DEBRE BIRHAN UNIVERSITY

COLLEGE OF COMPUTING

DEPARTMENT OF INFORMATION SYSTEMS

POST GRADUATE PROGRAM

DETECTING CODE SMELLS USING MACHINE LEARNING TECHNIQUES

BY

YORDANOS FISSEHAYE

A Thesis Submitted to Graduate Program in Partial Fulfillment of the Requirements for the Degree of Master of Science in Information Systems

JUNE, 2022

DEBRE BIRHAN

ETHIOPIA

DEBRE BIRHAN UNIVERSITY

COLLEGE OF COMPUTING

DEPARTMENT OF INFORMATION SYSTEMS

As members of the Board of examiners of the final Master's degree open defense, we certify that we have read and evaluated the thesis prepared by Yordanos Fissehaye under the title "Detecting Code Smells Using Machine Learning Techniques" and examined the candidate. This is therefore, to certify that the thesis has been accepted in partial fulfillment of the requirement for the degree of Masters of Science in Information Systems.

Name and Signature of Members of the Examining Board

Name	**Title**	**Signature**	**Date**
________________________	Chairperson	__________	__________
Solomon Demissie (Ph.D.)	Advisor	__________	__________
________________________	Examiner	__________	__________
________________________	Examiner	__________	__________

DECLARATION

This thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree in any university. This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by citations giving explicit references. A list of references is appended.

____

Yordanos Fissehaye

June, 2022

This thesis has been submitted for examination with my approval as university advisor.

____

Solomon Demissie (Ph.D.)

June, 2022

DEDICATION

This thesis work is dedicated to my families and friends.

ACKNOWLEDGMENT

First and foremost, I would like to thank the almighty God and his Mother Saint Marry for giving me the strength and ability to undertake this study. Without His permission, none of this would have been possible.

I would like to express my gratitude to my advisor, Solomon Demissie (Ph.D.), for his unconditional support and direction throughout the writing of my thesis. His inspiring advice, insightful comments, and patience were extremely essential and valuable in my thesis work. Besides, I really appreciate his concern and want to thank him for always being there whenever I needed his support.

I would really like to thank all the respondents who showed their willingness to take part in the study and provide their knowledge in the quality assessment test of the selection stage of the Systematic Literature Review part.

I would also like to thank my friends who helped me by providing the documents whose access is restricted.

An acknowledgement of this kind would be incomplete without the deepest gratitude to my families and my friends who are always the source of my strength. I want to thank them for their support and unconditional love. Thank you all.

Yordanos Fissehaye

Table of Contents

List of Tables

List of Figures

List of Acronyms

ACM	Association for Computing Machinery
AMWNAMM	Average Methods Weight of Not Accessor or Mutator Methods
ATFD	Access to Foreign Data
ATLD	Access to Local Data
AUC	Area Under Curve
BBNs	Bayesian Belief Networks
CBO	Coupling Between Objects classes
CC	Changing Classes
CDISP	Coupling Dispersion
CFNAMM	Called Foreign Not Accessor or Mutator Methods
CINT	Coupling Intensity
CLNAMM	Called Local Not Accessor or Mutator Methods
CM	Changing Methods
CNN	Convolutional Neural Network
CYCLO	Cyclomatic Complexity
DIT	Depth of Inheritance Tree
DT	Decision Tree
FANIN	Number of modules that call a given module
FANOUT	Numbers of modules that called by a given module.
FDP	Foreign Data Providers
FE	Feature Envy
FN	False Negative
FP	False Positive
GBT	Gradient Boosting
IEEE	Institute of Electrical and Electronics Engineers
KNN	K-Nearest Neighbor
LAA	Locality of Attribute Accesses
LCOM5	Lack of Cohesion in Methods
LM	Long Method

LOC	Lines of Codes
LOCNAMM	Lines of Codes without Accessor or Mutator Methods
LR	Linear Regression
MaMCL	Maximum Message Chain Length
MAXNESTING	Maximum Nesting Level
MCD	Multi Class Dataset
MeMCL	Mean Message Chain Length
ML	Machine Learning
MLD	Multi Label Dataset
MLP	Multi-Layer Perceptron
MNB	Multinomial Naïve Bayes
NB	Naïve Bayes
NBG	Naive Bayes Gaussian
NIM	Number of inherited methods
NMCS	Number of Message Chain Statements
NMO	Number of Methods Overridden
NOA	Number of Attributes
NOAM	Number of Accessor Methods
NOAV	Number of Accessed Variables
NOC	Number of Children
NOCS	Number of Classes
NOI	Number of Interfaces
NOII	Number of Implemented Interfaces
NOLV	Number of Local Variable
NOM	Number of Methods
NOMNAMM	Number of Not Accessor or Mutator Methods
NOP	Number of Parameters
NOPA	Number of Public Attributes
NOPK	Number of Packages
PRC	Precision-Recall Curve
RBFs	Radial Basis Function Networks

RF	Random Forest
RFC	Response for a Class
ROC	Receiver Operating Characteristic
SLR	Systematic Literature Review
SMO	Sequential Minimal Optimization
SMOTE	Synthetic Minority Over-sampling Technique
SVMs	Support Vector Machines
TCC	Tight Class Cohesion
TN	True Negative
TP	True Positive
WMC	Weighted Methods Count
WMCNAMM	Weighted Methods Count of Not Accessor or Mutator Methods
WOC	Weight of Class

ABSTRACT

Code smell is a poor design choice by developers that could compromise software systems' general maintainability, clarity, and complexity. It reflects poor design or implementation decisions in the source code, which makes it more change and fault-prone. Researchers developed a number of code smell detectors that use various sources of data to assist developers in discovering design faults. Despite their high accuracy, earlier research has identified three major drawbacks that could prevent code smell detectors from being used in practice: (i) Developers' subjective perceptions of code smells discovered by such tools, (ii) little agreement across different detectors, and (iii) difficulty in determining appropriate detection thresholds. Machine learning techniques are becoming increasingly popular as a means of overcoming these constraints.

Hence, this study has performed a Systematic Literature Review with the aim of exploring the code smells detected, Machine learning Techniques deployed and datasets used. The Systematic Literature Review was performed on three online databases. Accordingly, Long method, Feature envy, God class and Data class are the most widely studied code smells. While Random Forest, Decision tree, Naïve Bayes and SVM are the most widely used machine learning techniques. Additionally, Qualities corpus, Xerces and other project datasets that are not explicitly mentioned are the most commonly used datasets in the studies from 2017-2020. This research has also performed an experiment (Replication study) using four different machine-learning algorithms (J48, Random Forest, JRIP and Naïve Bayes). These algorithms were applied on two code smells (Long method and Feature envy) that are selected via conducting a Systematic Literature Review. 395 code smell samples were used. The four machine learning algorithms are chosen based on their strong performance in multi-class dataset as determined by mapping study. The results demonstrate that all the adopted algorithms have performed above 90 % accuracy with the exception that Random Forest algorithm shows the highest performance with respect to most performance metrics given the dataset while the worst performance was achieved by Naïve Bayes.

However, the dataset's lower prevalence of code smell instances and nature of projects resulted in different results that will need to be addressed in future studies. The research concluded that the application of machine learning to the detection of these code smells can provide high accuracy.

Keyword: Code Smells, Machine Learning, Systematic Literature Review **(SLR),** Replication study

CHAPTER ONE
INTRODUCTION

1.1 Background

Software systems must be regularly altered by developers during software maintenance and evolution in order to add new needs, expand existing functionality, or solve critical issues [1]. However, developers may not always have the time or willingness to keep the complexity of the system under control and find good design solutions before applying their modifications [2] due to time constraints or community-related factors [3]. As a result, development operations are frequently carried out in an undisciplined manner, which has the effect of eroding the system's original design by introducing so-called technical debt [4] [5]. One of the most dangerous forms of technical debt [6] [7] is code smells [8], which are indications of the presence of bad design or implementation choices in the source code.

Code smells are indicators of probable code or design flaws in a system [9] [10]. They are also known as design faults, and they refer to design problems that have a detrimental impact on the software's maintainability [11]; As a result, they may have an impact on maintenance procedures [12]. Code smells are generated by programming and design errors made by software developers while the software is being designed and programmed [10]. They can also occur as a result of improper analysis, incorrect integration of new models into the system, disregarding software development principles, and sophisticated code composition [9] [10]. These smells could have a negative impact on the system's overall quality, such as its maintainability and understandability. [13] [14].

Bloaters, Object-Orientation Abusers, Change Preventers, Dispensable, and Couplers are the five most common bad smells among the several types [8] [15]. Bloaters are code or classes that have been enlarged to the point that they are difficult to deal with. These smells don't appear right away; instead, they build up over time as the program progresses, especially if no one tries to get rid of them. All of the smells in the Object-Orientation Abusers category, on the other hand, involve the erroneous or inadequate implementation of object-oriented programming principles. The smells in the Change Preventers category appear when you need to change something in one place in your code, but you also need to change a lot of other places. As a result, the software development process is more complicated and costly. The smells in the dispensable category occur when a section of the code is no longer needed and can be eliminated, making the code cleaner, more efficient, and easier to understand. Finally, all of the Couplers smells lead to excessive class coupling or show what happens when coupling is substituted with excessive delegation.

Identifying and fixing faults as they occur is a typical technique to avoid deterioration, and one of the key theories for doing so in object-oriented design is the detection of code-smells [16]. Code smell detection can be defined as the task of identifying potential code or design problems in a system [7]. Many scholars have proposed various techniques to cope with the occurrence of code smells in systems as a result of this.

Detecting bad smells in code or design and then doing the required refactoring methods as needed is a great way to improve the code's quality. This is because these smells make the system more difficult to maintain and, as a result, render it more prone to failure [17]. Consequently, bad smells are unlikely to cause failure directly, but they may do so indirectly, lowering software quality [18]. The technique of detecting bad smells can be done manually or utilizing automatic detection strategies. It is deemed subjective to perform the code smell detection method manually. This means that each code smell looks at a specific type of system piece (classes, methods, etc.) that may be assessed based on its characteristics [19]. Some of the disadvantages are, it is prone to errors and time-consuming [20], the programmers' personal interpretation is equally open to them [8], and there is no agreement among developers on how to define them [21] [22]. Automated approaches based on source code have been developed to eliminate this subjectivity [11] [23] [24] [25].

However, a significant portion of those approaches is dependent on code metrics [25] [26] [27] [28] [29]. These strategies use measures and thresholds that aren't always constant, resulting in an increase in false positives that don't represent true problems [22] because it does not take into account information about the system's context, domain, scale, or design [30]. As a result, given the subjective definition of code smells, it is necessary for the approach to be aware of and sensitive to the specific environment. Automated tools are proposed as a result [31] [32] and adoption of Machine Learning (ML) techniques is one of these approaches for detecting code smells [22] which are the most recently popular tools, which have shown to improve the overall performance of numerous prediction models and are highly promising [33] [34].

To construct a model, a machine learning classifier must first be trained using a set of code smell instances. The models are then utilized to find or detect code smells in previously unknown or new occurrences. The resulting model's power is determined by a number of factors, including the dataset, machine learning classifiers, and the classifier's parameters.

So, this study has applied machine learning techniques for the detection of two selected code smell types. Accordingly, a set of experiments are conducted on the target dataset.

1.2 Motivation of the Study

Codes instances are the building blocks of a complete software system. Hence, the overall quality of software systems depends on the quality of each of those code fragments. But, due to different factors like lack of time or willingness stated earlier, developers tend to evolve a software system that do not comply with the basic principles of software development. The result of such poor design then introduces the existence of one of the so-called technical debts which is code smells. Even if the existence of code smells does not necessarily imply system failure, one way or another, the aggregation of such things may bring a long-term effect in the software product. They might have an everlasting influence on the maintainability and extendibility of the software system. So, it is highly required to perform appropriate refactoring and to do so code smells should be detected earlier [8].

Different researchers have proposed a set of techniques to detect code smells ranging from the manual method to the use of tool based techniques. All of those techniques have their own limitations and areas of improvement in terms of detection like, consensus among developers, selection of threshold and the nature of dataset employed. This study focuses on the adoption of Machine learning techniques, because according to many researchers, it is a very recent trend and hot research area in the detection of code smell and need to be improved in terms of applying these techniques for code smell detection. This study is carried out with the motivation of addressing the potential gaps that are still not taken in to consideration by the previous works. Hence, tried to explore the connection between code smells and machine learning techniques.

1.3 Statement of the Problem

One of the most common problems in modern software engineering is the quality of program source code. With the introduction of agile approaches, there has been a rise in interest in ensuring and evaluating the quality of source code, which is the most important artifact in software development. The necessity to develop, adopt, and validate appropriate models and methodologies in that field has also become critical. Accordingly, there has been a continuous work done on these methodologies. In addition, there is recent shift away from human-based assessment toward automated methods [11].

Code smells [8], which are indicators of poor design or implementation choices in the source code, are one of the most dangerous forms of technical debt that degrade the quality of software products [6] [7]. Indeed, past study has demonstrated that they not only significantly diminish coders' ability to read source code [35], it also makes the affected classes more susceptible to change and faults [36] [37]. As a result, they pose a significant danger to maintainability effort and costs [14] [38].

Code smells can be detected by different techniques ranging from manual methods to automated source code analysis [29]. Manual code smell detection by developers is an error-prone, expensive, and time-consuming job that is dependent on the developer's level of experience and perception [27]. There are also indications that the removal of code smells is not being accomplished to a desirable degree, mainly to the fact that most developers are unaware of their presence [39]. Another challenge with identifying code smells is that it is subjective to the developer's perspective; what one developer deems a code smell may not be one for other developer [21] [22].

Attempts were made to develop meaningful metrics that can give suggestions on the existence of design faults in order to lessen the subjectivity component in code smell identification [21]. Another method for eliminating subjectivity is to utilize automatic tools to discover defective code, which translate source code into an intermediate representation and then do static analysis based on rules, metrics, and thresholds to find code smells [39] [25]. This method, on the other hand, ignores information about the system's context, domain, size, and design [30]. What appears to be a smell may actually be the finest approach to develop or design a (part of a) program. This lack of context resulted in false positives, with more than half of the automatically discovered code not being related to architectural issues [30].

On the contrary, heuristics-based approaches for detecting code smells have also been proposed in prior studies [22] [18]. They use a two-step approach in which they compute a set of measures and then apply thresholds to those metrics to distinguish between smelly and non-smelly classes. Those heuristic approaches differ in (i) the algorithms used to detect code smells (e.g., a combination of metrics or the usage of more complex methodology like Relational Topic Modeling) and (ii) the metrics employed (e.g., based on code metrics or historical data). Although it has been demonstrated that such detectors have reasonable performance in terms of accuracy, earlier research has revealed a number of significant limitations that may restrict their implementation in practice [39] [40]. Code smells detected by heuristic-based detectors, in particular, can be subjectively perceived by developers [41] [42]. At the same time, there is little consensus between them [11]. More crucially, the majority of them need that criterion be specified to separate smelly code components from non-smelly ones [39] and naturally, the choice of thresholds has a significant impact on their accuracy.

However, academic and industrial organizations are becoming interested in Machine Learning (ML) techniques for detecting code smells [43]. The fundamental benefit of these techniques is that they can uncover patterns that are difficult to specify using pre-determined or statistical criteria, as well as patterns that people cannot see [44]. While the research community looked into the approaches used by scholars and practitioners in the field of heuristics-based code smell detectors, [45] [46] [47], only a sprinkle of information exists on the methodologies used to create code smell prediction models using Machine Learning Techniques. As a result, this study has explored such strategies for code smell detection.

Accordingly, constructing the research questions is a significant step in the process of this study. So, the following four research questions are defined to achieve the research objective:

1. Which code smells are most commonly detected using machine learning techniques?
2. Which machine learning techniques are efficient to detect code smells?
3. What datasets have been used for code smell detection?
4. What result achieved after developing and evaluating the performance of the proposed machine learning model that would be used for detection of code smells?

1.4 Objective of the Study

1.4.1 General Objective

The general objective of this study is to assess the application of machine learning in the detection of code smells and design and develop a machine learning model that can able to detect code smells.

1.4.2 Specific Objectives

To accomplish the general objectives stated above, the study undergoes the following specific objectives:

- Conduct a literature review to systematically assess and analyze different researches on code smell prediction models and machine learning techniques that has been utilized to detect code smells.
- Select a dataset that influence the performance of the proposed machine learning model.
- Extract a dataset that influence the performance of the proposed machine learning model.
- Explore an appropriate machine learning technique to detect code smells.
- Identify an appropriate machine learning technique to detect code smells.
- Conduct an experimental work with the aim of experimenting and making comparison between selected machine learning techniques to find the most-efficient one for developing the final proposed model.
- Identify the performance metrics used to evaluate the proposed machine learning predictive model in terms of the detection of code smells.
- Evaluate the performance of the final developed model for its efficiency in code smell detection.

1.5 Significance of the Study

Both software development organizations/teams, and their clients can save money by improving software maintainability and quality [38]. Quality can also be a competitive advantage, affecting the company's long-term viability. Hence, there is an ongoing effort made to increase software quality by detecting code defects earlier. There are a set of smell detection techniques proposed by different scholars. Even if many studies have been undergone on this area, the ML branch is still in development and has much of potential for improvement when compared to static techniques. There is also a scarcity of empirical data to help the creation of new research on machine learning approaches for detecting code smells [22]. Solidifying empirical support can aid in recognizing the weaknesses of machine learning approaches and taking efforts to improve their performance, while also offering data to the research community about their benefits and gaps. Even though various researchers have presented a set of machine learning techniques, little attention has been paid to the adoption of other components of the approach. As a result, this study was able to analyze how this method might influence the detection process by offering a multi class approach instead of a binary class approach (a dataset representing the instances as smelly or not) that have been used in almost all the reviewed studies. The final result might then be utilized as a starting point for other researchers looking to follow a different method. It can be used as a starting point, and other methodologies can be used to improve the results obtained in this study. Because machine learning can give new and more effective ways of discovering code-smells than heuristics and metrics-based approaches, it is an area that has recently gotten a lot of interest [22]. It can also aid software firms in reducing rework and improving quality and reliability, as well as software engineers in increasing productivity.

1.6 Scope and Limitations of the Study

This study mainly focuses on two different but complimentary tasks. The first task is undergoing the systematic literature review. The systematic literature review in this study was performed in the documents from online digital libraries such as, IEEE explore, Springer and ACM. Additionally, the SLR is bounded to the year interval 2017 to 2020, hence only papers published in the specified time interval are assessed.

The second task which is the experimental work, was performed using supervised machine learning techniques. After applying all the appropriate preprocessing task, feature reduction has been applied and this is to filter the most important yet representative attributes/features. Feature reduction plays a vital role in enhancing the performance of the learning algorithm [48]. Then finally, a set of classification algorithms were applied. The classification techniques in this study were selected from different categories to investigate which category is best for predicting the specific code smell types. Accordingly, single classifiers from each category namely, tree-based classifiers, Bayesian classifiers and rule-based classifiers were applied. Finally, their performance was evaluated using the performance measurement metrics accuracy, precision and recall and F-measure.

This study focuses on finding and evaluation of the best performing machine learning model for the detection of code smells. As a result, the task of identification and prediction for new instances is out of the scope of the study. Accordingly, detection was made based on the portion of instances of a dataset and the performance of the classifiers was evaluated on the other portion. The class identification and prediction task for new unseen instances (which are out of the data used for training the algorithms), is not considered. Additionally, this study did not consider the detection of other types of code smells (other than the specified one) that might be found in the instances of the same dataset.

1.7 Organization of the Thesis

This study is organized in to seven chapters. The first chapter gave an initial overview on the overall work done by introduce the basic and core concepts, setting the major motive for this study and objectives for performing this study.

The Second chapter deals with the introduction of basic terms of this study which are, code smells and refactoring, machine learning techniques for code smell detection and Systematic Literature Review. Additionally, related works were revised to understand what has been done in this area and filtered out the potential gap in the area of detection of code smells using machine learning techniques.

The Third chapter is about methodology adopted by the study. This chapter deals with the presentation of the general research design and the proposed architecture. Hence, tries to give an overview of the methodology followed in this study.

The Fourth chapter is the first methodology adopted by the study which is, Systematic Literature Review. This chapter deals with study searching, selection and a systematic review of the selected studies. Hence, tries to find an answer to the first three research questions of this study.

