method)

As we have developed a model via machine learning in Random forest based on which we predict software faults. For validation of our model, we used a project named CSV from defects4j. we have complete information of source code and faulty methods of the project. We picked 30 different methods of the project and extract smells of these methods using smells extraction tool iPlasma. Among these 30 methods, 15 methods are faulty, and 15 methods are non-faulty. After that, we input csv file into our model and our model predict faulty methods. our model predicts 10 faulty methods and 20 non-faulty methods as shown in Table 15 and figure 18.

Table 15: Predicted and actual results (Method)

Sr no.	Method	Actual	Predicted
1	M1	NF	NF
2	M2	F	F
3	M3	F	NF
4	M4	NF	NF
5	M5	F	F
6	M6	F	NF
7	M7	NF	NF
8	M8	NF	NF
9	M9	NF	NF

10	M10	NF	NF
11	M11	F	NF
12	M12	F	F
13	M13	F	F
14	M14	NF	NF
15	M15	NF	NF
16	M16	F	NF
17	M17	NF	NF
18	M18	F	F
19	M19	F	F
20	M20	F	NF
21	M21	NF	NF
22	M22	NF	F
23	M23	NF	NF
24	M24	NF	NF
25	M25	F	F
26	M26	NF	NF
27	M27	NF	NF
28	M28	F	F
29	M29	F	NF
30	M30	F	F

To check that how efficiently our model predicts, we use a strategy named Percentage of right prediction to check the percentage of right prediction, as shown in Equation 3 and we use percentage of wrong prediction to check the percentage of our wrong prediction, as shown in Equation 4.

% of Right Prediction = methods predicted right/ total number of methods* 100 (3)

= 25/30

= 83.33%

% of Wrong prediction = methods predicted wrong/ total number of methods* 100(4)

= 5/30

= 16.66%

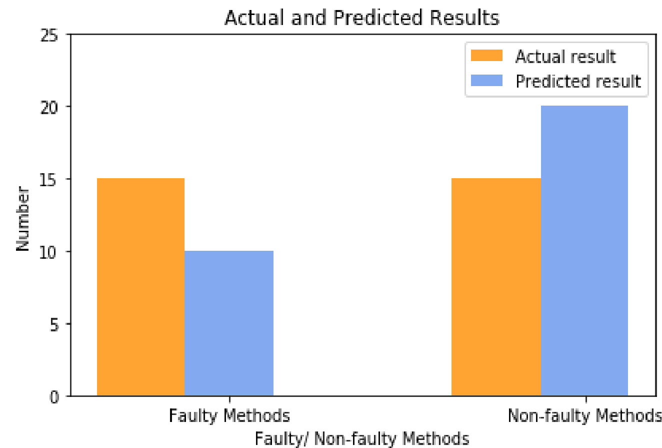

Figure 18: Actual and Predicted Results (Method)

Software development life cycle has many phases among them quality is one of the most important stages. Early-stage fault prediction plays an important role in improving the quality of software. It saves time and cost. Software development is becoming an emerging field with the enhancement of applications used in day-to-day life and it is increasing interaction with technology at a rapid pace. This vast usage of applications enhances the importance of Quality Assurance. Bugs found after production release can be very costly. Limiting the number of bugs in software is an exertion key to software engineering, faulty code neglects to satisfy the reason it was composed for, lastly fixing it costs time and money. resources in a software development life cycle are quite often restricted and thusly ought to be allocated to where they required most to avoid bugs, they ought to center around the most fault-prone areas of the project. Having the option to predict where such areas may be would permit a greater turn of events and testing endeavors to be allocated to the correct places.

Utilizing bug prediction models at the degree of individual methods can save effort in many aspects. It saves effort in terms of time and terms of cost. Moreover, this approach of predicting faults at the method level reduces manual inspection efforts for developers.

5.3 Comparison:

We do comparison of our methodology with some other proposed methodologies. The basic criterion of comparison is at which level faults are predicted. After level, the next criteria is number of code smells used for fault prediction at class level and accuracy, Precision, recall, and f1 score of our methodology and other methodology.