The Fifth chapter is another methodological part, which is the experimental work (replication of a study). Hence, depending on the major ideas of the reference work, a set of experiments are conducted and compared using different performance measurement metrics.

The Sixth chapter deals with the presentation of basic findings of the SLR as well as the experimental work. Additionally, a comparison between the original study and the replicated one is made. As a result, an attempt will be done to answer all the research questions specified at the beginning of the study.

The final chapter, chapter seven is a conclusion part. Accordingly, in this chapter a conclusion was made based on the results found from the prior chapters and a bench mark was suggested for future studies to contribute to the knowledge.

CHAPTER TWO
LITERATURE REVIEW

2.1 Introduction

Maintenance is one of the most expensive aspects of software development [49]. According to the data published in [50] and [51], maintenance operations account for more than 75% of overall software costs. Maintainers of software spend over 60% of their time trying to understand the code [52]. Even with carefully built systems, source code quality tends to deteriorate as the project progresses, because a system's original design is rarely prepared for every new requirement, and changes must be made fast by several personnel without adequately adapting the system's structure [53]. Software inspection is a well-known technique for increasing code quality that includes a thorough analysis of the system's design, code, and documentation to look for probable issues based on previous experience [54]. However, software systems may continue to have various weaknesses that could cause a problem. Code smells are a result of bad design, and maintaining software with code smells is a time-consuming operation [55].

2.1.1 Code Smells

Code smells are code snippets with design problems, their presence in the code complicates the software maintenance and degrades the quality of software [49]. According to Azeem [56], Code smells are warning signs of possible software problems, and they have a detrimental impact on program quality. They are unambiguous indicators that the system's design is deteriorating, and that this long-term deterioration may lead to software rot. During the software design and programming stage, code smells are caused by programming and design errors made by software developers [10]. They can also happen for a variety of causes, including faulty analysis and integration of new models into the system, disregarding software development rules, and developing complex scripts [9] [10]. Each code smell looks at a specific type of system element (classes, methods, etc.) that can be evaluated based on its characteristics [19]. The code smells provide guidelines for identifying unwanted behaviors and typical coding errors, although they are still subject to programmers' interpretation [8].

Different methods can be used to identify code smells. Various coding smells detectors have been proposed by Marinescu, R. [45], Moha, N. etal [29] and Munro, M.J. [46]. According to Kessentini, W. etal [18], methods for detecting code smells are divided into seven categories (i.e., cooperative-based approaches, visualization-based approaches, machine learning-based approaches, probabilistic approaches, metric-based approaches, symptoms-based approaches, and manual approaches). According to Fontana, F.A. [11], because the same symptom can be caused by several diseases, or even by no disease at all, human judgment is required when evaluating smells in the context of the project where they are discovered, analogous to clinical diagnosis. However, due to the enormous number of code bases, manual code smell identification is an extremely time-consuming operation. Furthermore, the main difficulty of manual code smell detection according to the research in [22] [21] [47] is that the concept of what constitutes and does not constitute a code smell in a given situation may not be agreed upon by all developers working on the same project.

The deployment of heuristic based code smell detection technologies has been introduced to address this issue. Heuristic-based techniques, on the other hand, use a two-step procedure, with the first phase consisting of defining a set of measurements and the second step consisting of applying a threshold to those metrics to distinguish between smelly and non-smelly classes [56]. As a result, using a heuristic-based approach to detect code smells is regarded subjective, and the threshold used has an impact on its accuracy.

Hence, Automated approaches (tool-based) based on the source code have been offered in various publications in order to eliminate the subjectivity of interpretation [11] [23]. Recent research is focused on developing automatic detection tools to assist humans in locating smells when code size gets too huge for manual review, and these techniques could be useful in easing the process of locating code smells in large code bases [11]. A variety of commercial and open-source code smell detection techniques are available. Some are research prototypes that detect specific code smells, while others detect a broad spectrum of code smells [52] [57]. However, a significant portion of those approaches is dependent on code metrics [26] [45] [29]. Machine learning algorithms are one of these automatic smell detection methodologies. To address the restrictions mentioned previously, machine learning techniques are increasingly being used to detect code smells [22].

Machine learning approaches differ from heuristic-based approaches in that they use classifiers to distinguish between instances' smelliness rather than predefined thresholds based on calculated metrics. Even if machine learning approaches are now being used to address code smell issues and the findings are promising [58], when compared to static techniques, the ML branch is still under development and has plenty of room for improvement [49]. The number of studies examining the use of machine learning techniques is growing, but each one employs different models and techniques to accomplish this goal [25]. Machine learning approaches are now being used to solve code smell issues, and the results are encouraging. To construct a model, a machine learning classifier must first be drilled using a set of code smell instances. The models are then used to discover or detect code smells in novel or unusual situations. The produced model's power is determined by a number of factors connected to the dataset, including the machine learning classifiers, the parameters of the classifier itself, etc. [58] According to the researchers in [56], the model can be trained using data from the project under examination (within project strategy) or data from other software projects (cross project strategy).

Furthermore, there is a scarcity of empirical data to assist the creation of new research on machine learning approaches for detecting code smells [41]. Further development of empirical support can aid in understanding the shortcomings of machine learning approaches and taking steps to improve their performance by providing information on their benefits and gaps [49].

When a code smell is detected, it is suggested to do refactoring. Refactoring is a technique for changing the internal structure of a program without changing its behavior. Refactoring increases the code quality but has no effect on the system's behavior [59] [8]. High quality, high performance, cheap cost, reusability, implementation, and easy software development are all benefits of refactoring [9] [60].

2.1.1.1 Code Smell and Types

As defined earlier, Code smells are inherent features of software that might suggest a code or design fault that makes it difficult to evolve and maintain software, as well as trigger refactoring. They are closely linked to the technique of redesigning software to improve its internal quality [11]. A bad smell indicates a problem in the code that has to be addressed through refactoring [61]. So, they're not bugs; they only make it tough for software developers to understand the project's source code. Meanwhile, these code smells may make it difficult for software developers and maintainers to restructure and upgrade project source code [62]. Code smells aren't patterns to avoid; they're indicators that something has to be looked at further [63]. As a result, refactoring can be used to address these design flaws or structural issues in software [61].

Refactoring is the practice of altering a software system in such a way that it improves its internal structure while leaving the code's external behavior unchanged [8]. Refactoring's primary goal is to improve the design of current code [8] [64]. By modifying the structure of the code/design without changing the general behavior of the system, refactoring enhances several attributes of code/design such as maintainability (understandability and readability), extensibility, and so on [57]. Additionally, according to Mantyla, M., etal [64], the enormous complexity of a software system may obstruct its further development. Refactoring is just a method of ensuring that future development is possible. The researcher Fowler, M. etal [8] tries to convince people to refactor with the following arguments: -

- **Refactoring makes software easy to understand.** This is undoubtedly true, as one of the purposes of refactoring is to make software more understandable and source code self-documenting. Of course, there are times when developers argue over which type of code is the easiest to comprehend. In most circumstances, though, notions about what constitutes good programming style and good design should be consistent.
- **Refactoring helps you find bugs.** If we agree that refactoring improves understandability, this can be accepted by common sense. However, supporting the thesis with empirical evidence would be beneficial.
- **Refactoring helps you program faster.** This notion is reinforced by the rules of software evolution, which state that as software systems become more sophisticated, they become more difficult to design.
- **Refactoring improves the design of software.** Because effective software design is nearly always straightforward to understand, this argument goes hand in hand with Fowler's first argument.

Smells, using metaphor, are the symptoms of possible diseases, and refactoring procedures may be used to heal the diseases and eliminate their symptoms [11]. It does not imply that all code smells must be eliminated; it is dependent on the system. When they must be removed, however, it is preferable to do it as soon as possible. We must locate and detect smells in the code if we want to eliminate them; tool support for their identification is especially useful because many code smells go undiscovered while programmers are working [11]. Additionally, according to the researcher at [57], the majority of code smell detection solutions rely on static analysis and code metrics, and do not take into account aspects such as system size, language structure, or context. To put it another way, any design-based refactoring requires the tool to comprehend the code's real semantic purpose.

There are different kinds of code smells. Here are the most common and well-known code smells identified by Fowler, M. etal [8]. The researcher has introduced 21 kinds of code smells which have been used by different researchers. They are listed and described shortly as follows:

Table 2. 1 Code smell with their description

No	Code smell	Description
1	Alternative class with different interfaces	A situation in which a class can interact with other classes yet their interfaces are different.
2	Comments	Poor code structure is compensated by the usage of comments.
3	Data class	A data-only class that doesn't have any logic.
4	Data clumps	Data items that are frequently found together.
5	Divergent change	When a class must be updated each time another class is updated.
6	Duplicate code	Code that duplicates the functionality of another piece of code.
7	Feature envy	A method that is more concerned with the qualities of other classes than with the properties of its own.
8	Inappropriate intimacy	When two classes are inextricably linked.
9	Incomplete library class	When a program uses a library that isn't complete. Large Class: a class with a lot of instance variables or methods that tries to perform a lot of things.
10	Lazy class	A class that isn't doing its job properly and should be eliminated.
11	Long method	A lengthy procedure that is difficult to comprehend, modify, or extend.
12	Long parameter list	A lengthy and difficult-to-represent parameter.
13	Message chain	A series of calls from one object to another that do not provide any new functionality.
14	Middle man	When one class delegated a large portion of its behavior to another.

15	Parallel inheritance hierarchies	A situation in which there are two parallel class hierarchies that are linked.
16	Primitive obsession	It represents the use of primitives rather than small classes, which makes it less useful and reusable.
17	Refused bequest	A child class completely supports the implementation of its parent class.
18	Shotgun surgery	When a class is changed, all other classes must be changed as well.
19	Speculative generality	When unneeded code is written in anticipation of future software modifications.
20	Switch statements	Instead of polymorphism, type codes or run-time class type detection are used.
21	Temporary field	A variable in the class is only utilized in specified scenarios.

Later in 2003, Mantyla [65] was able to categorize the above code smells identified by Fowler et al. into 7 categories as presented below:

- **The Bloaters:** The code or classes in this category have been enlarged to the point where they are difficult to work with. These smells don't appear right away; rather, they build up over time as the program progresses, especially if no one tries to get rid of them. The first type of code smell in this category is the *Long Method,* which contains an excessive number of lines of code, makes it difficult to alter, replace, and recognize. Splitting this procedure into several methods is the best answer for this smell. The second type of code smell is *Large Class*. When a single class tries to do too much, it becomes a large class, which usually includes multiple instances and responsibilities. This smell makes the class more difficult to reuse and maintain. The simplest way to get rid of this smell is to divide the class and use class extraction. The third is *Primitive Obsession*. In some cases, tiny classes should be used instead of primitive types. For simple tasks, primitives, such as particular strings for phone numbers, ranges, and currencies, are used instead of little objects. The fourth type which is *Parameter List*, occurs if a method has more than four arguments, making parameter lists more difficult to distinguish and use, as well as inconsistent. The final type is category is *Data Clumps*. Various areas of the code, for example, parameters to connect to a database, may have comparable groups of variables. Data clumps should be split up into separate classes.

- **The Object-Orientation Abusers:** All of the smells in this category are the result of improper or insufficient object-oriented programming approaches. *Switch Statements* are the first form of bad smell in this group. This smell appears when the code has a series of if statements or a sophisticated switch operator. *Temporary Field* is the second form of bad smell in this group. Temporary fields are typically established for usage in algorithms with a large number of parameters. As a result, rather than defining a huge number of parameters, the programmer constructs fields in the class for these data. These fields are solely used in the algorithm and are otherwise unusable. This type of smell is tough to detect. The removal of this odor improves the readability and organization of the code. *Refused Bequest* is the third type of bad smell in this group. When programmers build inheritance between two completely different classes, but the subclass only uses a few of the superclass's methods and properties, the Refused Bequest smell appears. Delegation, rather than inheritance, is the greatest method to deal with this smell. *Alternative classes with different interfaces* is the fourth type of bad smell in this group. This code smell occurs when programmers build two classes with similar functionality but different names for their methods.
- **The Change Preventers:** These smells emerge when you need to update anything in one part of your code but also need to change a number of other parts. As a result, the software development process is more complicated and costly. There are three types of bad smells in this category. *Divergent Change* is the first type of bad smell in this category. When multiple changes are done to a single class, this bad smell develops. Splitting the class's conduct is the greatest technique to get rid of this smell. When various classes have the same behavior, for example, the classes should be unified by inheritance. This will improve the code's organization and decrease code duplication. *Shotgun Surgery* is the second form of bad smell in this group. When a single change is done to numerous classes at the same time, this is known as shotgun surgery. Because one task has been divided among a vast number of classes, this smell occurs. Moving the existing class behaviors into a single class is the best method to get rid of this smell. This will make the code more organized, decrease duplication, and make it easier to maintain. The third type of bad smell in this category is

Parallel Inheritance Hierarchies. When you build a subclass for a class and then realize you need to make a subclass for another class, you get this smell [1].

- **The Dispensables:** These smells appear when a piece of code is no longer needed and might be deleted, making the code cleaner, more efficient, and easier to understand. This category includes six different forms of bad smells. *Comments* are the first sort of bad smell in this group. This smell appears when the script contains a lot of explanatory comments. The *Duplicate Code* is the second form of bad smell in this category. When the same or extremely identical code appears in multiple places of a program, it causes the program code to become big. This bad smell can be eliminated by encapsulating the duplicated code in a new method. The third foul odor in this category is *Lazy Class*, which is a worthless class in which every class developed requires effort and time to learn and maintain. Eliminating these classes is the greatest approach to get rid of the smell. The fourth type of bad smell in this group is *Data Class*. A Data Class is a class that merely includes fields and rarely has any logic. The Data Class contains getters and setters' methods for fields. *Dead Code* is the fifth sort of bad smell in this group. This happens when a piece of code is never executed. *Speculative Generality* is the sixth type of bad smell in this category. This happens when a parameter, field, method, or class isn't used. The reason for this bad smell is that code is occasionally written to support anticipated future features that are never deployed. As a result, the code becomes more difficult to comprehend and maintain.
- **The Encapsulaters:** such as Message Chains and Middle Man, are smells that deal with data transmission or encapsulation.
- **The Couplers:** All of the smells in this category either lead to excessive class coupling or demonstrate what happens when coupling is substituted by excessive delegation. There are six types of bad smells in this group. *Feature Envy* is the first type of bad smell in this category. When a method accesses the data of another object more than its own data, it emits a bad smell. This usually happens when fields are moved to a Data Class. If this happens, the data operations should also be relocated to this class. The second type of bad smell in this group is *Inappropriate Intimacy*. This happens when one class does its work by using the internal methods and properties of another class. The third type of bad smell in this group is *Message Chains*. Message Chains occurs when a client wishes another object, that object requests another object, and so on. The fourth type of bad smell in this

group is *Middle Man*. When a class performs only one action and delegated tasks to another, it is known as Middle Man. This smell could be the result of a hasty removal of Message Chains. *Incomplete Library Class* is the fifth kind of code smell in this category. When libraries are no longer able to meet the needs of their users, this occurs.

- **Others:** includes others that do not fall into any of the other categories mentioned above.

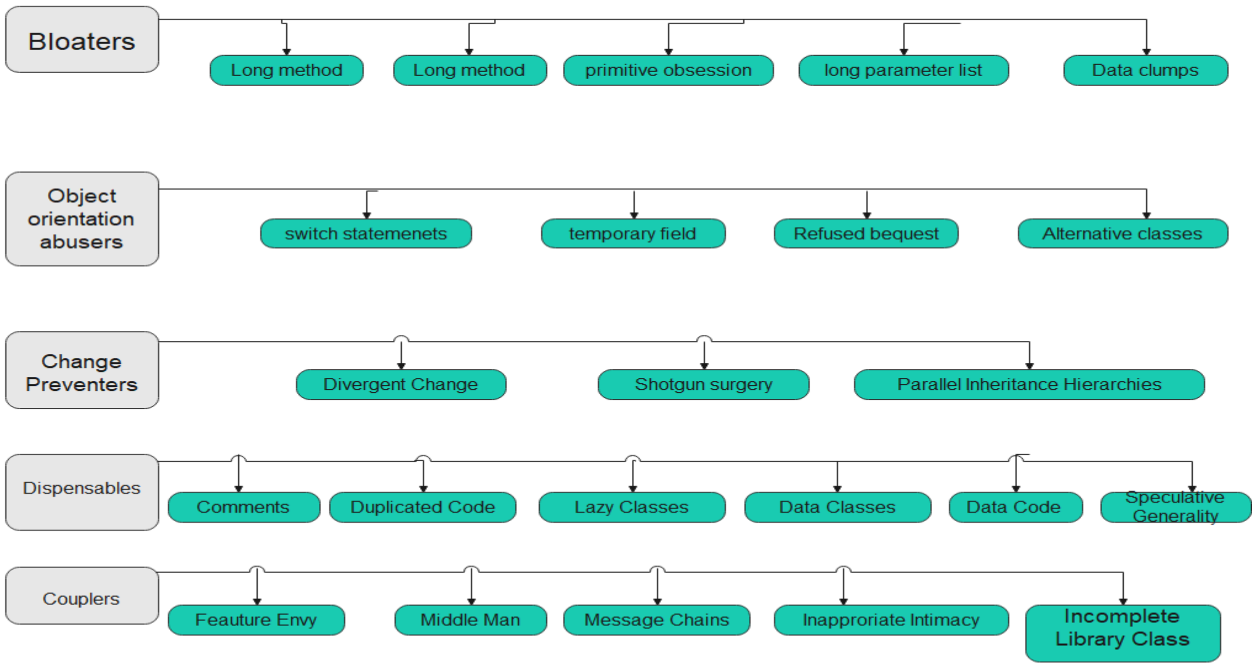


Figure 2. 1 Code smell categories [56]

The occurrence of code smells has the ability to reduce the extendibility and maintainability of software products. So, it is better to handle them properly and to do so, they have to be detected as fast as possible. Having this in mind, there are a number of studies carried out on the detection of code smells. Those studies propose different kinds of approaches to wards the detection of code smells. Some of them propose the use of manual approach in code smells detection, others propose the use of several automatic detection tools. Different approaches have been given more attention in past decades. But, due to the limitations of those approaches, recent studies have extensively examined the use of machine learning approaches.

2.1.2 Machine Learning Techniques

The earlier the code smells are detected, the less will be the cost of refactoring and the better the software quality will be. Therefore, the detection step plays a vital role in improving the results of the other steps and by that on the performance of the software refactoring. Better machine learning techniques are a good solution to cope with the ambiguity caused by the lack of consensus (conflicting perceptions of developers conduct to subjective code smell interpretations) [48]. Meaning, to deal with the problem of subjectivity, various machine learning approaches have been presented that can learn and discriminate the properties of smelly and non-smelly source code parts (classes or methods) [66].

Machine learning is a field of study in which computers may learn and complete tasks without being explicitly programmed [67]. It is a branch of computing algorithms that is constantly developing and aims to replicate human intelligence by learning from the environment. In a sense, these algorithms are "soft programmed" in that they continuously improve at doing the required goal by dynamically altering or adapting their architecture. Training is the adaption process where samples of the input data are given together with the intended results. The algorithm then refines itself to its best ability so that it can generalize and create the desired outcome for unknown, previously unexplored data. The "learning" component of machine learning is training. [68]. It can be classified in to three broad categories supervised, semi- supervised and unsupervised. The supervised machine learning classifies a given instances in to previously known class label. Hence the class label of each instance is known from the beginning where as in the unsupervised machine learning there are no predefined class labels. Class labels are decided after the natural grouping of similar populations has been made first. The third semi-supervised machine learning technique is a hybrid approach that incorporates the features of both supervised and unsupervised machine learning techniques where there is a portion of the dataset containing an already labeled instances and there is also some portion containing the unlabeled instances/ data.

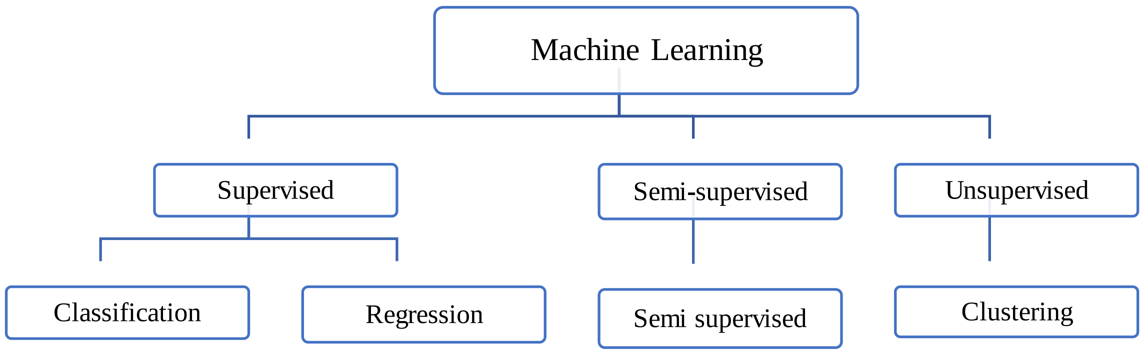


Figure 2. 2 Machine learning taxonomy [48]

2.1.2.1 Supervised Machine Learning Technique

The task of supervised classification entails employing algorithms to teach a machine the relationship between cases and class labels. The supervision comes in the form of previously labeled instances, from which an algorithm constructs a model to predict the labels of new instances automatically [66]. According to Fernandes, E. etal [39], Supervised Machine Learning methods identified by Kotsiantis, S.B. et al., at [44] like Decision tree, learning set of rules, single layered perceptron, Multi layered perceptron (MLP), Radial Basis Function (RBF) network, Naïve Bayes, Bayesian Network, Instance Based Learning and Support Vector Machine are the most used method for detecting code smells. But some of the supervised machine learning techniques according to [58] are listed below.

- **Multilayer Perceptron (MLP)** [69] [70] is a type of artificial neural network (ANN) that consists of an input layer, at least one hidden layer, and an output layer. Every node is a neuron with a nonlinear activation function that has a weighted association with other nodes in the next layer. MLP uses the back propagation technique for all of its training.
- **Support Vector Machines (SVMs)** [71] [72] are supervised learning models which will be used for classification or regression. Based on the notion of structured risk minimization, the researcher at [71] defined SVM and empirical error minimization and geometric margin maximization are the goals of SVM.
- **Radial Basis Function Networks (RBFs)** [73] [74] is a form of neural network that consists of three layers: an input layer, a hidden layer, and a linear output layer. RBF networks are utilized for categorization, function approximation, and system control. There

are three varieties of RBF networks: multiquadric, polyharmonic spline, and Gaussian RBF networks.

- **Bayesian Belief Networks (BBNs)** is a probabilistic graphical model that can be used to represent a set of variables and their conditional dependencies. The probabilistic dependencies among the associated random variables are represented by the edges linking the nodes in this model [75]
- **Naive Bayes (NB)** according to the researchers in is a supervised learning algorithm that uses the Bayes algorithm with the "naive" assumption that each pair of characteristics is conditionally dependent [74] [76].
- **Linear Regression (LR)** (LR) is a modeling technique that uses linear predictor functions to find the correlation between the goal and independent variables in datasets [77] [78].
- **Random Forests (RF)** is a supervised learning approach that contains a large number of unpruned classifications or regression trees, each of which is based on the values of a random vector investigated separately and with the same distribution across the forest. Both classification and regression issues can be solved with RF [79].
- **Multinomial Naive Bayes (MNB)** Multinomial naive Bayes is a form of NB that was first used to classify text [76].
- **Decision Tree (DT)** is one of the most popular supervised learning methods for regression and classification The C4.5 algorithm is the most widely used method for producing decision trees [80].

2.1.2.2 Unsupervised Machine Learning Technique

As discussed above, supervised machine learning techniques can function in an environment where the class labels are already specified. So, having the dataset that contains unlabeled instances, there is no guarantee for the supervised techniques to undergo the classification task. In such scenarios unsupervised learning techniques plays a vital role in assigning those instances in to their proper groups.

Code smell datasets may contain labeled instances. But, still according to different studies on code smell detection, the datasets being used by the researchers contain a large portion of unlabeled data. Meaning, a given dataset represents the occurrence of a single type of code smell and the class labels will be **smelly** (containing that specific smell type) and the rest are labeled as **non-**

smelly. But in fact, the second category which has been blindly labeled as non-smelly, might comprise other smell types. This might potentially influence the prediction performance of a given machine learning technique because prediction result will be biased towards the larger class (class covering the majority of instances). As a result, unsupervised procedures are essential for reducing the bias produced by the unlabeled cases and categorizing the unlabeled instances into appropriate groupings. As a result, the non-smelly parts of the dataset can be categorized according to their degree of similarity using the clustering method. Clustering is a machine learning approach that groups data points together. A clustering technique can be used to classify each data point in a certain group given a set of data points. There are a variety of clustering strategies to choose from. According to Luiz, F.C. [49], hierarchical and partitional clustering techniques are the two types of clustering algorithms. Hierarchical clustering techniques create nested clusters in a cyclical manner, whereas partitional clustering algorithms find clusters concurrently.

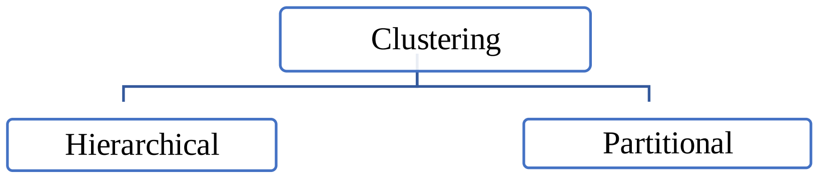


Figure 2. 3 Clustering technique

2.1.2.3 Semi-supervised Machine Learning Technique

The other semi supervised technique can be used when a dataset contains both labeled and unlabeled instances. Semi-supervised learning can be applied in place of supervised learning, using unlabeled data for training [81].

Machine learning approach can be applied in different disciplines. It can be applied in health industry, financial industry, educational industry, retail industry and different others. Its application on these different disciplines has been getting attention these days. This is because there is an increasing awareness on the advantage of tying them to their business process. Machine learning approach has the ability of upgrading performance of a given business in a way that enables them in a better decision making and better understanding of their domain easily.

Among those different fields, Software engineering is one area. This study focuses on the application of machine learning approach in the software industry. One among the major challenges in the software industry is the occurrence of code smells, which indeed have the ability to hinder a quality of a given software or even degrade the system's performance. Those code smells should be detected as early as possible to reduce unnecessary maintenance cost. It is believed that machine learning approach is best way to enable the early detection of occurrence of code smells.

2.1.3 Evaluation Metrics

The performance of machine learning techniques can be evaluated using different metrics. Some of them are true positive rate, false positive rate, precision, recall, f-measure and accuracy. The detail information on how those metrics evaluate the performance is presented as follows.

- **Correctly classified instances: -** Is the total number of instances that are labeled correctly under their true class.
- **Incorrectly classified instances: -** Is the total number of instances that are labeled incorrectly into other classes by the classifier.
- **Time taken to build the model: -** Is the amount of time a given algorithm takes to train it-self and able to make a prediction accordingly.
- **Confusion Matrix:-** shows the true positives, false positives, true negatives and false negatives.

		True Class	
		Positive	**Negative**
Predicted Class	**Positive**	**TP**	**FP**
	Negative	**FN**	**TN**

- **TP Rate: -** Is the percentage of instances that are actual positives that are labeled correctly as actual positives. TP Rate is also referred as recall or sensitivity. It incorporates all the True positives and False Negatives. It is calculated by

$$\frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

- **FP Rate: -** Is the rate of instances that are in fact negative but mistakenly classified to be positive by the classifier. These instances are not actual positives. It incorporates all the True Negatives and False positives. The FP Rate is calculated by

$$\frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

- **Precision**: - Is the ratio of true positives to the total number of the true positives and false positives. It is calculated by

$$\frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

- **Recall: -** the value is similar to True Positive Rate.
- **F-Measure: -** is a performance measurement technique that incorporates the properties of both precision and Recall. F-Measure is calculated by

$$\frac{2 * (\mathrm{Precision} * \mathrm{Recall})}{(\mathrm{Precision} + \mathrm{Recall})}$$

- **Accuracy: -** Is the percentage of correctly classified instances (True positives and True Negatives) to the overall predictions made. It is calculated by

$$\frac{(TP + TN)}{(TP + TN + FP + FN)}$$

2.1.4 Systematic Literature Review

A Systematic Literature Review (SLR) is a defined as a process for locating, evaluating, and interpreting published literature in order to examine a certain research issue or phenomenon [82]. Since the main aim of the SLR is to find out the relevant and available published literatures and studies that enables to answer the research questions, a series of steps need to be performed towards achievement of study goal. So, a clear and traceable search strategy should be followed. The search strategy consists of a series of steps such as search term formation, resource identification, selection process and quality assessment. Finally, the information from the selected papers will be synthesized in order to extract the answers to the mentioned research questions.

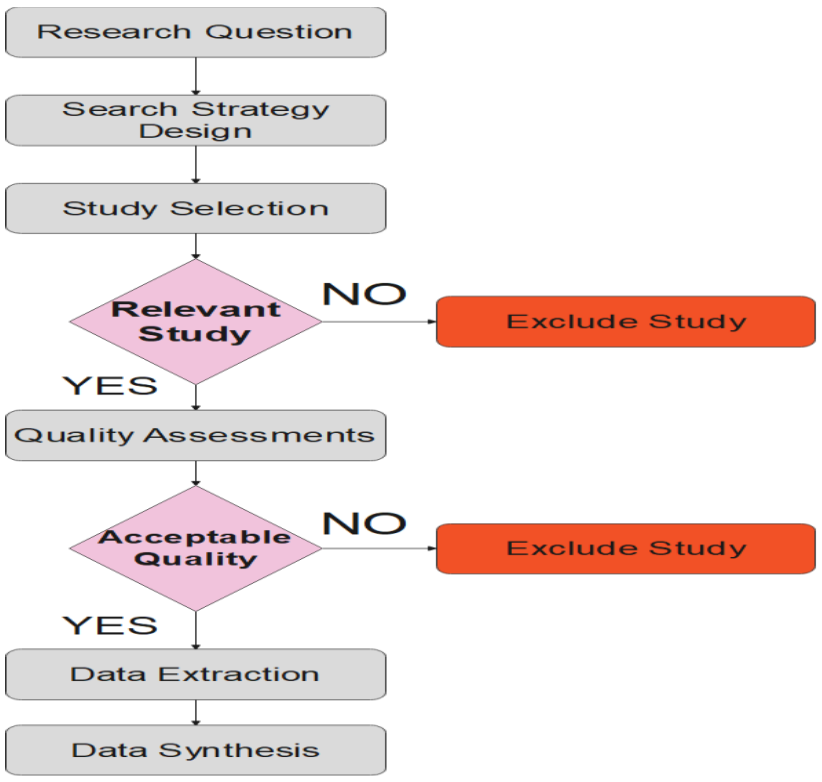


Figure 2. 4 The Systematic Literature Review (SLR) process phases adopted from [58]

As the above figure depicts, the basic steps in any Systematic Literature Review are defining the research question, designing a search strategy, performing study selection, setting quality assessment, data extraction and data synthesis. The steps are described as follows: -

- **Research Question**

Defining the research question is the first stage in any systematic literature review. The research questions that the study aims to answer should be clearly specified. Accordingly, the next stage of the SLR would be held based on the research questions formed. All the important keywords will be constructed in a way that they can enable to answer the research questions of the study.

- **Search Strategy**

The second stage in Systematic Literature Review is search strategy. This search strategy contains a series of three steps. These steps are search term identification, search term formation and resources to be searched. In the search term identification stage, a set of words that are synonyms with every core point of the research question will be explored. Then, the appropriate search terms will be merged together in order to form a complete search term in the second stage search term formation. Once the search term is formed, the third step is finding the place to apply the search term in order to retrieve the necessary documents. Hence, the selection of digital libraries to dig from will be performed. Then, the search term will be applied in the selected digital libraries and accordingly, all documents related to the search term will be retrieved.

- **Study Selection**

The third stage which is the study selection stage focuses on the filtration of the most important documents from the less important ones. Accordingly, the documents that are going to have a high relevance towards answering the research questions specified will be selected. In order to do so, a set of exclusion and inclusion criteria will be applied to filter the retrieved documents. Finally, all the documents that can satisfy the criteria will pass to the next stage, quality assessment.

- **Quality Assessment**

This is the fourth stage where a quality of the selected documents will be evaluated by peer academicians or domain experts. The documents will be evaluated for their true representativeness of the specific issues raised as a research question. Accordingly, the information from the papers that pass this stage will be utilized.

- **Data Extraction**

This is the fifth stage in systematic literature review. Hence, the information with respect to the research questions will be extracted and utilized from the selected papers. The information extracted then will be used for synthesis in the final stage.

- **Data Synthesis**

This is the final stage in any systematic literature review where the answer for the previously specified research questions will be provided. The implication of the information from the previous step will be reported in this stage.

2.1.5 Replication Study

The process of repeating a research study, usually with different contexts and individuals, to evaluate if the original study's fundamental conclusions can be applied to different things or situations is known as replication [66]. Replication study aims at repeating a study’s data and methodology in order to prove or disprove whether a result of a given study can have the same effect in another environment. A researcher can use the same dataset to repeat a study and see if other methodologies can affect the dataset in the same way. Similarly, a researcher can use the same methodology used by the reference work and apply it on other dataset to check if the given methodology can influence the new dataset similarly. Replication study enables a methodological or subject dependency comparison between of a reference work and one’s original work.

The task of replication study requires the selection of a reference work with which one’s own work can be compared. Therefore, the researcher is required to select a reference work in to which a modification can be made.

2.2 Related Work

Kaur, A. et al., at [83] proposed J48 technique to detect the occurrence of pattern-smell pairs that co-exist and those that don’t contain any smell in them claiming that design patterns and code smells are too related in a way that the presence of design pattern contributes for the absence of code smells and vice versa. For this study, they first selected open-source systems (eclipse version 3.6 and 3.7) then they selected tools for extracting smells and patterns from the selected source code and finally prepared datasets containing classes extracted by smell detection tools and classes extracted by pattern detection tools. Then j48 is selected as learning algorithm. The study have used an integrated platform in such a way that i-plasma is used for code smell detection and web of patterns is used for design pattern detection. The study runs 10 different experiments on the selected datasets and two parameters were used to measure the performance of the proposed models and those are PRC (Precision-Recall curve) and ROC (Receiver Operating characteristic Curve). Finally, the main focus of this study is finding the basic relationships among design patterns and code or bad smells and they have tried to detect multiple disharmonies in the code. The code smells detected in the dataset are Brain class, Data class, God class, Request pattern bequest and schizophrenic class. On the other side, the patterns considered are Adapter, Bridge, Template and singleton. Hence after running a set of experiments on the selected patterns versus code smells, they were able to conclude the following major findings.

- Singleton design pattern shows no presence of bad smells.
- The code smell “request pattern bequest” shows no presence in the presence of design patterns.
- With respect to the least smell presence, God class exhibits the least presence in the presence of Adapter, template and Bridge pattern.

The study of Kaur, A. et al., at [83] focuses on the detection of design pattern and code smells existence patterns. Whereas, this study focuses on the application of ML techniques for the detection of existence of code smells.

Hadj-Kacem, M., et al. at [48] tried to detect code smells using Deep learning claiming that deep learning can give better performance compared to other machine learning approaches. In this study, the researchers have used a combined approach containing a supervised and un-supervised techniques for the detection of the four selected method level and class level smells. This study uses four datasets that were adopted from Fontana, F.A., et al, at [22]. The four datasets represent the first two class level code smells (God class and Data class) and the other two method level code smells (Feature envy and Long method). The unsupervised learning that is used for dimensionality reduction is Auto-encoder which is an unsupervised neural network that is trained with feed forward and back propagation algorithms [84]. The supervised learning used for classification was Artificial Neural Network (ANN) which consist of input, hidden and output layers, in which the neurons are connected, and for each connection, weights are set. Because it's a binary classification, the output neuron will determine whether or not it's a code smell based on its kind [48]. So, the major contribution of this paper is concluded as two-fold where the first is they proposed a combined detection approach using two deep learning techniques; Auto-encoder and Artificial neural network. The second is comparison of the combined approach and the basic classifier showing that there is a significant performance improvement when the number of features is reduced using the Auto encoder and that dimensionality reduction plays an important role in achieving better result in the study. Finally, the performance was evaluated using Precision, Recall and F-measure. The final result for the four smell types is presented as follows:

Table 2. 2 Performance summery of the Deep learning approach by [39]

Code smell	Precision	Recall	F-measure
God class	99.28%	98.58%	98.93%
Data class	98.92%	98.22%	98.57%
Feature Envy	96.78%	96.09%	96.44%
Long method	96.78%	97.83%	97.30%

Unlike this study, the work of Hadj-Kacem, M., et al. at [48] don’t consider the detection of existence of multiple smells in a single dataset. Each of the datasets used by the researcher are independent and experiments were done on these independent datasets (for each smell type).

Luiz, F.C. et al., at [49] tries to identify code smells with machine learning techniques. The aim of this study was development of a mapping study of machine learning techniques on the selected code smells and an experiment was conducted on a standardized dataset (Landfill) that reflects a real project scenario. The researcher performs a systematic literature review (SLR) and undergo the process from a clear search strategy up to selection of relevant papers using a set of inclusion criteria, exclusion criteria and snowball technique as well. Then after a result of the SLR has been found, a state of art based on the selected machine learning techniques has been applied to the targeted dataset. Some of the techniques used by this study are Association rule mining, SVM, Text based, Genetic algorithm, Clustering, Decision Tree, Random Forest, Semi-supervised, Nearest Neighbor, Linear Discriminant Analysis and Naive Bayes. The smells covered by this study are BLOB, Divergent change, Duplicated code, Feature envy, God class, Large class, Long method, Long parameter list, Message chain, Middle man, Shotgun surgery and Speculative generality. The performance of each technique was measured by Precision, Recall and F-measure. Finally, the results indicate that in terms of f-measure the best average performance was provided by Decision Tree, followed by Random Forest, Semi-supervised and Nearest Neighbor techniques. When analyzing the techniques by precision, Linear Discriminant Analysis followed by Association Rules, Semi-supervised and Decision Tree present the highest performance. According to the result of recall, Naive Bayes classifier performing worst in general. Also, Random Forest for Middle Man, Decision Tree for Message Chain and Text-Based technique for Long Method shows a bad performance. In general, the technique used for Long Method performed closer to the original experiments, but 34% worse than it, while the rest were 50% outperformed by the original works and some even performed below 86% compared to the original work.

Even if Luiz, F.C. et al., at [49] have performed a very detail SLR and a number of experiments, they have used 12 different datasets representing 12 smell types. They didn't consider the existence of more than one smell type in a single smell instance. Whereas in this work, two independent datasets have been merged to explore the existence of two code smells in a single dataset. Additionally, they have directly applied the ML techniques on a highly imbalanced dataset which is an issue in most smell datasets. The SLR is also performed in a very recent works compared to the SLR in reviewed work.

Guggulothu, T. et al., at [66] in 2019, tried to detect code smell using multi label classification approach, their claim was that the existing machine learning techniques can only detect a single type of smell in the code element and this doesn't correspond to a real-world scenario; therefore, they proposed multi label classification methods to detect whether the given code element is affected with multiple smells or not. They have considered two method level datasets containing long method and feature envy which were initially adopted from Fontana, F.A. et al., [22]. Those datasets first contain single label methods and the researchers merged those separate datasets so that the final dataset contains a multi label method level smells. They build the dataset MLD (Multi Label Dataset) in a way that common instances are added to MLD with their corresponding two class labels and the remaining instances of each single class label dataset are joined into MLD. Then after performing the necessary transformation on the dataset, they have applied five tree-based classifiers namely, B-Random Forest, Random Forest, B-J48 Unpruned, B-J48 Pruned, J48 Unpruned and the best performance according to F-measure, ROC area and Accuracy is gained by Random Forest with the results 96.0%, 97.7% and 95.9% respectively.