We find several fault prediction approaches providing fault prediction at the class level and the method level. A study proved by giving empirical evidence that code smells are helpful in fault prediction. The metrics they used in their study are source code and code smells with different machine learning techniques like Naïve Bayes, random forest, and Logistic Regression techniques. [53]

The author in another study evaluates class level change-prone prediction power using code smells. they used many machine learning approaches. The result of this experiment indicates positive relation between code smells and class change proneness with a probability superior to 70%. [55].

Another study looks into that that how code smells and fault prediction are related. To study this relation, they used many techniques like ADTree, Naïve Bayes, Logistic regression, and Multilayer perceptron. Results of this study showed that the combined model improves F-measure up to 20%. [56].

Another author in his study confirms that code smells are directly related to software faults and the performance of fault prediction models. They use Naive Bayes and Logistic Regression techniques. [57]

In another study, the author assesses the benefaction of a proportion of the intensity of code smells. for this, they add code smells to the existing prediction model which used process and product metrics. they compare both existing and new models. As of result of this experiment by adding code smells a predictor, the accuracy of the prediction model increases [54]

It is worth mentioning that most of the approaches do fault prediction at the class level leaving method level predictions. Instead, we predict faults at both class and method levels. We validate our methodology using a few case studies. We show a comparison of different fault prediction techniques proposed by different authors with our approach as shown in Table 16.

Table 16: Comparison

Papers	Fault prediction						
	Class level						Method level
	Prediction	Attributes	Accuracy	Precision	Recall	F1-score	
[59]	✓	5	99.73%	83%	95%	88%	**X**
[60]	✓	6	-	76%	84%	80%	**X**
[61]	✓	2	-	80.61%	90.98%	-	**X**
[62]	✓	6	-	-	-	75%	**X**
[75]	✓	5	-	90%	72.7%	50%	**X**
[16]	✓	8	-	-	62%	-	**X**
[52]	✓	1	-	-	-	-	**X**
[76]	✓	8	54.09%	-	-	-	**X**
[77]	✓	1	86.68%	-	-	61%	**X**
Our methodology	✓	12	99.78%	98%	97%	98%	✓

Chapter 6: Conclusion:

This chapter discusses the conclusion of our work on software fault prediction using cod smells, results, and summarization of our research work. In the addition, this chapter also provides an overview of performed research and future work.

Different researchers as discussed in literature do research work on fault prediction using code smells at the class level. Different authors use different types of code smells and use different machine learning or regression techniques for fault prediction. All this work is done at the class level.

There are more than 30 code smells in the literature, but the authors used 5-8 different code smells. Impressive work has been done to predict faults at the class level but very few studies are there to predict faults at the method level. Researchers that are predicting faults at the method level used process metrics or software metrics for example CK metrics. no work is done in the field of fault prediction at the method level using code smells.

We proposed an effective and efficient model. We use pseudo labeling technique with random forest at class level fault prediction and we use bagging classifier with random forest as the base classifier for fault prediction at method level. The model developed using these techniques can predict software faults at class and method level using code smells. The model can enhance and used for object-oriented or process metrics provided with the structured dataset. The dataset used in the model building consists of different types of code smells that are extracted from source code. For class-level prediction we are using 12 different code smells and for method level we use 6 different method level code smells. The labeled dataset is collected from projects available at defects4j. the dataset is pre-processed and structured according to the requirements of the model being implemented. Our contribution in this research work is the model developed to predict software faults using code smells at class level and method level via machine learning technique.

The accuracy and performance of our model depend upon training dataset. Our system generated 99% and 86% test accuracy under semi/supervised machine learning algorithm for class level prediction and method level prediction respectively.

In addition to accuracy, we use performance evaluation metrics to evaluate our model. Performance evaluation metrics have precision, recall, and F1-score, for unbiased evaluation we use 10 cross-validations.

For validation of our model, we use defects4j. we extract code smells from projects available at dafects4j and input these code smells to our model, as output our model predict faulty instances. We use the percentage of the right prediction and the percentage of the wrong prediction to check the efficiency of our predictions. We achieved a satisfactory percentage of right prediction both at the class level and at method level.

From our experiment and results, we observe that we can predict faults using code smells and code smells are a good index for software fault prediction at class level and method level.