The research conducted in the study of Guggulothu, T. et al., at [66] is relatively similar with this work than the rest. It has considered existence of two smells. They have adopted a multi label dataset. The multi class dataset was transformed using three problem transformation techniques namely, binary relevance, classifier chaining and label chaining. Their study indicates that the classifier chaining and label chaining results better detection than the binary one. The researchers in the study have undergone the experiments on an imbalanced dataset containing a set of less relevant attributes. They have applied tree based algorithms (Random Forest and J48). In this work, a multi class dataset was used in order to represent a real label correlation. Additionally, this study has tried to apply four different ML algorithms.

In the work of Azeem, M.I. et al., at [56], systematic literature review and meta-analysis was performed based on machine learning techniques for code smell detection. The researchers claim that even if heuristic-based code smell detectors have been carefully studied by different research communities, there is still a noticeable lack of knowledge on the use of machine learning to detect code smells using machine learning techniques and whether there are point of improvements to allow better detection. So, the major objective of the study was to explore the use of machine learning approaches to detect code smells and hence performed a systematic literature review. They have considered papers published from 2000-2017 and using the SLR process they were able to identify 15 papers in which each of them actually adopted machine learning techniques. In this study they have tried to find answers for code smells considered, machine learning techniques adopted, evaluation strategies and a meta-analysis on the performance achieved by the models proposed so far. The final result of the analysis shows that God Class, Long Method, Functional Decomposition, and Spaghetti Code are the smells that have been extensively studied in the selected papers. The most commonly used machine learning algorithms for code smell detection were Decision Trees and Support Vector Machine. Additionally, in terms of performance, JRip and Random Forest were the most effective techniques for code smell detection. The study also performed a meta-analysis from which the researchers have also tried to infer the following conclusions that proper selection of metrics (independent variables) has an influence on the performance of prediction models by almost 29%. Additionally, the study has clearly indicated that in most of the studies, prediction models were experimented in both with–in and cross project setting but 33.33% of them were conducted on cross project scenario and they have noticed that cross models are 19% less effective than with–in project models. The other thing is, the researchers have mentioned that few studies were performed on large scale which they think they find it hard to generalize the reported findings and only one primary study investigated the performance of machine learning techniques on a manually built dataset which reports the existence of a comprehensive set of code smells.

The work of Azeem, M.I. et al., at [56] is based on the application of systematic literature review and the SLR was performed on the years 2000 to 2017. But this study focuses on a very recent papers which the researchers have not covered. An experimental work on the detection of code smells is also out of scope in the reviewed work.

Al-Shaaby, A. et al., [58] tried to perform a systematic literature review to systematically review the studies carried out to detect code smells using machine learning techniques from the year 2005 to 2018. In order to carry out the SLR, the researchers have followed a general guideline defined by Kitchenham, B. et al., at [85]. They have conducted their search in five online databases to extract the relevant studies. They have conducted a set of exclusion, inclusion criteria and quality assessment to obtain the primary studies. Then finally come up with 17 primary studies that focus merely on the application of machine learning techniques to detect the presence of code smells. The researchers undergo this study believing that the final result will provide knowledge about code smells detected, machine learning techniques being used, datasets used, accuracy measures and most commonly used tools. Finally, after analyzing those 17 primary studies, the researchers have concluded that SVM, J48, Naïve Bayes, Random Forest, SMO and JRip were the most widely used techniques in 2013, 2016 and 2018 while in 2017 Artificial Neural Networks were the only techniques that have received attention. They have also mentioned that according to their findings, it is only the supervised machine learning technique that has been applied to the area of code smell detection. With respect to the smells detected, they have noticed the occurrence of around 28 types of smells in the selected papers. 12 of the papers out of 17 detected more than one smell type while the rest five have detected one smell type. They have also tried to analyze the datasets that were used for code smell detection, dataset name, size, type (if they have used commercial, student or open source), availability, and language. Accordingly, fourteen studies used open-source systems, while the rest three studies have used industrial systems. Only two studies have used systems written in C and C# the other studies have used systems written in java. Most studies used software metrics as the independent variables except one study that have used textual metrics called string tokenization. With respect to the tools, WEKA and Tensor Flow implemented in Python were the most commonly used tools to implement the machine learning algorithms.

Al-Shaaby, A. et al., [58] is based on the application of systematic literature review. This study doesn't apply any experimental work on the detection of code smell.

Di Nucci, D. et al., at [86] has tried to assess the application of machine learning algorithms in detection of code smells in their study. The main aim of their study was to investigate if there is still enough room for improvement in the topic of code smell detection using machine learning algorithms. The baseline for their study was the work done by Fontana, F.A. et al., at [22] that has applied around 32 different ML algorithms to detect four code smell types. Fontana et al. have conducted a large-scale study and consider two class level smells and two method level smells which are Data Class, Large Class, Feature Envy and Long Method and they have reported that most of the classifiers used by the study exceeds the performance 95% with respect to both accuracy and F-measure. The main reason for going through Fontana's work according to the researchers was that they found a number of possible limitations in the study that might threaten the generalizability of the final results and conclusion. According to the researchers, even if Fontana et al. have tried to propose the use of machine learning techniques, which mainly solve the high rate of human and tool subjectivity by providing the learning algorithms, the ability to differentiate between smelly and non-smelly instances in code elements, the performance of machine learning algorithms is still highly dependent on multiple grounds. Those grounds as to the researchers can be the nature of the dataset that is constructed, the improper generalization of other possible class labels in to one class label which can cause bias, the ratio between the class labels (there might even be a non-realistic balance between the class labels) and a strongly different distribution of the metrics between the two groups of instances. Hence, they have tried to critically investigate the work of Fontana, F.A. et al., at [22]. They have proposed a carry out study on the use of machine learning algorithms for code smell identification, with the goal of addressing the issue of metric distribution of smelly and non-smelly elements across diverse datasets. The dataset used incorporates code elements influenced by various forms of code smells, with a less evenly distributed distribution of smelly and non-smelly instances and a smoother boundary between the metrics distributions of the two groups of instances, resulting in a more realistic scenario [86]. Finally, their finding shows that the high performance reported by the researchers at [22] was not due to the mere fact that capabilities of the machine-learning techniques used for code smell detection is high. Rather, what contributes to this higher result according to their perspective was the specific dataset employed by the researchers. While testing code smell prediction models on the revised dataset, the performance is up to 90% less accurate in terms of F-Measure than those reported by the researchers at [22].

The work of Di Nucci, D. et al., at [86] apples 32 ML algorithms on four independent datasets. It doesn't consider the introduction of existence of more than one smell type in a single smell instance.

Table 2. 3 Summary of the Machine learning-based detection approaches by different researchers

No	(Author, Publishing year)	Methodology adopted				No of systems or Datasets
		Type of Machine learning			Machine learning techniques used	
		Supervised	Unsupervised	Combined		
1	Kaur, A. et al.,2018 [83]	✓			J48	2 datasets (Eclipse version 3.6 and 3.7)
2	Hadj-Kacem, M.et al.,2018 [48]			✓	Auto encoder and ANN	4 datasets
3	Luiz, F.C. et al.,2018 [49]	✓	✓		Association rule mining, SVM, Text based, Genetic algorithm, Clustering, Decision Tree, Random Forest, Semi-supervised, Nearest Neighbor, Linear Discriminant	1 dataset (Landfill)

					Analysis and Naive Bayes.	
4	Guggulothu, T. et al.,2020 [66]	✓			Random forest, B-Random forest ,B-J48 Unpruned, B-J48 Pruned and J48 Unpruned	2 datasets (Multi-label dataset containing two method level smells)
5	Azeem, M.I. et al.,2019 [56]	Performed SLR and meta- analysis studies from 2005-2017 that focus on code smell detection using ML techniques.				
6	Al-Shaaby, A. et al.,2020 [58]	Performed SLR on code smell detection using ML techniques studies from 2005-2018.				
7	Di Nucci, D. et al.,2018 [86]	Replication on the reference work done by the researchers at [22] using 2 datasets.				

2.3 Contribution of the papers

The contribution made by all the researcher’s work mentioned above is remarkable. They have tried their best to improve gaps in the area of detection of code smells using machine learning techniques. Some of them have performed a thorough systematic literature review giving insight about the increasing demand of study on this topic. The researchers at [58] [56] [49] have performed their SLR on the papers from (2005-2018), (2005-2017) and (2002-2016) respectively. They all have presented their clear steps on how they have performed the SLR. The initial and basic step in any SLR process is the formation of keyword that enables a search in online databases. However, the way the researchers at [56] have constructed their keyword is relatively convincing. Because these researchers have followed a very clear SLR guideline proposed by another researcher. So, the set of steps which they have followed in order to construct their keyword is quiet representative than the other two researchers. With respect to the SLR, findings of the studies represent the result of papers from 2002 to 2018. Hence, further study is required to perform SLR on recent papers.

Additionally, the other researchers at [49] [86] [66] [48] [83] have tried to perform an experimental work on the application of machine learning techniques in the detection of code smell. Hence, they have adopted different machine learning techniques. Di Nucci, D. et al., [86] have replicated a reference work and applied 32 different algorithms from WEKA. The researchers were forced to replicate a study because they were uncomfortable with the results of the reference work and tried to see on what is behind this higher result. Then they were able to point out that it is the dataset that contributes to this high result. This significantly implicates that the dataset and its quality play a vital role in the performance of a given learning algorithm. Guggulothu, T et al., [66] , on the other hand introduced a new concept which is multi-label classification approach to detect whether a given code is affected by more than one smell. Hence unlike the other researchers, their dataset (which is prepared manually) contains different smells under a single instance and the rest were labeled as non-smelly. But these researches have used tree-based classifiers only and other classifiers might perform better. Luiz, F.C. et al. at [49], apart from the SLR, they have tried to replicate a reference work and applied a wide range of machine learning techniques on a wide range of smells. He has applied both supervised and unsupervised techniques. He has used a large-scale dataset to undergo his experimental work. The other researcher Hadj-Kacem, M. et al., [48] contributes to the body of knowledge by introducing the use of hybrid technique in the detection of smells. They have also showed that feature reduction plays a vital role in the performance of classification algorithm which the others do not. Finally, Kaur, A. et al., [83] used only one learning technique and the scope of their study was a bit different from this study. It focuses on the relationship between code smell and design patterns.

All the papers that have been reviewed are found to be very critical and have made a remarkable contribution to this study. Having the strength and weakness of each papers described above, this study will try to fill the gaps in the following forms: -

- ➢ As for the SLR, this study was performed on very recently published papers from 2017 to 2020 (which is not covered by the others) and will follow a clear step to perform the SLR.
- ➢ Regarding the experimental work this study has applied feature reduction in order to filter out the most important metrics to the detection of the specified code smells.
- ➢ Additionally, data balancing has been applied on the dataset in order to mitigate the unrealistic proportion of instances, which almost all smell datasets suffer from.

- Classification was made with different supervised techniques from different categories.
- This study has also considered the existence of more than a one code smell in a single instance of method. Hence, introduces a multi class dataset containing the existence of two method level code smells which has not been previously covered by other studies.

CHAPTER THREE

METHODOLOGY

This chapter presents the general research methodology followed in the study. Hence, the two broad research methodologies followed will be discussed. Additionally, the general architecture of the proposed work is also presented in this section.

3.1 Research Design

In order to achieve the objective of the study, this research has followed a secondary research approach for the first methodology (SLR) and a design science research methodology for the second methodology (Replication Study) which is the experimental work.

3.1.1 Secondary research

Secondary research is a type of research in which a summary, collation and synthesis of prior research is performed. Primary research is concerned with the generation of data, whereas secondary research is concerned with the analysis of data obtained from primary research sources. It uses a primary source to synthesize the information from them. Secondary research involves research material published in research reports and other relevant materials. These resources can be found in public libraries, educational institutions, websites, and data from previously completed surveys. In order to conduct secondary research, the following steps have to be performed.

3.1.1.1 Identify the topic of research

Identifying the area that needs to be researched is required before beginning secondary research. After that, a list of the research's characteristics and objective should be developed. Hence, in the SLR part of this study, first an initial understanding on the area of code smell detection using ML techniques is built. Then a set of research questions are set according to the nature of the research problem. Three research questions were formulated so that they can be answered at the end when the information from the primary sources is analyzed. Accordingly, the SLR tried to explore the code smells detected, the ML techniques adopted and the datasets used for code smell detection.

3.1.1.2 Identify research sources

Once the topic is chosen and the research questions were set, the next stage is to search for sources from which a researcher can dig out information. This stage is to narrow down the sources of information that will supply the most relevant facts and information for the research. Accordingly, in the SLR part of this study, three online data bases namely IEEE, Springer and ACM are explored. All the relevant primary research papers that are published in the year 2017-2020 are used.

3.1.1.3 Collect existing data

Once the data collection sources have been narrowed down, there is a need to check for any historical information that is closely linked to the subject. Data for research can be found in a variety of places, including educational institutions, public libraries, government and non-government organizations, and so on. In the SLR part of this work, documents from the mentioned sourced are collected.

3.1.1.4 Combine or compare

This stage focuses on the gathering of important information from the retrieved documents**.** This helps to analyze and synthesize the information from these documents and finally provide an answer to the research questions. In this stage, according to the intention of the research work, the information gathered can be either combined or compared in order to assemble data into a usable format be analyzed.

3.1.1.5 Analyze data

Finally, after combining or comparing the data or information, they should be analyzed in such a way that they can give meaning and enable to answer the research questions specified initially at the start of the research work.

3.1.2 Design science Research

Design science Research or Constructive Research, is a scientific method for creating items that benefit people [87]. According to the researcher at [88], design science research can be defined as a study that creates a new functional artifact to address a specific type of problem and examines its usefulness in solving specific type of problem. It focuses on the development and evaluation of artifacts that can improve the performance of existing artifact. These artifacts can be algorithms, interfaces, methodologies (such as models) and languages. This type of research gives explicit standards for evaluation and iteration with in projects. The major reason behind adopting a Design science Research methodology is that, this research methodology is proven to be a suitable methodology in computer related fields.

As a result, since the overall aim of this study is to develop an artifact (model) that can be able to properly detect the presence of code smells, this broad research strategy is required and suggested.

According to the researcher at [87], Design science consists of a set of processes for its accomplishment. The processes include problem awareness, suggestion, development, evaluation and conclusion. The general framework of design science research is depicted in the figure below.

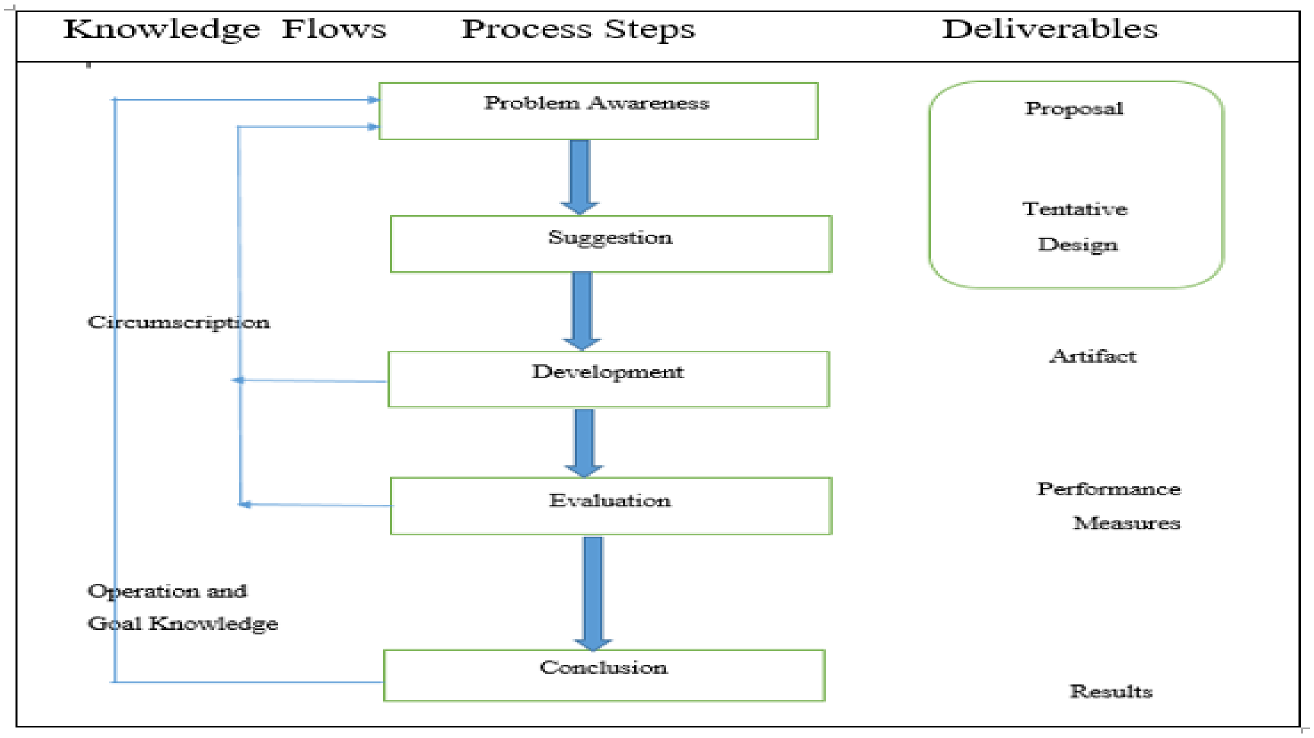


Figure 3. 1 Design science research methodology framework [87]

3.1.2.1 Problem Awareness

Problem Awareness is the first task in the design science research methodology adaptation process. This stage responds to the query, "What is the issue?". As a result, it's vital to define the study problem and show how a solution might be applied. An understanding of a problem in a specific domain is almost a half way to its solution. So, prior to further activity, a given research/study should be able to pin point the potential gaps in the specific domain. Accordingly, starting from the problem definition, the proper solution for the specific problem will be built. Hence, there are a number of ways used in order to identify the problem of a given area. This study has explored different published documents like journals and conference papers. As a result, it was able to recognize that code smell detection using ML techniques is a hot research area now a days with a number of gaps that need to be addressed by future studies. Even if a continuous effort has been done to address some issues, it has still an open area of improvement. Accordingly, this study was able to find out that the use of imbalanced dataset, the nature of the dataset used and selection of relevant metrics are some of the key factors influencing the performance of a given model.

3.1.2.2 Suggestion

In this stage of design science research, an answer to the question "How should the problem be solved?" will be provided. Hence, an alternative idea on how to solve a given problem will be provided. Additionally, the objectives (general as well as specific) of the research work and the specific methodology to be adopted will be set.

Accordingly, this study has proposed the use of a replication study with the aim of repeating the study of the reference work. Hence, an experimental work was proposed with a slight modification on the reference work.

3.1.2 .3 Development

Development stage is concerned with the creating an artifact that can solve the problem. These artifacts include models, methods or an instantiation in which a research contribution is embedded.

In this stage, the research has tried to design a machine learning model. Hence, it describes the dataset used, dataset formation, data preprocessing and selection of algorithm in the research as follows.

Dataset Preparation

The datasets used in this study are adopted from the reference work [66]. Two code smell datasets containing the existence of Long Method and Feature Envy have been used. Then, a multi class dataset has been formed. In order to form the multi class dataset the common instances from those two datasets were used.

The original datasets (the independent Long Method and Feature Envy) contain 82 software metrics and a total of 420 instances in each of them. From the total 840 instances, the common instances that constitute the multi class dataset are 395. All the metrics contain a numeric value.

This study has applied two most important preprocessing tasks that are suitable to the nature of the dataset. The dataset contains an imbalanced proportion of instances and a considerably too many attributes. The preprocessing tasks applied are attribute selection and data balancing technique. Hence, document analysis and information gain ratio are used to filter some attributes. In order to balance the dataset, SMOTE (Synthetic Minority Oversampling Technique) has been used.

After the appropriate preprocessing tasks has been made, the dataset has been feed to the ML algorithm to train themselves with.

Choosing a Machine Learning Algorithm

The selection of machine learning algorithms that is suitable for the dataset adopted is a critical issue in classification and detection process. As a result, after examining the specified tasks in relation to the nature of the research problem in this study, the researcher selected four classification algorithms that are believed to be appropriate. The selected machine learning algorithms are from different categories. The researcher selects those four classification algorithms because they are easy to understand, most frequently applied and appropriate to the specific dataset used. The general working of the algorithms used in this study are discussed as follows:

J48 is a decision tree generation technique developed by C4. 5, which is an extension of ID3. It is built on a recursive divide and conquer approach, which is a top-down technique. At the root node, it chooses which attribute to split on, then generates a branch for each conceivable attribute value, splitting the instances into subsets, one for each branch that extends from the base node. J48 algorithm works according to the principle listed below.

Step 1: From each training data attribute, calculate the gain ratio, split information, and entropy.

Step 2: Create a root node with the highest gain ratio from the attribute choices.

Step 3: By deleting the previously selected qualities, calculate gain ratio, split information, and entropy for each attribute.

Step 4: Create an internal node with the highest gain ratio from the attribute choices.

Step 5: Step Pruning trees should be done.

Step 6: Verify that all of the tree's attributes have been created.

- If you replied no, return to stages 4 and 5.
- If you responded yes, proceed to the next stage.

Step 7: Remove any unneeded limbs from the tree.

Step 8: Create rules based on node ordering, starting with the root node, and write them in the form of an IF-THEN rule.

Random forests also known as random decision forests. It is used for classification, regression and other problems by training a large number of decision trees. It's an ensemble method that's superior to a single decision tree because it averages the results to reduce over-fitting. The working of Random Forest algorithm can be described with the following steps.

Step 1: Begin by randomly choosing samples from a dataset.

Step 2: Following that, this algorithm will create a decision tree for each sample. The forecast result from each decision tree will then be obtained.

Step 3: Voting will take place in this phase for each expected outcome.

Step 4: Finally, as the final prediction result, choose the prediction result with the most votes.

JRip is a rule-based classifier that used an IF-THEN rule for classifying instances. JRip implements a learner for propositional rules. It uses Incremental Pruning to Reduce Errors on a Repeated Basis (RIPPER). This algorithm goes through four stages Growing a rule, Pruning, Optimization and Selection.

Step 1: The first stage is the Growth stage. In this stage, J48 algorithm generates the rule automatically by eagerly adding characteristics to it until it fulfills the terminating requirements.

Step 2: The second stage is pruning stage. Until a pruning metric is attained, each rule is incrementally trimmed, allowing the pruning of any eventual sequence of the characteristics.

Step 3: The third stage is pruning stage. Each of the created rules is further optimized in this stage.

- Adding characteristics to the original rule indiscriminately.
- Grow a new rule on its own, going through a growth and pruning process.

Step 4: The fourth stage is the selection stage. Hence, the best rules are preserved in this stage, while other rules that are no longer relevant are deleted from the model.

Naive Bayes are a type of "probabilistic classifier" based on Bayes' theorem and strong independence requirements between features. It is a machine learning classifier that is basic, yet effective and widely used. It is a probabilistic classifier that uses the Maximum A Posteriori decision rule to create classifications in a Bayesian context. It's suitable for both binary and multi-class classifications. It performs well in multi-class predictions as compared to the other algorithms [74]. The Naïve Bayes algorithms works according to the following steps.

Step 1: Calculate the prior probability for each of the class labels given.

Step 2: For each class, calculate the likelihood probability for each attribute.

Step 3: Calculate the posterior probability using the Bayes Formula.

Step 4: Given that the input belongs to the higher probability class, determine which class has the greater probability.

It is calculated by the formula below. Where p(c/x) means the posterior probability of a class (target) given a predictor (attribute), P(c) denotes the prior probability of a class, P(x/c) denotes the probability of a predictor given a class, and P(x) denotes the prior probability of a predictor.

$$p(c/x)=\frac{p(x/c)p(c)}{p(x)}$$

Then, those ML algorithms have been applied on the target dataset and as a result a model have been built. These models contain information with respect to different metrics.

3.1.2.4 Evaluation

This stage gives an answer to the question “How well does the artifact work?”. Accordingly, the performance of the created artifact will be assessed. In order to evaluate the performance of a given model, there are a number of performance measurement mechanisms.

In machine learning approach, the performance of a model can be measured using different metrics. The metrics used in this study are Confusion matrix, Precision, recall, f-measure and accuracy. Therefore, the performance of the four ML techniques selected has been tested against the mentioned performance measurement techniques.

3.1.2.5 Conclusion

This is the final stage where the basic findings of the study will be presented and concluded. Hence, the results of the experimental work and the performance of the developed artifact will be generalized. An answer will also be provided by integrating the core ideas of the research work which is, communicating the problem, the specific solution applied and the effectiveness of the solution towards addressing the relevant issues raised.

3.2 Proposed Architecture

An architecture is a blueprint describing how the overall proposed work operates. Hence, this section of the study presents the detail of the overall experimental process followed in order to produce the final result. The figure below illustrates the architecture of the proposed work.

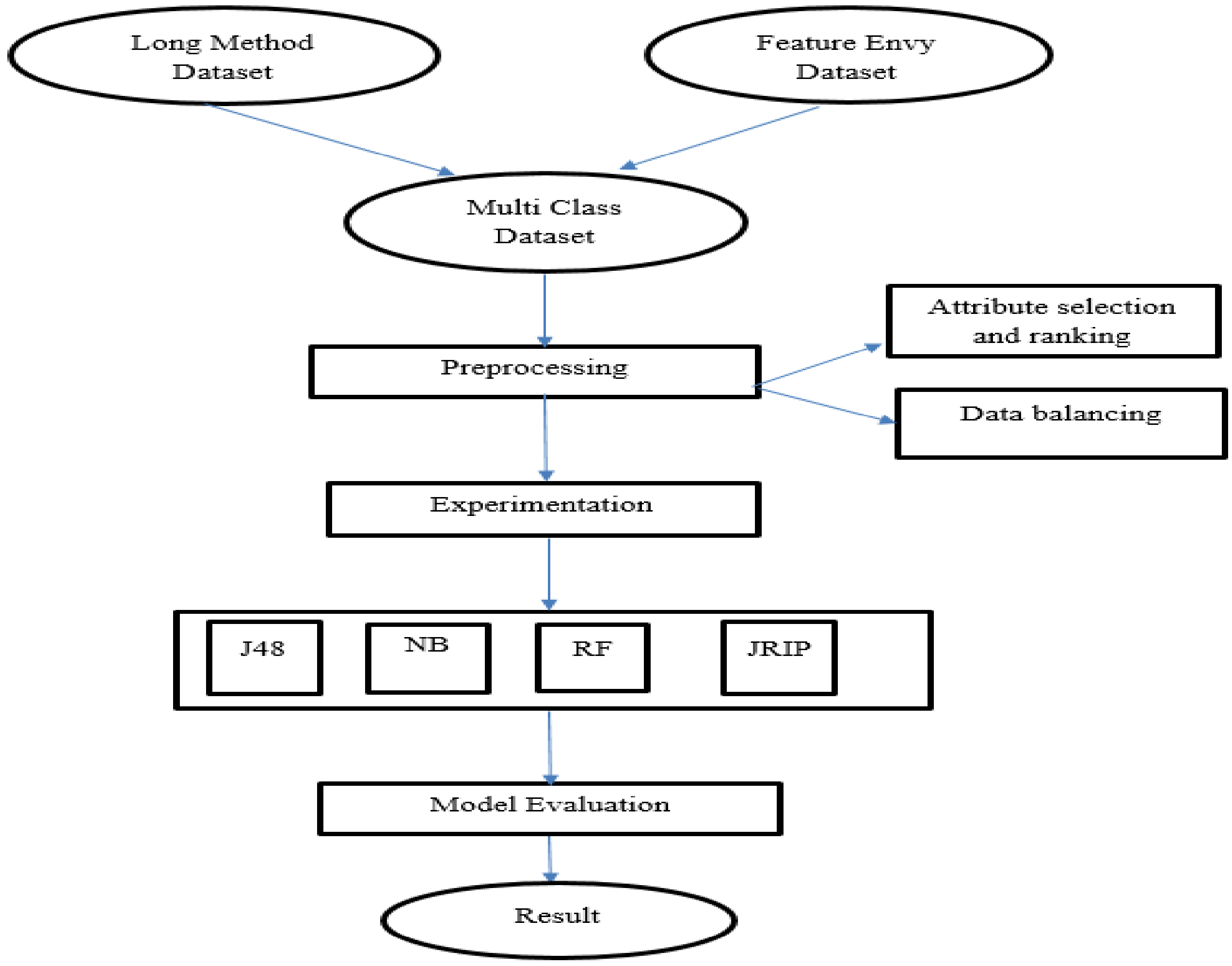


Figure 3. 2 Architecture of the proposed work

As tried to depict in the above figure, two separate datasets “Feature Envy Dataset” and “Long Method Dataset” have been used to construct the Multi Class Dataset (MCD). The common instances from each dataset and their corresponding class label constructed the MCD. Then, after the merged dataset is prepared, the second stage which is preprocessing, has been made.

Accordingly, two most important preprocessing tasks that are proper to the specific dataset adopted have been performed. These preprocessing tasks are attribute selection and data balancing. This dataset contains around 82 attributes, 395 common instances and an imbalanced proportion of class labels. Using attribute selection process, 46 attributes were selected from the total 82 attributes. Additionally, one of the balancing techniques, SMOTE have been applied to balance the proportion of instances in the dataset.

Then having the preprocessed dataset, four machine learning algorithms (J48, Random Forest, JRip and Naïve Bayes) from different categories (Rule based, Tree based, and Probabilistic classifiers) were selected and applied. All of these ML algorithms are applied on 10-fold cross validation where the dataset is portioned in to ten equal partitions (folds) and training is made on each partition iteratively until all the partitions are used as training and testing data. Then, after all machine learning techniques are trained, the result of each algorithm is evaluated using different evaluation metric. Finally, the results of all the ML algorithms will be compared in terms of the specified metrics. Those metrics are accuracy, TP rate, FP rate, precision, recall and f-measure.

CHAPTER FOUR

SYSTEMATIC LITERATURE REVIEW

This chapter presents the methodology followed in the systematic literature review (SLR) performed by the study. It contains every single step followed to conduct the (SLR) and finally presents the basic findings after the synthesis of the information gained.

4.1 Search Strategy

The search strategy in this study consists of search term identification, identification of resources to be searched, search process and identification of criteria for selection of papers. Primarily, a search string that enables to extract all the required materials from digital libraries should be prepared using a set of different terms that could be directly related to the study. Then, the next step is to go for the places to dig in to, which are online sources of materials named digital libraries. Having the digital libraries and the search string, a number of documents will be retrieved. But the main work is to discriminate the relevant paper from the huge set of documents. To do so, a set of exclusion, inclusion and quality assessment criteria is required, then after, a certain number of papers that have direct importance to the study will be selected. Finally, after all related studies have been selected, the papers that pass after the mentioned steps will be utilized for final analysis to answer the set of research questions to be addressed.

4.1.1 Search Term Identification

In this study, a search string from Azeem et al. [56] has been adopted. Since the research question and nature of their study is quite similar with this study and additionally, they have followed a very clear search strategy to prepare the search term, it's preferred to directly adopt their keyword. The researchers in order to select the search string, they have followed a five-step clear guideline which is proposed by the researchers at [85]. Those steps were presented as follows.

1. Used the research questions for the derivation of major terms, by identifying population, intervention, and outcome.
2. For all the major terms, found the alternative spellings and/or synonyms.
3. Verified the keywords in any relevant paper.

4. Used Boolean operators for conjunction in case a certain database allows it, i.e., we used the OR operator for the concatenation of alternative spellings and synonyms whereas the AND operator for the concatenation of major terms.
5. Integrated the search string into a summarized form if required.

The researchers finally summarized the results of each steps mentioned above so that they will be able to design better search terms (string) for their study. The steps' outcomes are as follows: -

- For the first step i.e. identification of population, intervention, and outcome, the researchers have implicated Code smell detectors as population, machine learning techniques as Intervention and Code smells as outcomes.
- The alternative spellings and synonyms for the aforementioned relevant phrases are identified in the second stage: -
  - For the term Code Smells, the synonyms listed are “code smells” OR “code smell” OR “code bad smells” OR “bad code smells” OR “bad smells” OR “anomalies” OR “antipatterns” OR “antipattern” OR “design defect” OR “design-smells” OR “design flaw”.
  - For the term Machine Learning, the alternative words listed are “machine learning” OR “supervised learning” OR “classification” OR “regression” OR “unsupervised learning”.
  - For the term Prediction, the alternative words listed are “prediction” OR “detection” OR “identification” OR “prediction model” OR “model”.
  - For the term Software, the synonyms identified are “software” OR “software engineering”.
- For the third step, they stated that they have checked all possible keywords in the papers they thought are relevant and they did not find any other alternative words or synonyms to add into the set of relevant terms.
- For the fourth step, they have used a Boolean operator to connect the above representative search terms and bring up the overall search term used for navigation of relevant documents. Hence, they were able to come up with the search term “((“code smells” OR “code smell” OR “code bad smells” OR “bad code smells” OR “bad smells” OR anomalies OR anti-patterns OR antipattern OR “design defect” OR “design-smells” OR “design

flaw”) AND (“machine learning” OR “supervised learning” OR classification OR regression OR “unsupervised learning”) AND (software OR “software engineering”))”

- Finally, for the fifth step, they have indicated that some digital libraries like IEEE Xplore digital library have search term limitation. So, they come up with a more concise yet still representative search term which is “((“code smells” OR “code bad smells” OR “bad smells” OR antipatterns OR “design defect” OR “design-smells” OR “design flaw”) AND (“machine learning” OR “supervised learning” OR “unsupervised learning”) AND (detection OR identification OR “prediction model”) AND (software OR “software engineering”))”.

4.1.2 Search Term Formation

As tried to be discussed in the above steps, the final search string which is proposed by [56], is directly adopted for this study as a final search string.

“((“code smell” OR “code bad smell” OR “bad smell” OR anti-patterns OR “design defect” OR “design smells” OR “design flaw”) AND (“machine learning” OR “supervised learning” OR “unsupervised learning”) AND (“detection” OR “identification” OR “prediction model”) AND (“software” OR “software engineering”))”

4.1.3 Resources Searched

According to [56], Selection of proper resources to search for a relevant literature plays a significant role in systematic literature review. There are different repositories that are available online. According to [85], the online digital libraries IEEE Xplore digital library, ACM digital library, Science direct, Springer link, Scopus and Engineering village, are recognized as the most representative for Software Engineering research and are used in many other SLR studies. Additionally, these databases are reported as the popular venues for publishing papers on machine learning and bad smell detection studies [58]. But, from those mentioned online sources, this study uses the following three resources to search for all the available literature relevant that enables to find answers to the research questions. The selection of those databases was driven merely by the choice of search term. Meaning, the other databases have a search term limitation that they can handle a shorter search term with a fewer Boolean operator hence, they are not considered in the study. The final result doesn’t represent the information from those excluded databases.

Table 4. 1 Online digital Libraries accessed

No	Name	URL
1	IEEE	https://ieeexplore.ieee.org/ Xplore/home.jsp
2	ACM	https://dl.acm.org/
3	Springer	https://link.springer.com/)

With the above adopted search string, all the available documents published on those resources from the year 2017 to 2020 have been searched and downloaded in to the initial set of materials to choose from. The materials considered in this study are articles, journals and conference papers. Accordingly, the IEEE Xplore digital library initially retrieves a total of 41 documents with the given keyword and year interval. From those documents, 36 of them were conference papers, 4 were journals and the rest document was in the category of early access articles.

Springer on the other hand initially returns a total of 823 search results given the predefined search term. But, the content type to be considered in the study excludes other type of documents that are not articles, conference papers and journals. So, the study considers a total of 365 documents which consists of 268 articles and 97 conference papers.

ACM in the same way retrieves 163 search results containing articles and conference papers related to every word that are included in the search string from the year 2017 to 2020.

A total of 569 candidate papers were retrieved from those three online databases. This number shows that there is a significant increase in the number of researches made around this area and the topic is getting a huge attention from the research community these days comparing to the studies done on the same ground earlier before 2015.

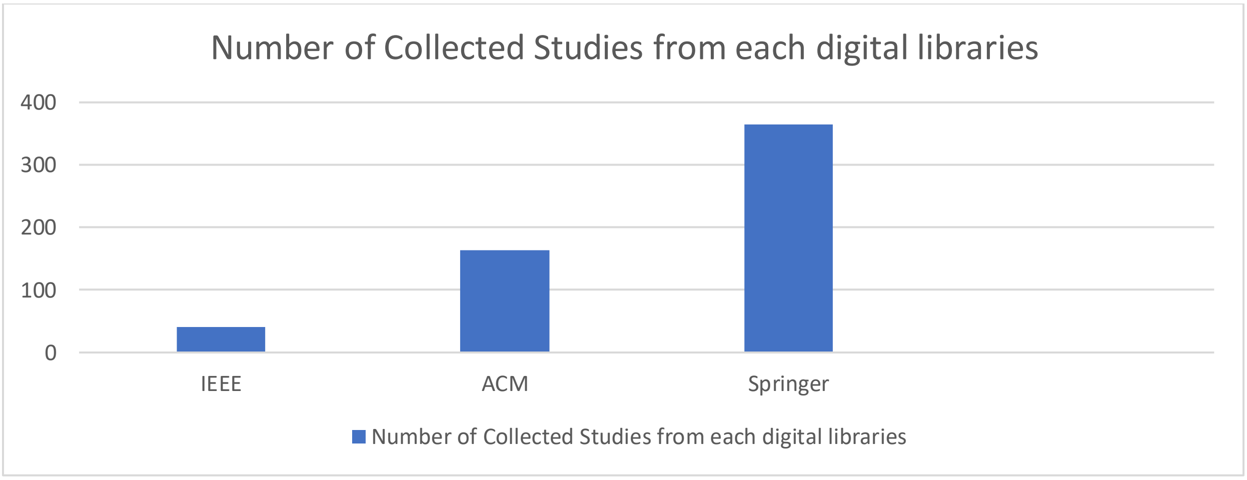


Figure 4. 1 Number of collected studies from 2017 to 2020

It is believed that those search results include all the relevant papers needed to conduct the SLR and hence decrease the likelihood if excluding important papers. But, still there are lots of papers that might not have much importance to enable answering the research questions mentioned. So, a clear filtration of relevant papers is required. This filtration process will be conducted using a set of an inclusion, exclusion criteria and quality assessment techniques. For this reason, only paper that can address the following questions will be selected for the final mapping study.

Research Question 1: Which code smells are most commonly detected using machine learning techniques?

Research Question 2: Which machine learning techniques better applicable to detect code smells?

Research Question 3: What datasets have been used for code smell detection?

4.2 Study Selection

It is worth nothing to include all types of papers (i.e., journal, conference, workshop, and short papers) with the aim of collecting a set of relevant sources as more comprehensive as possible [56]. So, selection of proper resources plays a vital role in answering the questions mentioned earlier. This process of selection should rely on a concrete and clear criterion. Selection of papers driven with a manual interest could cause in an inappropriate generalization. Hence, in order to come up with the best representative papers, the study presents those sets of criteria to minimize the bias of excluding important materials and inclusion of inappropriate resources. So, as depicted in the above table there are initial set of 569 candidate papers. This number shows that there is a significant increase in the number of researches made around this topic these days when comparing to the studies done on the same ground earlier before 2015.

4.2.1 Exclusion Criteria

Exclusion criteria aims at excluding the resources that couldn't have much importance with respect to the objectives of the study and the research questions to be addressed. Beginning from the initial set containing 569 candidate papers, it is required to filter out the papers that meets the following requirements.

- Papers that were written in other languages rather than in English.
- Papers that have either abstract and keyword or any other segmented text and the rest full text is not available.
- Papers that focus on design pattern or any other related concept and not exactly related to the word code or bad smells.
- Papers that put their main focus on other code smell detection tools rather than pure machine learning techniques.

In this study, the selection of papers with respect to the listed points above is made by checking the Title, Abstract as well as the keyword of the documents.

4.2.1.1 IEEE documents exclusion process

With respect to the first point of exclusion criteria, one of the documents downloaded from IEEE digital library fulfills the requirement hence it is excluded due to the fact that it is written in other language. Another paper has also met the second criteria and its full text is unavailable hence, it is also excluded from the list. Having the third criteria, 16 documents were excluded because their main focus was on topics that are not exactly related to code or bad smell detection. Finally, in the fourth stage 9 papers met the criteria and were excluded from the list of documents to pass to the next stage of filtration. After the exclusion criteria has been made to all documents, 14 papers out of 41 made it to the next stage.