For future work we are planning to use code smells with deep learning, and we are planning to use some other software metrics or process metrics with machine learning and deep learning to predict faults at the method level.

References:

1. Grottke, M. and K.S. Trivedi, *A classification of software faults.* Journal of Reliability Engineering Association of Japan, 2005. **27**(7): p. 425-438.
2. *software bug accessed*. 2018; Available from: https://en.wikipedia.org/wiki/Software_bug.
3. *defects4j*. Available from: https://en.wikipedia.org/wiki/Code_smell.
4. Fowler, M., et al., *Refactoring: improving the design of existing code, ser*, in *Addison Wesley object technology series.* 1999, Addison-Wesley.
5. Moha, N., et al., *Decor: A method for the specification and detection of code and design smells.* IEEE Transactions on Software Engineering, 2009. **36**(1): p. 20-36.
6. Halstead, M.H., *Elements of software science*. Vol. 7. 1977: Elsevier New York.
7. McCabe, T.J., *A complexity measure.* IEEE Transactions on software Engineering, 1976(4): p. 308-320.
8. Chidamber, S.R. and C.F. Kemerer, *A metrics suite for object oriented design.* IEEE Transactions on software engineering, 1994. **20**(6): p. 476-493.
9. Bansiya, J. and C.G. Davis, *A hierarchical model for object-oriented design quality assessment.* IEEE Transactions on software engineering, 2002. **28**(1): p. 4-17.
10. Tang, M.-H., M.-H. Kao, and M.-H. Chen. *An empirical study on object-oriented metrics*. in *Proceedings sixth international software metrics symposium (Cat. No. PR00403)*. 1999. IEEE.
11. Martin, R., *OO design quality metrics.* An analysis of dependencies, 1994. **12**(1): p. 151-170.
12. Ma, W., et al., *Empirical analysis of network measures for effort-aware fault-proneness prediction.* Information and Software Technology, 2016. **69**: p. 50-70.
13. Menzies, T., J. Greenwald, and A. Frank, *Data mining static code attributes to learn defect predictors.* IEEE transactions on software engineering, 2006. **33**(1): p. 2-13.
14. Ma, Y., L. Guo, and B. Cukic, *A statistical framework for the prediction of fault-proneness*, in *Advances in Machine Learning Applications in Software Engineering*. 2007, IGI Global. p. 237-263.
15. Koru, A.G. and H. Liu. *An investigation of the effect of module size on defect prediction using static measures*. in *Proceedings of the 2005 workshop on Predictor models in software engineering*. 2005.
16. Ma, W., et al. *Do we have a chance to fix bugs when refactoring code smells?* in *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. 2016. IEEE.
17. Olague, H.M., et al., *Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes.* IEEE Transactions on software Engineering, 2007. **33**(6): p. 402-419.
18. Cruz, A.E.C. and K. Ochimizu. *Towards logistic regression models for predicting fault-prone code across software projects*. in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. 2009. IEEE.
19. Burrows, R., et al. *The impact of coupling on the fault-proneness of aspect-oriented programs: An empirical study*. in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. 2010. IEEE.
20. Kapila, H. and S. Singh, *Analysis of CK metrics to predict software fault-proneness using bayesian inference.* International Journal of Computer Applications, 2013. **74**(2).
21. Dejaeger, K., T. Verbraken, and B. Baesens, *Toward comprehensible software fault prediction models using bayesian network classifiers.* IEEE Transactions on Software Engineering, 2012. **39**(2): p. 237-257.
22. Singh, Y., A. Kaur, and R. Malhotra. *Software fault proneness prediction using support vector machines*. in *Proceedings of the world congress on engineering*. 2009.