4.2.1.2 Springer documents exclusion process

From the considered 365 documents, 8 documents were written in other languages hence excluded in the first round. The second round excludes 35 documents since their full text isn't available. Having the third criteria, 120 documents were excluded because their main focus was on topics that are not directly related to code, bad smell detection or design defect. Forth criteria also filtered 198 papers because even if the focus of those papers is on code smell, they have either applied other techniques rather than machine learning or their orientation was not merely on the detection of code smells. Finally, only four papers pass all the criteria.

4.2.1.3 ACM documents exclusion process

The first point of criteria exclusion leads to the exclusion of 2 papers since they were written in another language. With the second point, 5 papers were excluded because they didn't contain full text. There were also 121 papers in which their main focus was on ideas far from the detection of code smell and fulfill the third criteria hence excluded from the study. Additional 24 studies were excluded mainly because their aim was on other detection mechanisms rather than pure machine learning techniques. Accordingly, only 11 studies were able to make it to the second inclusion criteria.

The first criteria result in the exclusion of 27 documents from IEEE, 361 documents from Springer and 152 documents from ACM. Totally, 540 documents were excluded from the initial set of 569 documents. Hence 29 documents passed to the next stage.

4.2.2 Inclusion Criteria

Documents that met the constraints reported below were included in the study.

- All the articles directly reporting machine learning techniques for code smells detection.
- All the articles directly reporting the detection of different types of code smell.
- Articles that provide new strategies for improving the performance of existing code smell detection machine learning techniques.

From the 29 documents that have passed to this stage, the papers that are found to satisfy the inclusion criteria are 22 documents. While IEEE contributes 11 papers from the total documents, the rest 11 of them were from Springer and ACM contributing 2 and 9 documents respectively.

4.2.3 Quality Assessment

Once the selection is made with the above criteria, measuring of the quality of the publications is highly required. Because all papers should be acknowledged for their worth of ability of addressing the research questions properly. So, in order to assess the credibility of the selected papers, the below mentioned checklists have been used.

- Are the code smells that the proposed approach detects well-defined? (Q1)
- Is the machine learning classifier used clearly defined? (Q2)

- ➢ Is the dataset being used by the researcher mentioned clearly? (Q3)

Each of the above questions was marked as “Yes”, “Partially” or “No”. Studies that are marked as partial are considered because some details could have been driven from the full text, even if they were not explicitly reported. These answers are scored as follows:

- “Yes” =1,
- “Partially” =0.5, and
- “No” =0.

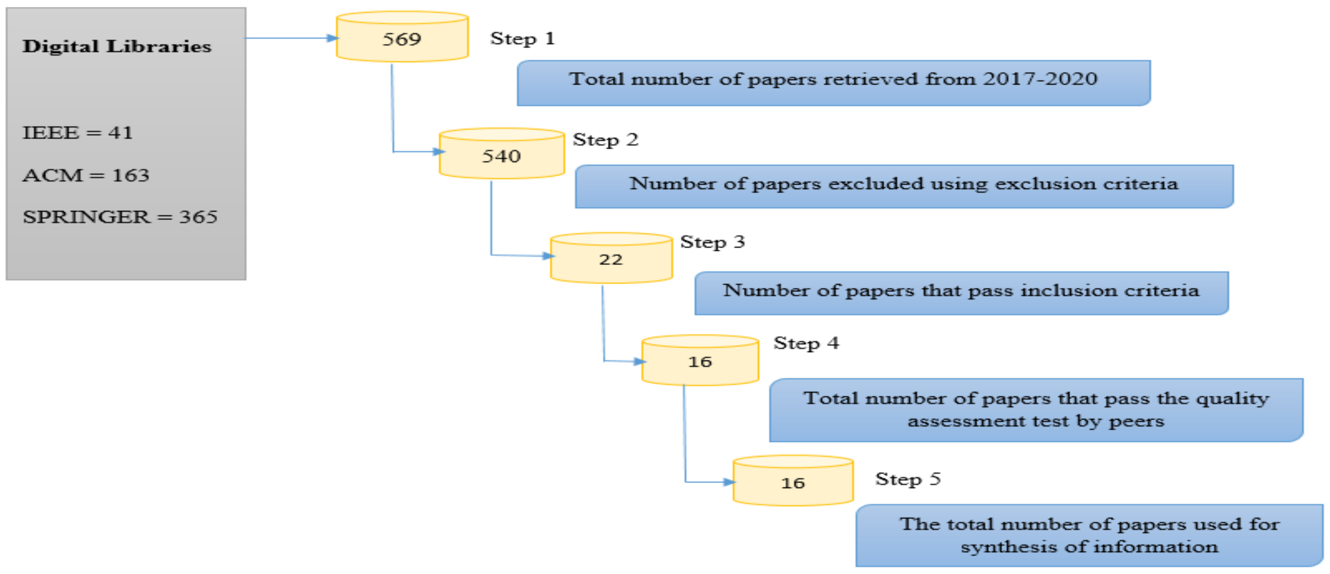
For each selected primary study, its quality score was computed by summing up the scores of the answers to all the three questions. The study classified the quality level into High (score = 3), Medium ($2 \leq$ score < 3), and Low (score < 2). For this task, a group containing three academicians whose background is in computer related fields and have accomplished their postgraduate program in related fields have been selected using purposive sampling. They have jointly discussed and evaluated each of the studies with respect to the listed quality checklists.

Then this table represents the list of score and status given to all the 22 Papers that pass the inclusion criteria according to the purposively selected academicians.

Table 4. 2 List of papers that pass the inclusion criteria

No	Study	Quality questions			Total Score	Status
		Q1	Q2	Q3		
1	Mhawish, M.Y. et al., 2020 [89]	1	1	1	3 (High)	Pass
2	Guggulothu, T. et al., 2020 [66]	1	1	1	3 (High)	Pass
3	Cruz, D. et al., 2020 [90]	1	1	1	3 (High)	Pass
4	Pecorelli, F. et al., 2019 [91]	1	1	1	3 (High)	Pass
5	Luiz, F.C., 2019 [49]	1	1	1	3 (High)	Pass
6	Rubin, J. et al., 2019 [92]	0.5	1	0	1.5(Low)	Failed
7	Oliveira, D. et al., 2020 [93]	1	0	0	1(Low)	Failed
8	Liu, H. et al., 2019 [94]	0.5	1	0	1.5 (Low)	Failed
9	Azadi, U. et al., 2018 [95]	1	1	1	3 (High)	Pass
10	Guo, X. et al., 2019 [96]	1	1	0.5	2.5(Medium)	Pass
11	Pecorelli, F. et al., 2019 [97]	1	0	0	1(Low)	Failed
12	Kaur, A. et al., 2017 [98]	1	1	1	3 (High)	Pass
13	Gupta, H. et al., 2019 [99]	1	1	1	3 (High)	Pass
14	Karađuzović-Hadžiabdić, K. et al., 2018 [100]	1	1	1	3 (High)	Pass
15	Das, A.K. et al., 2019 [101]	1	1	1	3 (High)	Pass
16	Kiyak, E.O. et al., 2019 [102]	1	1	1	3 (High)	Pass
17	Singh, R. et al., 2020 [103]	0	1	1	2 (Medium)	Pass
18	Di Nucci, D. et al., 2018 [86]	1	1	1	3 (High)	Pass
19	Jesoudoss, A. et al., 2019 [104]	1	1	0	2 (Medium)	Pass
20	Yang, Y. et al., 2018 [105]	0	1	1	2 (Medium)	Pass
21	Thongkum, P. et al., 2020 [106]	0	1	0.5	1.5 (Low)	Failed
22	Chen, D. et al., 2019 [107]	0	0.5	1	1.5 (Low)	Failed

Finally, after the quality assessment has been done, 16 studies that scored in high and medium levels have been selected as a final result of the selection process. The table below presents the overall information starting from the initial set of documents retrieved to the number of papers that pass the final stage for synthesis. Here is the overall SLR execution process.



Digital Libraries
IEEE = 41
ACM = 163
SPRINGER = 365
569
Step 1
Total number of papers retrieved from 2017-2020
540
Step 2
Number of papers excluded using exclusion criteria
22
Step 3
Number of papers that pass inclusion criteria
16
Step 4
Total number of papers that pass the quality assessment test by peers
16
Step 5
The total number of papers used for synthesis of information

Figure 4. 2 Systematic Literature Review (SLR) execution process

Table 4. 3 Overall number of filtered documents throughout the selection process

No	Digital Library	Total number of papers retrieved	Excluded Papers	Papers that pass inclusion criteria	Papers that failed inclusion criteria	Papers that pass Quality assessment
1	**Springer**	365	361	2	2	2
2	**ACM**	163	152	9	2	5
3	**IEEE**	41	27	11	3	9
Total		**569**	540	22	7	16
Total of excluded and included papers			**569** papers			

Here is the graphical representation of the overall selection stage and result.

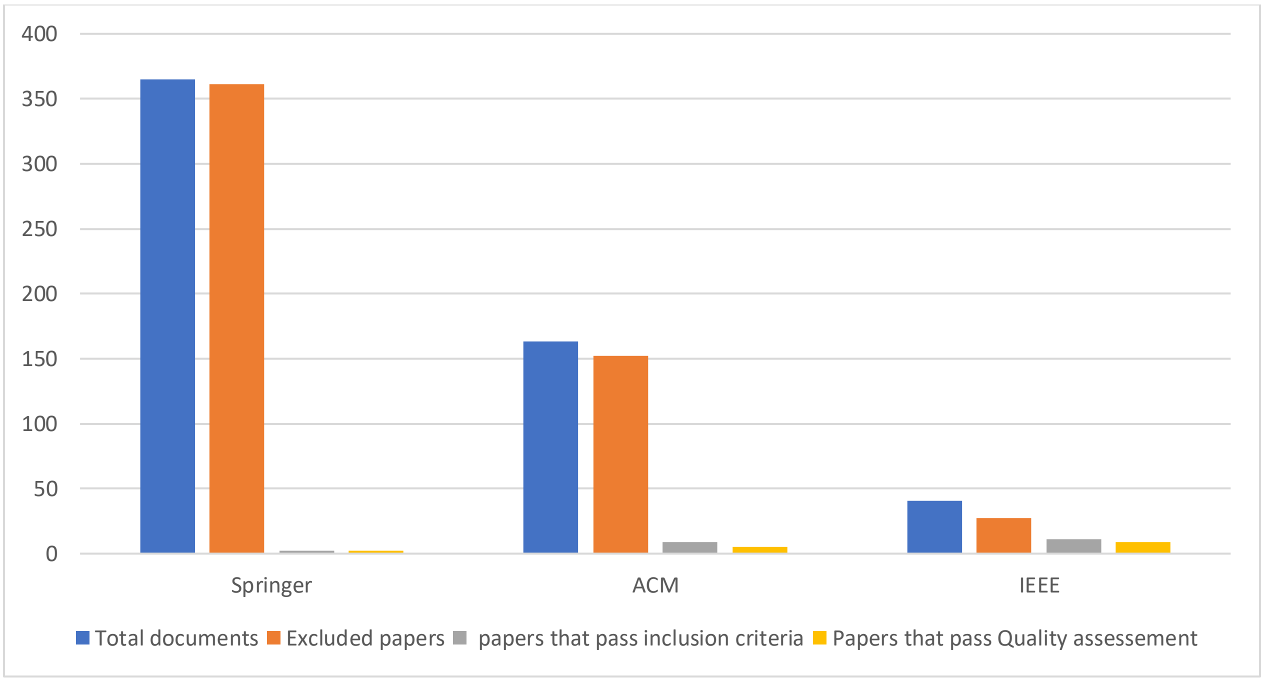


Figure 4. 3 Number of filtered documents throughout the selection process

4.3 Data Extraction

For the purpose of easy presentation, the selected studies are represented below and will be alternatively used accordingly throughout the study. So, following this, the study tried to answer the first three research questions that are expected to be addressed at the end of the study. The final 16 papers (2 from Springer, 5 from ACM and the rest 9 from IEEE) are represented as follows.

Table 4. 4 Quality approved papers and their representation

No	Study	Representation	Digital Library
1	Kaur, A. et al., 2017 [98]	S1	IEEE
2	Gupta, H. et al., 2019 [99]	S2	IEEE
3	Karađuzović-Hadžiabdić, K. et al., 2018 [100]	S3	IEEE
4	Das, A.K. et al., 2019 [101]	S4	IEEE
5	Kiyak, E.O. et al., 2019 [102]	S5	IEEE
6	Singh, R. et al., 2020 [103]	S6	IEEE
18	Di Nucci, D. et al., 2018 [86]	S7	IEEE
8	Jesoudoss, A. et al., 2019 [104]	S8	IEEE
9	Yang, Y. et al., 2018 [105]	S9	IEEE
10	Mhawish MY, et al., 2020 [89]	S10	Springer
11	Guggulothu T et al., 2020 [66]	S11	Springer
12	Luiz, F.C. et al., 2019 [49]	S12	ACM
13	Pecorelli, F. etal.,2019 [91]	S13	ACM
14	Azadi, U. et al., 2018 [95]	S14	ACM
15	Cruz, D. et al., 2020 [90]	S15	ACM
16	Guo, X. et al., 2019 [96]	S16	ACM

As described earlier the studies collected are bounded to the year interval 2017 to 2020. But the total number of studies gained after the filtration using necessary steps clearly indicates that this topic is gaining more attention these days compared to the reviewed papers. There were different SLR researches made in earlier years. They were able to get this much papers with a huge time interval but in recent times a comparable or even greater number of papers were retrieved with in a smaller time interval showing a significant increase of interest. The number of filtered papers by year is described as follows.

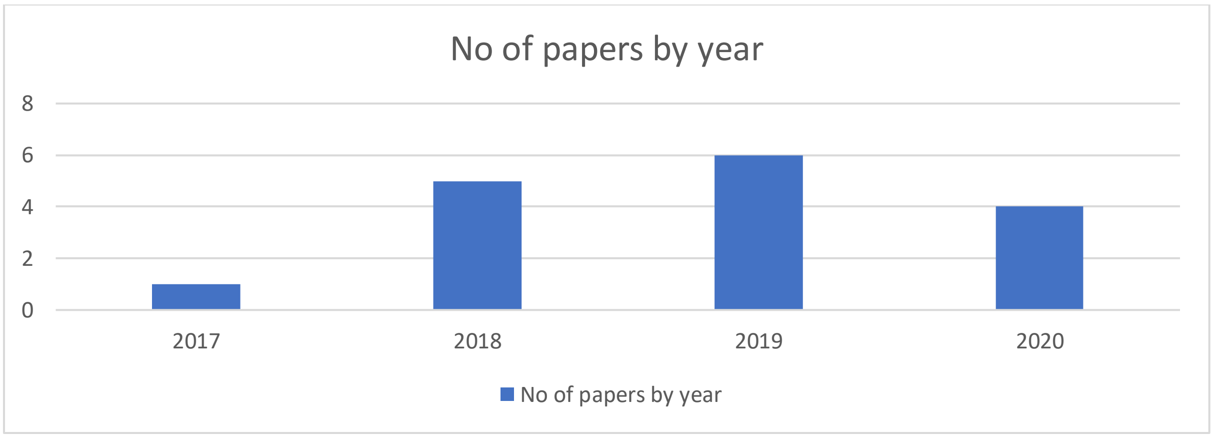


Figure 4. 4 Number of studies in each year after selection process

In order to answer the research questions, it is highly required to understand the basic and detailed information of all the selected studies. Hence, all the required information of the papers is summarized below.

Table 4. 5 Summary of all selected studies with their detailed information

No	Study	Machine Learning Technique used	Code smell detected	Dataset	Evaluation Metrics used
1	S1	SVM	Data class, Feature envy, God class and Long method	XercesV2.7.0, Argo UMLV0.1.9.8 (Open source software)	Precision and Recall
2	S2	Linear regression, Logistic regression, Polynomial regression, Decision tree, ELM-linear, ELM-RBF, ELM polynomial	Blob, Complex class, Swiss Army Knife, Long method, Internal Getter/Setter, No low memory Resolver, Member ignoring method, Leaking inner class	629 open source projects (Available on GitHub)	AUC

3	S3	J48, JRip, Naïve Bayes, Random Forest, SMO, LibSVM, KNN, Decision Tree, Multi-Layer Perceptron, Voted Perceptron	Data class, God class, Feature envy, Long method	76 software systems of Qualitas corpus composed of Java systems	Accuracy
4	S4	CNN	Brain class, Brain method	10 Java projects (Larger in size)	Accuracy
5	S5	C4.5, Random Forest, Naïve Bayes, SVM, Neural network, Ensemble ML, Bagging ML	Long Method, Feature envy, God class, Data class	74 open source java projects in Qualitas corpus	Accuracy, AUCROC, Hamming score, F-measure, Exact match ratio,
6	S6	CNN, SVM, RF, NN, DT, NBG, LR, Ensemble ML, SMOTE	Not specified	Apache dataset (Camel, JEdit, Lucence, Poi, Synapse, Xalan, XeresD)	F-measure
7	S7	J48, JRip, RF, NB, SMO, LibSVM	Data class, God class, Feature envy, Long method	74 software systems belonging to Qualitas Corpus	F-measure
8	S8	RF, SVM	Bloated code detector, Lazy class detector, Primitive obsession detector, Duplicated code detector, Feature envy detector, Too many Literal detector	Not specified	Not specified

9	S9	Linear Regression, Decision Tree, Random Forest	Not specified	Poi, Velocity, Xalan, Xeres with different versions	F-measure, Recall, Precision and AUC
10	S10	Deep Learning, Decision Tree, GBT, SVM, RF, MLP	Data class, Large class, Feature envy, Long method	Open source	Precision, Recall, F-score, Area Under the ROC curve (AUC), Accuracy.
11	S11	B-Random Forest, Random Forest, B-J48 Unpruned, B-J48 Pruned, J48 Unpruned	Long method and Feature envy	Qualitas corpus 74 Open source projects	Accuracy, Hamming Score, Exact Match Ratio
12	S12	LightGBM, Ensemble, XGBoost, Cat Boost, Naïve Bayes, JRip, Soft voting, Association rule, Under/Over sampling	Divergent change, Feature envy, Large class, Long method, Parallel inheritance, Shotgun surgery	Landfill, publicly available (Open source)	Recall Precision and F-measure
13	S13	Class balancer, SMOTE, Resample, Cost sensitive classifier, NB	God class, Spaghetti code, Class data should be private, Complex class, Long method	125 release of 13 software systems	Precision, Recall, F-measure and Matthews Correlation Coefficient (MCC).
14	S14	Rule Based, Decision Tree and Bayesian algorithm	God class, Long method, Data class, Feature envy, Long	Software from Qualitas Corpus Repository	Not specified

			parameter list and Switch statement		
15	S15	Naïve Bayes, Linear regression, Multi-Layer Perceptron, Decision tree, k-Nearest Neighbor, Gradiant Boosting Machine, Random Forest	God class, Refused Bequest, Long method and Feature envy	Software from Qualitas Corpus Repository	F-measure, Accuracy, Recall and Precision
16	S16	Deep semantic based approach	Feature envy	74 software systems	F-measure, Recall and Precision

4.4 Data Synthesis

The main aim of undergoing a Systematic Literature Review is to find all resources that enables to answer the specific research questions identified by the study. In this study, the SLR seeks to provide a complete analysis into (i) the sorts of code smells considered by previous researchers, (ii) the types of machine learning algorithms utilized by researchers, and (iii) the dataset used by the studies. Hence, with the above detailed information on the studies, this study has tried to analyze and answer the research questions one by one. The questions are:-

- ✓ The code smells that have been detected in the studies
- ✓ The machine learning techniques deployed in the studies
- ✓ The type of dataset being used by the researchers

Question 1 The code smells that have been detected in the studies

The search and filter approach deployed delivers a total of 16 studies and within those studies 26 types of code smells were detected. The detected smells are Data class, Feature envy, God class Long method, Blob, Complex class, Swiss Army Knife, Internal Getter/Setter, No low memory Resolver, Member ignoring method, Leaking inner class, Brain class, Brain method, Bloated code detector, Lazy class, Primitive obsession, Duplicated code, Too many Literal, Large class, Divergent change, Parallel inheritance, Shotgun surgery, Spaghetti code, Class data should be private, Long parameter list and Switch statement. Here is the summarized information of which studies consider which code smell type.

Table 4. 6 Code smell types and the studies considering them

N<u>o</u>	Code smell	Studies	Frequency
1	Data class	S1, S3, S5, S7, S10, S14	6
2	Feature envy	S1, S3, S5, S7, S8, S10, S11, S12, S14, S15, S16	11
3	God class	S1, S3, S5, S7, S10, S13, S14, S15	8
4	Long method	S1, S2, S3, S5, S7, S10, S11, S12, S13, S14, S15	11
5	Blob	S2	1
6	Complex class	S2, S13	2
7	Swiss Army Knife	S2	1
8	Internal Getter/Setter	S2	1
9	No low memory Resolver	S2	1
10	Member ignoring method	S2	1
11	Leaking inner class	S2	1
12	Brain class	S4	1
13	Brain method	S4	1
14	Bloated code detector	S8	1
15	Lazy class	S8	1
16	Primitive obsession	S8	1
17	Duplicated code	S8	1
18	Too many Literal	S8	1
19	Large class	S10, S12	2
20	Divergent change	S12	1
21	Parallel inheritance	S12	1
22	Shotgun surgery	S12	1
23	Spaghetti code	S13	1

24	Class data should be private	S13	1
25	Long parameter list	S14	1
26	Switch statement	S14	1

So, finally it will be easy to generalize the report that tells which code smell have been given more attention in recent studies. Accordingly, the following generalization has been made: -

- The code smells Long method and Feature envy have been considered by 11 studies.
- The code smells God class and Data class have been considered by 8 and 6 studies respectively.
- The code smells Complex class and Large class by has been considered by 2 studies.
- The rest code smells have been considered by a single study.

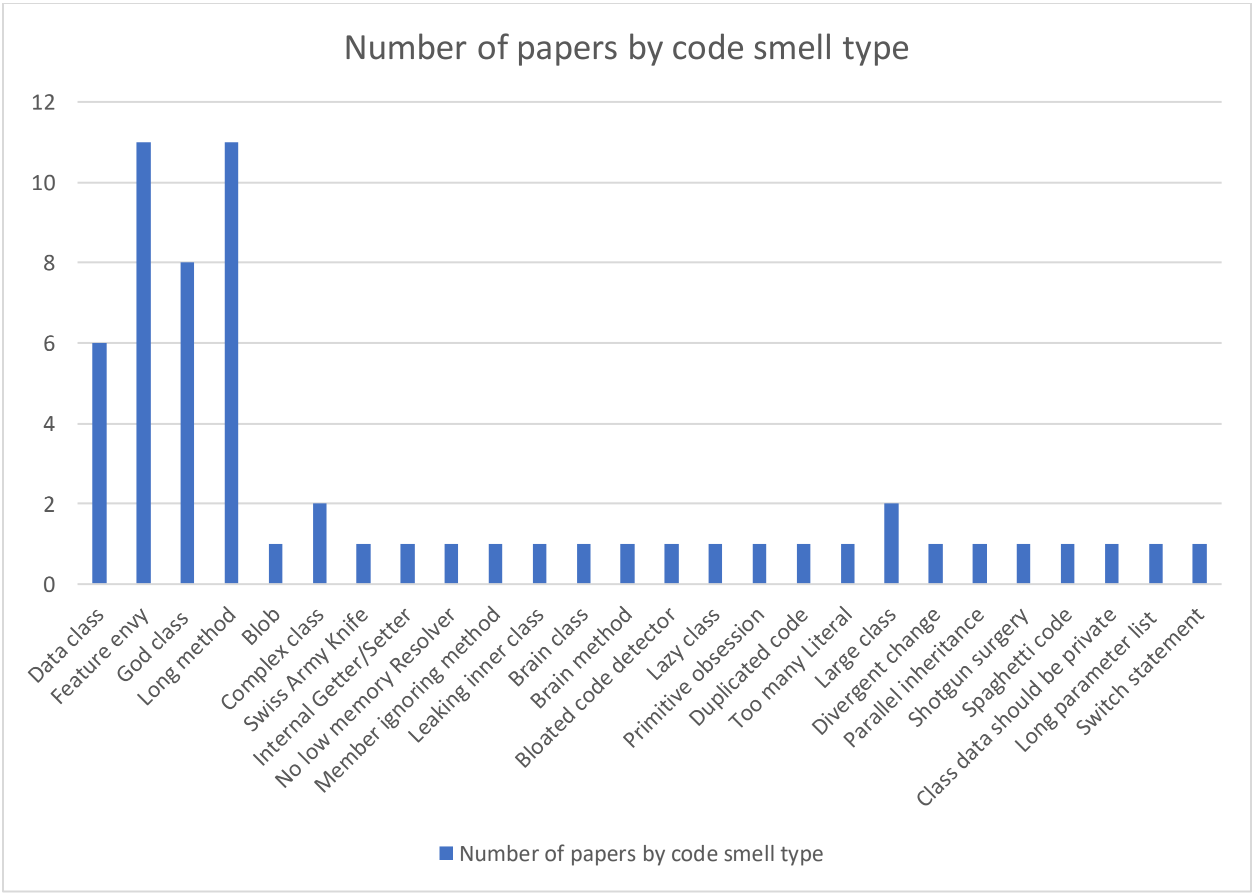


Figure 4. 5 Number of papers by each code smell

Question 2 The machine learning techniques deployed in the studies

A total of 38 machine learning techniques have been deployed in the 16 studies. The techniques are J48, Random Forest, JRip, Naïve Bayes, SVM, Linear regression, Logistic regression, Polynomial regression, Decision tree, ELM-linear, ELM-RBF, ELM polynomial, SMO, LibSVM, Multi-Layer Perceptron, Voted Perceptron, CNN, C4.5, Neural network, Bagging ML, NBG, Deep Learning, , GBT, LightGBM, , XGBoost, CatBoost, Soft voting, Association rule, Under/Over sampling, Class balancer, SMOTE, Resample, Cost sensitive classifier, One class classifier, Rule Based, Bayesian algorithm, KNN, Ensemble ML and Deep semantic based approach. The summarized information of which studies consider the use of which Machine learning techniques is presented below.

Table 4. 7 Machine learning techniques and the studies considering them

No	Machine learning techniques	Studies	Frequency
1	J48	S3, S7, S11	3
2	Random Forest	S3, S5, S6, S7, S8, S9, S10, S11, S15	9
3	JRip	S3, S7, S12	3
4	Naïve Bayes	S3, S5, S7, S12, S13, S15	6
5	SVM	S1, S5, S6, S8, S10	5
6	Linear regression	S2, S9, S15	3
7	Logistic regression	S2	1
8	Polynomial regression	S2	1
9	Decision tree	S2, S3, S6, S9, S10, S14, S15	7
10	ELM-linear	S2	1
11	ELM-RBF	S2	1
12	ELM polynomial	S2	1
13	SMO	S3, S7	2
14	LibSVM	S3, S7	2
15	Multi-Layer Perceptron	S3, S10, S15	3
16	Voted Perceptron	S3	1
17	CNN	S4, S6	2

18	C4.5	S5	1
19	Neural network	S5, S6	2
20	Bagging ML	S5	1
21	NBG	S6	1
22	SMOTE	S6, S13	2
23	Deep Learning	S10	1
24	GBT	S10, S15	2
25	LightGBM	S12	1
26	XGBoost	S12	1
27	CatBoost	S12	1
28	Soft voting	S12	1
29	Association rule	S12	1
30	Under/Over sampling	S12	1
31	Class balancer	S13	1
32	Resampling	S13	1
33	Cost sensitive classifier	S13	1
34	Rule Based	S14	1
35	Bayesian algorithm	S14	1
36	KNN	S3, S15	2
37	Ensemble ML	S5, S6, S12	3
38	Deep semantic based approach	S16	1

So, according to the information above, the study concludes the following findings about machine learning techniques used in the studies from 2017-2020: -

- The machine learning technique Random Forest has been considered by 9 studies and hence is the leading machine learning technique used in recent studies.
- The machine learning techniques Decision tree, Naïve Bayes and SVM have been considered by 7, 6 and 5 studies respectively.
- The machine learning techniques J48, JRip, Linear Regression, MLP and Ensemble ML have been considered by 3 studies.

- The machine learning techniques SMO, LibSVM, CNN, Neural Network, SMOTE, GBT and KNN each have been considered by 2 studies.
- The rest machine learning techniques have been considered by a single study.

Here is the graphical summary of number of papers by machine learning techniques

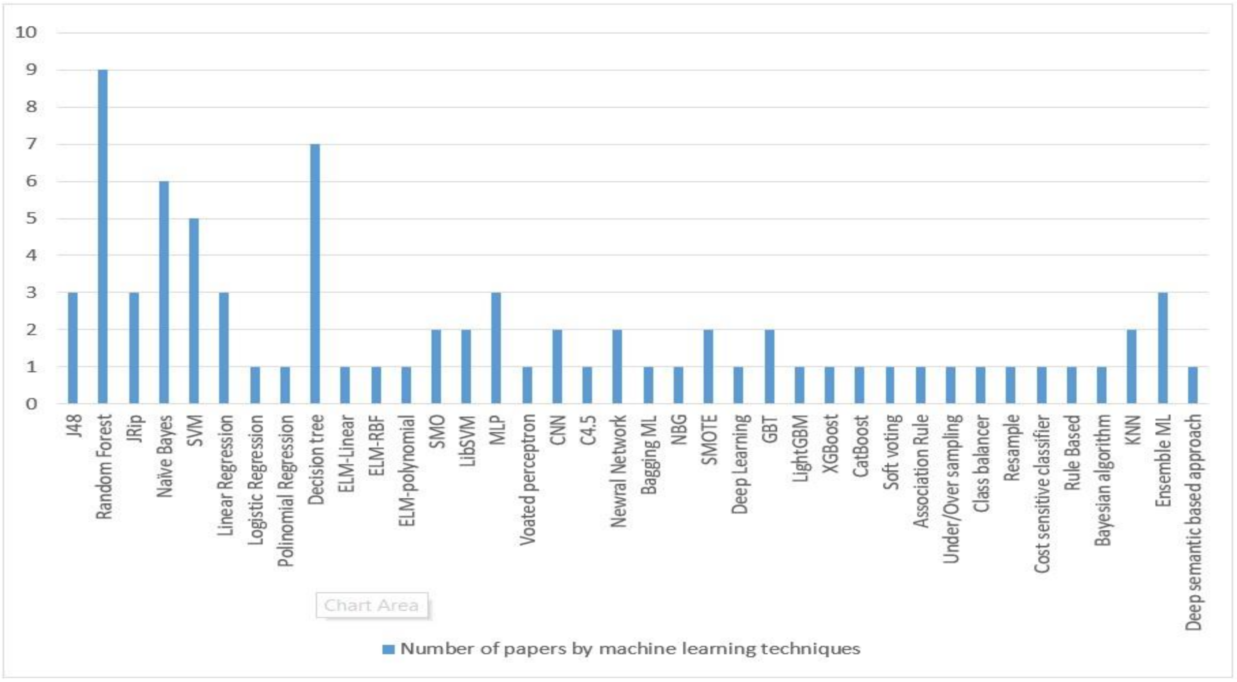


Figure 4. 6 Number of papers by each machine learning technique

With regard to the evaluation metrics deployed, Precision, Recall, F-measure, Accuracy, AUCROC, Exact match ratio, Hamming Loss, Matthews Correlation Coefficient are the evaluation metrics used in the selected papers. Here is the summarization of information about those metrics in the studies.

Table 4. 8 Evaluation metrics and studies considering them

No	Evaluation metric	Studies	Frequency
1	Precision	S1, S9, S10, S12, S13, S15, S16	7
2	Recall	S1, S9, S10, S12, S13, S15, S16	7
3	F-measure	S5, S6, S7, S9, S10, S12, S13, S15, S16	9
4	Accuracy	S3, S4, S5, S10, S11, S15	6
5	AUC ROC	S2, S5 S9, S10, S11	5
6	Exact match ratio	S5, S11	2
7	Hamming Loss	S5, S11	2
8	Matthews Correlation Coefficient	S13	1

According to the table summarized, F-measure is the leading evaluation metric identified which is used by 9 studies. Precision and Recall on the other hand are used by 7 studies and Accuracy metric is considered by 6 studies.

Then by merging the concepts of the first two previous research questions (Q1 and Q2), this study explores the scenario that which machine learning techniques is best in the detection of a specific code smell with respect to the most widely used evaluation metrics.

Accordingly, this study has considered the four leading code smells containing two method level smells (Long Method and Feature Envy) and two class label smells (Data class and God class). Then the best machine learning technique has been tested against the previously identified (most commonly used) evaluation metrics which are precision, recall, accuracy and F-measure.

The first metric precision is calculated by the total number of correctly labeled instances (true positives) divided by the sum of total correctly labeled instances (true positives) and total incorrectly labeled instances (false positives). The second metric which is recall, is calculated by the total number of correctly labeled instances (true positives) divided by the sum of total correctly labeled instances (true positives) and total incorrectly labeled instances (false negatives). The other metric F-measure is calculated by doubling the product of Precision and Recall divided by the total of Precision and Recall. Finally, Accuracy of a given learning algorithm is calculated by the sum of total correctly labeled instances (true positives) and (True negatives) divided by the total number of instances.

1. Long Method

The code smell Long Method has been detected by 11 papers (S1, S2, S3, S5, S7, S10, S11, S12, S13, S14 and S15). Here is the summery of the performance of the best machine learning technique used in the studies considering Long Method code smell. But this table excludes the results of S2 and S14. The evaluation metric used in S2 is AUC and hence is out of scope. The other study S14 hasn't clearly specify the performance metrics so it is not included in the table.

Table 4. 9 Best performed machine learning techniques and their performance in each studies considering Long method smell.

No	Studies	Evaluation metric and result in %		ML technique used
1	S1	Precision	75.5	SVM
		Recall	75.45	
2	S3	**Accuracy**	**99.76**	**Random Forest**
3	S5	Accuracy	93.6	Random Forest
		F-measure	92.8	
4	S7	Accuracy	84	LibSVM
		F-measure	45	Naïve Bayes
5	S10	**Precision**	**90.16**	**Random Forest**
		Recall	**100**	
		F-measure	**94.8**	
		Accuracy	96.91	
6	S11	Accuracy	97.5	B-J48 pruned
7	S12	Precision	72	Catboost
		Recall	65	Random Forest
		F-measure	67.94	
8	S13	Precision	15	No balancing technique with
		Recall	80	One class classifier
		F-measure	23	No balancing technique
9	S15	Accuracy	99.1	GBM and KNN
		Precision	58.5	Random Forest
		Recall	44.9	Logistic Regression
		F-measure	23.3	Random Forest

Accordingly, higher results for the code smell Long method in terms of all the metrics Accuracy Precision, Recall and F-measure is gained with the classifier Random Forest with the results **99.76, 90.16, 100 and 94.8 respectively**. Additionally, higher Accuracy is registered by S3. Precision

Recall and F-measure exhibits higher result in S10. So, it clearly shows that Random Forest is the best machine learning technique in the detection of Long method with a very promising result.

2. Feature Envy

The code smell Feature Envy has been detected by 11 papers (S1, S3, S5, S7, S8, S10, S11, S12, S14, S15 and S16). The studies S8 and S14 don’t clearly specify the type of evaluation metric used hence, the information from them isn’t considered.

Table 4. 10 Best performed machine learning techniques and their performance in each studies considering Feature Envy smell.

No	Studies	Evaluation metric and result in %		ML technique used
1	S1	Precision	67.9	SVM
		Recall	62.25	
2	S3	**Accuracy**	**99.67**	**MLP**
3	S5	Accuracy	93.6	Random Forest
		F-measure	92.8	
4	S7	Accuracy	84	LibSVM
		F-measure	50	J48 pruned
5	S10	Accuracy	96.91	Random Forest
		Precision	96.43	
		Recall	98.18	
6	S11	Accuracy	97.5	B-J48 pruned
7	S12	Precision	79	Random Forest
		Recall	42	Soft voting
		F-measure	47.15	Random Forest
8	S15	Accuracy	96.5	KNN and GBM
		Precision	67.7	Random Forest
		Recall	38.6	Logistic Regression
		F-measure	28.1	Random Forest
9	S16	**Precision**	**97.07**	**Deep semantic based approach**
		Recall	**99.13**	**Deep semantic based approach**
		F-measure	**98.09**	**Deep semantic based approach**

According to the information in the table, higher results for the code smell Feature Envy in terms of Accuracy is gained by MLP (99.67) in S3. In terms of Precision, and Recall and F-measure, the classifier Deep semantic based approach has showed higher result 97.07 and 99.13 and 98.09

respectively in S16 compared to the other techniques. Showing that MLP and Deep semantic based approach could be the best machine learning techniques in the detection of the code smell Feature Envy.

3. Data Class

The code smell Data Class has been detected by 6 papers (S1, S3, S5, S7, S10 and S14). The study S14 hasn't clearly specify the performance metrics so it is not included in the table. The rest papers with their performed and ML technique used are listed below.

Table 4. 11 Best performed machine learning techniques and their performance in each studies considering Data Class smell.

No	Studies	Evaluation metric and result %		ML technique used
1	S1	Precision	85.55	SVM
		Recall	77	
2	S3	**Accuracy**	**98.57**	**Random Forest**
3	S5	Accuracy	96.9	Random Forest
		F-measure	**97.5**	**Random Forest**
4	S7	Accuracy	84	LibSVM
		F-measure	57	J48 pruned
5	S10	Accuracy	97.11	Random Forest
		Precision	**97.2**	**Random Forest**
		Recall	**100**	**GBT**

Hence, according to the information above, best accuracy, F-measure and precision result is gained by the Random Forest classifier in the studies S3, S5 and S10. On the other hand, the best Recall result is gained GBT in S10. So, the ML techniques Random Forest and GBT are the best techniques for the detection of the smell Data class.

4. God Class

The code smell God Class has been detected by 7 papers (S1, S3, S5, S7, S10, S13, S14 and S15). The other study S14 hasn't clearly specify the performance metrics so it is not included in the table. In addition, the study has tried to calculate the F-measure result given the precision and recall. But only in a case that both the recall and precision are gained by the same ML technique in the same study and if it’s believed that the F-measure result could be comparable to the F-measure of other studies. The F-measure of S1 is calculated by the following formula.

Table 4. 12 Best performed machine learning techniques and their performance in each studies considering God Class smell.

No	Studies	Evaluation metric and result %		ML technique used
1	S1	Precision	95.35	SVM
		Recall	90.15	
		F-measure	92.7	
2	S3	**Accuracy**	**97.86**	**JRip**
3	S5	Accuracy	96.9	Random Forest
		F-measure	97.5	
4	S7	Accuracy	84	LibSVM
		F-measure	58	B-J48 reduced error pruning
5	S10	Accuracy	97.11	Random Forest
		Precision	96.24	SVM
		Recall	96.24	Random Forest
6	S13	Precision	26	SMOTE and NB
		Recall	93	
		F-measure	41	
7	S16	**Precision**	**97.07**	**Deep semantic based approach**
		Recall	**99.13**	
		F-measure	**98.09**	

Hence, best accuracy results 97.86 is gained by JRip in S3. The best results F-measure and Recall and precision were gained by 98.09 and 99.13 and 99.07 were gained by Deep semantic based approach in S16. Indicating that JRip and Deep semantic based approach are the best machine learning techniques in the detection of God class.

Question 3 The dataset used in the studies

A different kind of datasets have been used in these studies. But this study has summarized the dataset being used as i. Kind of dataset used ii. Nature of dataset as an open source (publicly available) or other projects datasets (Industrial) and iii. Nature of dataset as cross-project or within project dataset. Some of the datasets used are Xceres, Argo UML, open-source projects Available on GitHub, software systems of Qualitas corpus, Apache dataset containing Poi, Xceres, Xalan, and Velocity with their different versions.