23. Goyal, R., P. Chandra, and Y. Singh, *Suitability of KNN regression in the development of interaction based software fault prediction models.* Ieri Procedia, 2014. **6**(1): p. 15-21.
24. Fokaefs, M., et al. *An empirical study on web service evolution*. in *2011 IEEE International Conference on Web Services*. 2011. IEEE.
25. Malhotra, R. and A. Jain, *Fault prediction using statistical and machine learning methods for improving software quality.* Journal of Information Processing Systems, 2012. **8**(2): p. 241-262.
26. Pai, G.J. and J.B. Dugan, *Empirical analysis of software fault content and fault proneness using Bayesian methods.* IEEE Transactions on software Engineering, 2007. **33**(10): p. 675-686.
27. Radjenović, D., et al., *Software fault prediction metrics: A systematic literature review.* Information and software technology, 2013. **55**(8): p. 1397-1418.
28. Gondra, I., *Applying machine learning to software fault-proneness prediction.* Journal of Systems and Software, 2008. **81**(2): p. 186-195.
29. Lu, H. and B. Cukic. *An adaptive approach with active learning in software fault prediction*. in *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*. 2012.
30. Abaei, G., A. Selamat, and H. Fujita, *An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction.* Knowledge-Based Systems, 2015. **74**: p. 28-39.
31. *Everything about code smells*. Available from: https://www.codegrip.tech/productivity/everything-you-need-to-know-about-code-smells/.
32. Walter, B., F.A. Fontana, and V. Ferme, *Code smells and their collocations: A large-scale experiment on open-source systems.* Journal of Systems and Software, 2018. **144**: p. 1-21.
33. Piotrowski, P. and L. Madeyski, *Software defect prediction using bad code smells: A systematic literature review.* Data-Centric Business and Applications, 2020: p. 77-99.
34. Rathore, S.S. and A. Gupta. *Investigating object-oriented design metrics to predict fault-proneness of software modules*. in *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*. 2012. IEEE.
35. Chen, J., et al. *Empirical studies on feature selection for software fault prediction*. in *Proceedings of the 5th Asia-Pacific Symposium on Internetware*. 2013.
36. Rathore, S.S. and S. Kumar, *Towards an ensemble based system for predicting the number of software faults.* Expert Systems with Applications, 2017. **82**: p. 357-382.
37. Singh, P., et al., *Fuzzy rule-based approach for software fault prediction.* IEEE Transactions on Systems, Man, and Cybernetics: Systems, 2016. **47**(5): p. 826-837.
38. Arshad, A., et al., *Semi-supervised deep fuzzy c-mean clustering for software fault prediction.* IEEE Access, 2018. **6**: p. 25675-25685.
39. Aziz, S.R., T. Khan, and A. Nadeem, *Experimental Validation of Inheritance Metrics' Impact on Software Fault Prediction.* IEEE Access, 2019. **7**: p. 85262-85275.
40. Kumar, L. and A. Sureka. *Analyzing fault prediction usefulness from cost perspective using source code metrics*. in *2017 Tenth International Conference on Contemporary Computing (IC3)*. 2017. IEEE.
41. Singh, P. *Comprehensive model for software fault prediction*. in *2017 International Conference on Inventive Computing and Informatics (ICICI)*. 2017. IEEE.
42. Khomh, F., M. Di Penta, and Y.-G. Gueheneuc. *An exploratory study of the impact of code smells on software change-proneness*. in *2009 16th Working Conference on Reverse Engineering*. 2009. IEEE.

43. Lozano, A., M. Wermelinger, and B. Nuseibeh. *Evaluating the harmfulness of cloning: A change based experiment*. in *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. 2007. IEEE.
44. Li, W. and R. Shatnawi, *An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution.* Journal of systems and software, 2007. **80**(7): p. 1120-1128.
45. Sjøberg, D.I., et al., *Quantifying the effect of code smells on maintenance effort.* IEEE Transactions on Software Engineering, 2012. **39**(8): p. 1144-1156.
46. Guo, Y., et al. *Domain-specific tailoring of code smells: an empirical study*. in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 2010.
47. Eken, B., et al., *An empirical study on the effect of community smells on bug prediction.* Software Quality Journal, 2021. **29**(1): p. 159-194.
48. Rio, A., *PHP code smells in web apps: survival and anomalies.* arXiv preprint arXiv:2101.00090, 2020.
49. Kessentini, M., *Understanding the correlation between code smells and software bugs.* 2019.
50. Sotto-Mayor, B., et al., *Exploring Designite for Smell-Based Defect Prediction.*
51. Van Emden, E. and L. Moonen. *Java quality assurance by detecting code smells*. in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* 2002. IEEE.
52. Zazworka, N., et al. *Investigating the impact of design debt on software quality*. in *Proceedings of the 2nd Workshop on Managing Technical Debt*. 2011.
53. Hall, T., et al., *Some code smells have a significant but small effect on faults.* ACM Transactions on Software Engineering and Methodology (TOSEM), 2014. **23**(4): p. 1-39.
54. Vidal, S.A., C. Marcos, and J.A. Díaz-Pace, *An approach to prioritize code smells for refactoring.* Automated Software Engineering, 2016. **23**(3): p. 501-532.
55. Liu, H., et al., *Schedule of bad smell detection and resolution: A new way to save effort.* IEEE transactions on Software Engineering, 2011. **38**(1): p. 220-235.
56. Lozano, A., M. Wermelinger, and B. Nuseibeh. *Assessing the impact of bad smells using historical information*. in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. 2007.
57. Dhillon, P.K. and G. Sidhu, *Can software faults be analyzed using bad code smells?: An empirical study.* Int J Sci Res Publ, 2012. **2**(10): p. 1-7.
58. Olbrich, S.M., D.S. Cruzes, and D.I. Sjøberg. *Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems*. in *2010 IEEE International Conference on Software Maintenance*. 2010. IEEE.
59. Ubayawardana, G.M. and D.D. Karunaratna. *Bug prediction model using code smells*. in *2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer)*. 2018. IEEE.
60. Palomba, F., et al., *Toward a smell-aware bug prediction model.* IEEE Transactions on Software Engineering, 2017. **45**(2): p. 194-218.
61. Pritam, N., et al., *Assessment of code smell for predicting class change proneness using machine learning.* IEEE Access, 2019. **7**: p. 37414-37425.
62. Catolino, G., et al., *Improving change prediction models with code smell-related information.* Empirical Software Engineering, 2020. **25**(1): p. 49-95.
63. Soltanifar, B., et al. *Software analytics in practice: a defect prediction model using code smells*. in *Proceedings of the 20th International Database Engineering & Applications Symposium*. 2016.

64. Palomba, F., et al. *Smells like teen spirit: Improving bug prediction performance using the intensity of code smells*. in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016. IEEE.
65. Giger, E., et al. *Method-level bug prediction*. in *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. 2012. IEEE.
66. Pascarella, L., F. Palomba, and A. Bacchelli. *Re-evaluating method-level bug prediction*. in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018. IEEE.
67. *pseudo labeling*. Available from: https://towardsdatascience.com/pseudo-labeling-to-deal-with-small-datasets-what-why-how-fd6f903213af.
68. Khalilia, M., S. Chakraborty, and M. Popescu, *Predicting disease risks from highly imbalanced data using random forest.* BMC medical informatics and decision making, 2011. **11**(1): p. 1-13.
69. Banker, R.D., et al., *Software errors and software maintenance management.* Information Technology and Management, 2002. **3**(1): p. 25-41.
70. Caram, F.L., et al., *Machine learning techniques for code smells detection: a systematic mapping study.* International Journal of Software Engineering and Knowledge Engineering, 2019. **29**(02): p. 285-316.
71. *Defects4J Dissection*. Available from: http://program-repair.org/defects4j-dissection/#!/.
72. *IPLASMA*. Available from: http://loose.cs.upt.ro/index.php?n=Main.IPlasma.
73. *Code Smells and their Collocations : A Large-scale Experiment on Open-source Systems*. Available from: http://doi.org/10.5281/zenodo.842778.
74. *joda-time*. june 26]; Available from: https://github.com/dlew/joda-time-android.
75. Bigonha, M.A., et al., *The usefulness of software metric thresholds for detection of bad smells and fault prediction.* Information and Software Technology, 2019. **115**: p. 79-92.
76. Qusef, A., M.O. Elish, and D. Binkley, *An exploratory study of the relationship between software test smells and fault-proneness.* IEEE Access, 2019. **7**: p. 139526-139536.
77. Gradišnik, M., et al. *Adapting God Class thresholds for software defect prediction: A case study*. in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2019. IEEE.