Table 4. 13 Type of dataset used

No	The dataset used	Studies	Frequency
1	Xerces	S1, S6, S9	3
2	Argo UML	S1	1
3	Apache dataset containing Camel, JEdit, Lucence, Poi, Synapse, Xalan and XeresD	S6	1
4	Qualitas corpus containing java projects	S3, S5, S11, S14, S15	5
5	Velocity	S9	1
6	Landfill	S12	1
7	Poi	S6, S9	2
8	Xalan	S9	1
9	Synapse	S6	1
10	Camel	S6	1
11	JEdit	S6	1
12	Lucence	S6	1
13	Others (names not mentioned but open source)	S10, S11, S13, S16	4

Table 4. 14 The nature of dataset being used (open source (publicly available) or other projects datasets (Industrial)

No	Type of dataset used	Studies	Frequency
1	Open source (publicly available)	S1, S2, S3, S5, S6, S7, S9, S10, S11, S12, S13, S14, S15	13
2	Other projects datasets (Industrial)	S4, S16	2

Table 4. 15 Nature of dataset as cross project or within project dataset.

No	Type of dataset used	Studies	Frequency
1	Cross-project dataset	S1, S2, S3, S4, S5, S6, S7, S9, S10, S11, S12, S13, S14, S15, S16	15
2	within project dataset	No studies	0

CHAPTER FIVE
REPLICATION STUDY

This section describes the methodology followed including the referred datasets and steps used to prepare the multi-class dataset by using the instances of two referred datasets. Then, it presents the application of selected machine learning techniques on the target dataset. Hence, a set of experiments conducted were presented.

5.1 Replication Study

Replication is the process of recreating a research study, usually with various scenarios and individuals, to see if the core conclusions of the original study can be applied to different parties and situations [66]. Replication study aims at repeating a study's data and methodology in order to prove or disprove whether a result of a given study can have the same effect in another environment. A researcher can use the same dataset to repeat a study and see if other methodologies can affect the dataset in the same way. Similarly, a researcher can use the same methodology used by the reference work and apply it on other dataset to check if the given methodology can influence the new dataset similarly. Replication study enables a methodological or subject dependency comparison between of a reference work and one's original work.

5.2 The smells Dataset

5.2.1 Code smell Dataset Information

The input for code smell dataset is an instance of a method or a class containing fragment of code. A set of software metrics for each code instances are then computed manually. Additionally, their final smell category will be detected using either manual detection method or using a detection tool. There are different software metrics. The most commonly applied software metrics according to [66] are Size, Complexity, Coupling, Encapsulation and Inheritance are presented in the table below. The use of those metrics differs from one code smell type to other. But the method level smells dataset incorporates most of the metrics calculated in class, package and project level due to the containment relation. The containment relation defines that a method is contained in a class, a class is contained in a package, and a package is contained in a project [66]. But this work focuses on method-based code smell instances hence, excludes metrics of class, package, and project.

Table 5. 1 Software metrics and their category

No	Quality Dimension	Metric Label	Metric Name	Data Type	Granularity
1	Size	LOC	Lines of Code	Numeric	Project, Package, Class, Method
		LOCAMM	Lines of Code Without Accessor or Mutator Methods	Numeric	Class
		NOPK	Number of Packages	Numeric	Project
		NOCS	Number of Classes	Numeric	Project, Package
		NOM	Number of Methods	Numeric	Project, Package, Class
		NOMNAMM	Number of Not Accessor or Mutator Methods	Numeric	Project, Package, Class
		NOA	Number of Attributes	Numeric	Class
2	Complexity	CYCLO	Cyclomatic Complexity	Numeric	Method
		WMC	Weighted Method Count	Numeric	Class
		WMCNAMM	Weighted Method Count of Not Accessor or Mutator Methods	Numeric	Class
		AMW	Average Methods Weight	Numeric	Class
		AMWNAMM	Average Methods Weight of Not	Numeric	Class

			Accessor or Mutator Methods		
		MAXNESTING	Maximum Nesting Level	Numeric	Method
		CLNAMM	Called Local Not Accessor or Mutator Methods	Numeric	Method
		NOP	Number of Parameter	Numeric	Method
		NOAV	Number of Accessed Variable	Numeric	Method
		ATLD	Access to Local Data	Numeric	Method
		NOLV	Number of Local Variable	Numeric	Method
3	Coupling	FANOUT	Numbers of modules that called by a given module.	Numeric	Class, Method
		FANIN	Number of modules that call a given module.	Numeric	Class
		ATFD	Access to Foreign Data	Numeric	Method
		FDP	Foreign Data Provider	Numeric	Method
		RFC	Response for a Class	Numeric	Class
		CBO	Coupling between Object Classes	Numeric	Class
		CFNAMM	Called Foreign Not Accessor or Mutator Methods	Numeric	Class, Method
		CINT	Coupling Intensity	Numeric	Method
		MaMCL	Maximum Message Chain Length	Numeric	Method

		MeMCL	Mean Message Chain Length	Numeric	Method
		NMCS	Number of Message Chain Statements	Numeric	Method
		CC	Changing Classes	Numeric	Method
		CM	Changing Methods	Numeric	Method
		CDISP	Coupling Dispersion	Numeric	Method
4	Encapsulation	NOAM	Number of Accessor Methods	Numeric	Class
		NOPA	Number of public Attributes	Numeric	Method, Class
		LAA	Locality of Attribute Access	Numeric	Method
5	Inheritance	DIT	Depth of Inheritance Tree	Numeric	Class
		NOI	Number of Interfaces	Numeric	Project, Package
		NOC	Number of Children	Numeric	Class
		NMO	Number of Methods Overridden	Numeric	Class
		NIM	Number of Inherited Methods	Numeric	Class
		NOII	Number of Implemented Interfaces	Numeric	Class

5.2.2 Software systems selected for dataset preparation

In this work, two method-level datasets (long method and feature envy) have been considered. These two datasets were originally prepared by Fontana at [22]. Then they were made available for other researchers by [86]. The researchers have prepared the datasets in a form that can be applied directly by any researcher. The researchers [66] have pointed out that Fontana [22] have analyzed the Qualitas Corpus software systems which was collected from [108]. Originally, the Qualitas Corpus consists 111 systems but, 74 systems are considered for smell detection. The remaining 37 systems cannot detect code smells as they are not successfully compiled [66]. The size (lines of code etc…) of the 74 Java projects are shown in table below. These projects also cover different application domains like database, tool, middleware, and games. The complete characteristics (sizes, release date, etc.) of each project and the domain they belong to is also made available online by the researcher.

Table 5. 2 Summary of the 74 open-source software systems from Qualitas corpus

Number of Projects	Number of lines in all projects	Number of packages in all projects	Number of classes in all projects	Number of methods in all projects
74	6,785,568	3420	51,826	404,316

4.2.3 Reference datasets

The datasets covering a variety of code smell types have been prepared by using automatic code smell detection tools to detect whether the source code element is smelly or not. Fontana [22] have computed metrics at all levels by using the tool “Design Features and Metrics for Java” (DFMC4J). According to [66] this tool parses the source code of Java projects through the Eclipse JDT library. Then the computed metrics become attributes of the datasets. PMD, Fluid Tool and iPlasma are the automatic code smell detection tools used to build code smell datasets. These datasets are available at http://www.essere.disco.unimib.it/machine-learning-for-code-smell-detection/. The datasets contain a set of software metrics computed and the final class describing the smelliness of a given instance as smelly or non-smelly which was previously detected by Fontana, F.A. et al., at [22]. In the two method level datasets (Long Method and Feature Envy) that are made available by Guggulothu, T. at [66], there are around 840 code instances. Each dataset contributing 420 instances. Additionally, among the 420 instances, 1/3 (140) are smelly and 2/3 (280) are non-smelly.

Table 5. 3 Smell distribution of the two original method level datasets

Total number of Instances	The Method level datasets			
	Long Method		Feature Envy	
	Smelly instances	Non-Smelly instances	Smelly instances	Non-Smelly instances
840	140	280	140	280

These datasets are composed of common instances that share common metrics and different instances that belong to either of the two datasets exclusively. But the metric distribution in those two datasets is equal. Both containing the same metrics throughout their dataset. According to Marinescu, R. at [109] these two smells also cover different object-oriented quality dimension problems such as complexity, size, coupling, encapsulation, cohesion, and data abstraction.

The main reason for choosing the datasets is, according to the researcher at [110], these code smells are more frequent and fault prone or change prone as per literature. It has also been reported that there is a strong correlation among the smells when detecting the code smell by the reference work.

- **Long method:** When a method has a lot of lines of code and needs a lot of data from other classes, it's called a long method. This increases the functional complexity of the method and it will be difficult to understand [66]. It represents a large method that implements more than one function [86].
- **Feature envy:** Feature Envy is a code smell that develops when a method utilizes much more data than another class in comparison to the one it is in [86].

According to [66], many studies have considered the detection of the existence a single code smell in a given method instance with the help of ML classifiers. That is the detection of an instance to be either Non- smelly or Smelly of a single code smell, making it less reliable. Hence, they planned to formulate a different approach which is a multi-label classification (MLC) problem using three multi label problem transformation methods and proposed the use of Multi-Labeled Dataset (MLD). Their study considers the two method level smells Long Method and Feature Envy. This initial study led to the idea of merging the datasets in one of the best ways and form a Multi-Class Dataset containing the existence of the two code smells and their intersection introducing better label correlation.

5.3 Limitations of the reference work

According to Di Nucci, D. et al. at [86], most of the ML techniques have achieved a higher result in terms of accuracy and F-measure result. Especially J48 and Random Forest classifiers higher than 95%. The researchers have pointed out two major aspects that they believe would influence the performance of the classification algorithms. The first is the representation of instances in the dataset and the second is the use of balanced dataset.

- The preparation in terms of representation of instances to be used in the dataset
  Almost all the datasets prepared by different researchers represent the occurrence of a single type of smell in a given code instance. In fact, there is a probability for a given code instance to be affected by more than one code smell type. This scenario is very unrealistic and might be the reason behind the very high result. The classification algorithm may find it easier to discriminate the smelly instances from the non-smelly one.
- Balanced dataset
  According to recent findings on the smelliness of code smells, a small fraction of a software system is usually affected by code smells, resulting in bias towards the largest class label. In order to alleviate the data imbalance issue, the researchers have assessed the possibility of a different balancing techniques between smelly and non-smelly instances.

Hence, based on the idea of the reference work, it has been indicated that multi class transformation can genuinely represent label correlation compared to the widely known binary relevance method. As a result, this study also merged datasets using one class label containing four classes.

The total number of metrics used by the reference work is also very large in number. This aggregation of less relevant attributes could cause in complexity which in turn can result the emergence of a less efficient model.

So, this study tries to address the above listed issues by introducing a multi class dataset, a more pure dataset (fewer attributes and class balancing approach) and using ML algorithms from different categories of classifiers selected through mapping study that has not previously been jointly studied by any practitioner.

5.4 The process of Multi-Class Dataset Preparation

In order to prepare the Multi-Class Dataset containing the existence of two code smell types, this study has used two different and stand-alone datasets mentioned above. Those two Datasets are:-

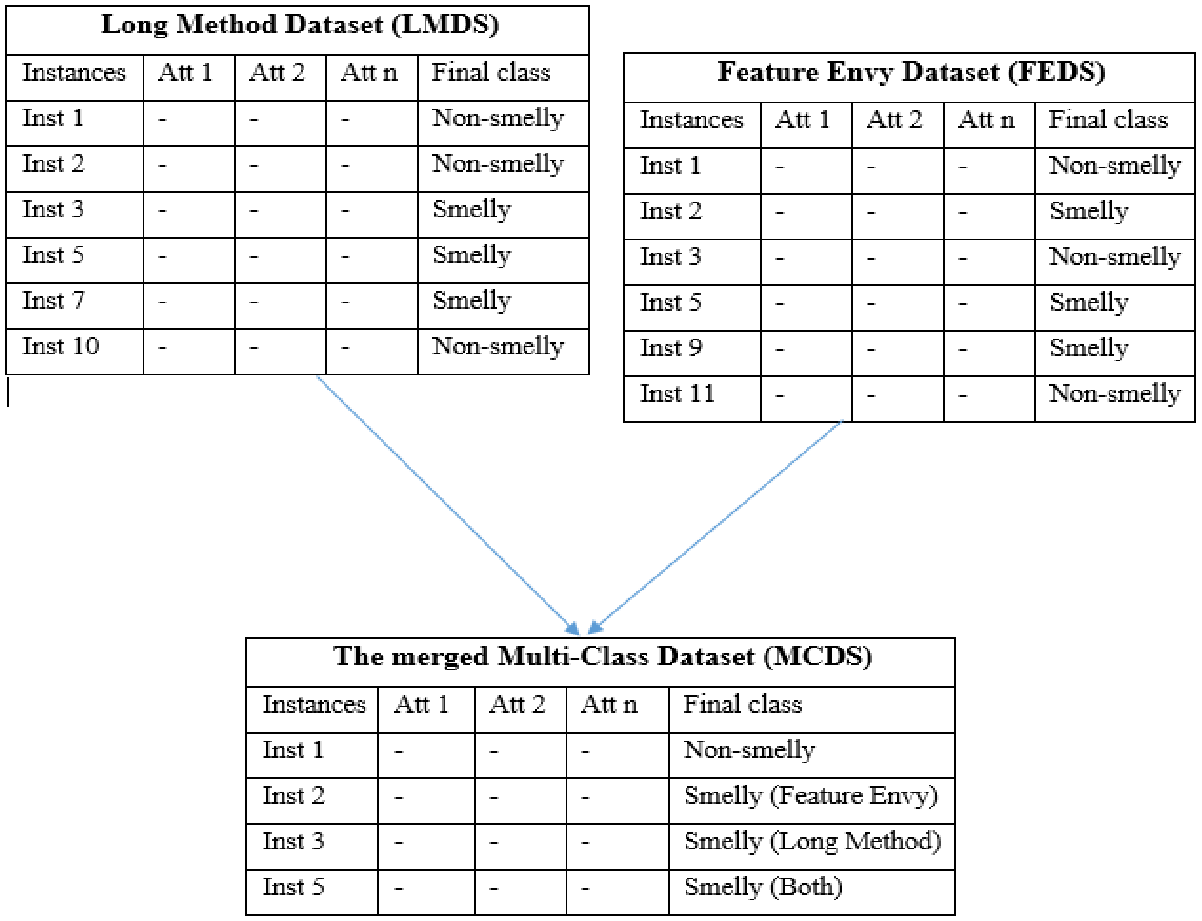
- The dataset containing the existence of a Long Method code smell (which is represented as smelly and Non-smelly).
- The dataset containing the existence of a Feature Envy code smell (again represented as smelly and Non-smelly) for each corresponding instances.

Having the binary class dataset, the study considers the common instances (that belong to both datasets) that are available in both Datasets and consider their final class label as the final result of the labeling. So, here is the rule followed by this study to prepare the MCD. But as a precondition the instances should be available in both Datasets. So, before all those rules below, it is checked that the precondition is met.

- If an instance is labeled as Non-smelly in both Long Method and Feature Envy Datasets, then it is labeled as Non-smelly.
- If an instance is labeled as smelly in Long Method and Non-smelly in Feature Envy Dataset, then it is labeled as Long Method.
- If an instance is labeled as Non-smelly in Long Method and smelly in Feature Envy Dataset, then it is labeled as Feature Envy.
- If an instance is labeled as smelly in both Long Method and Feature Envy Datasets, then it is labeled as Both (instance consisting the existence of the two code smells).

The reason for picking the common instances is that it is hard for the instances that are not available in the other Datasets to label as the value they contain in the Dataset containing them. This would raise a disparity problem [66], resulting in the confusion of the learning technique and preventing it from clearly distinguishing the difference between the one that is solely true code smell with the one that has the probability of being true or false.

For instance, if a given instance is available as positive (smelly) in the LM Dataset and doesn't exist in the FE Dataset, it shouldn't be considered as an instance containing LM only. When the fact is that this instance isn't considered in the FE doesn't guarantee that it's free from the FE code smell. This shows that there is a probability for this instance to become affected by FE code smell and should be considered as both. So, to keep this clarity and avoid further confusion for the learner, it is a best choice to discard such instances. Here is how the binary class Dataset are merged in to form the Multi-Class Dataset.



Long Method Dataset (LMDS)				
Instances	Att 1	Att 2	Att n	Final class
Inst 1	-	-	-	Non-smelly
Inst 2	-	-	-	Non-smelly
Inst 3	-	-	-	Smelly
Inst 5	-	-	-	Smelly
Inst 7	-	-	-	Smelly
Inst 10	-	-	-	Non-smelly

Feature Envy Dataset (FEDS)				
Instances	Att 1	Att 2	Att n	Final class
Inst 1	-	-	-	Non-smelly
Inst 2	-	-	-	Smelly
Inst 3	-	-	-	Non-smelly
Inst 5	-	-	-	Smelly
Inst 9	-	-	-	Smelly
Inst 11	-	-	-	Non-smelly

The merged Multi-Class Dataset (MCDS)				
Instances	Att 1	Att 2	Att n	Final class
Inst 1	-	-	-	Non-smelly
Inst 2	-	-	-	Smelly (Feature Envy)
Inst 3	-	-	-	Smelly (Long Method)
Inst 5	-	-	-	Smelly (Both)

Figure 5. 1 Multi Class Dataset preparation

Where, Inst stands for code instance, Att for Attribute and – representing the value an attribute holds for a given instance.

The instances Inst 7, Inst 9, Inst 10 and Inst 11 are exclusive, i.e. the instance in one Dataset doesn’t belong to the other Dataset and it’s hard to generalize the real class label of such data blindly, hence are rejected in the Multi-Label Dataset.

So, the major motive behind the preparation of the Multi-Class Dataset is that a single instance (method) may be affected by different code smells. The fact that an instance is not affected by a given smell doesn’t necessarily indicate it is not affected by the other/s or is free from any other code smells. Hence the Multi-Label Dataset used in this study increases the representativeness of the real-world scenario compared to the single class smell Dataset.

5.5 Design Metrics Definition and Computation Details

The smells dataset adopted in this study consists of a set of software metrics that are computed by the researcher at [22]. Some design metrics are stand alone and their value is derived directly from the origin (source code). While in case of some others, the result of a given design metric can depend upon one or more design information gained from other design metrics. So, in order to calculate the value of some metrics there is a need to consider other independent metrics. Hence, in order to keep the clarity, the metrics are presented according to the information flow required. This study has considered 55 initial design metrics at different levels. The metrics used along with their acronym in the original csv dataset and their computation detail is presented in the Appendix section of this study.

5.6 Experimental setup

5.6.1 Data preprocessing

A total of 395 common instances formed the MCD. Originally there were 82 software metrics (attributes) and a final class label consisting of four classes. The class distribution in this dataset is highly imbalanced. There are four classes in the dataset. The one that contains the existence of Non-smelly instances, the one that contains the existence of Feature Envy instances, the one that contains the existence of Long Method instances and the one that contains the existence of both Feature Envy and Long Method instances. Those classes contribute 223, 41, 46 and 85 instances respectively to the dataset. The Non-smelly instances still hold the largest proportion in the dataset similar to the reference datasets. While the rest classes cover a small proportion making it highly imbalanced. Additionally, some attributes in the dataset contains missing values. The experiments are conducted in WEKA which is one of the powerful data preprocessing and synthesizing tool [86].

5.6.1.1 Initial data preparation

Most real-world datasets contain different impurities. Those impurities could lead to wrong and complex findings and should be addressed well. Data cleaning is a process of preparing a purer data that should be fed to the learning algorithms. Data cleaning and preparation is an essential and prior step in the classification task. Properly prepared dataset provides a better result. The data cleaning process in this specific study incorporates data balancing and attribute filtration and selection. The dataset doesn’t contain any missing values for the selected metrics.

As mentioned above, the original dataset contains a total of 88 columns. But the task of classification becomes complex with the increased number of attributes. What is more is that all those attributes don’t have equal relevance to the classification purpose. So, it is better to discard such attributes. With this idea, in the first round five attributes were reduced because they don’t have any relevance to the classification purpose. After the removal of those attributes, the dataset was left with 82 attributes and the class label. But the dataset still requires further filtration. According to [66], the method level software metrics computed around the five object-oriented quality dimensions are the very critical attributes to consider. Accordingly, this study has filtered additional 36 attributes and discard them in the second round. Finally resulting in 46 columns containing the basic software metrics at method level. The detailed data preparation is presented as follows.

5.6.1.2 Attribute Selection

Another type of impurity a dataset encounter is a set of attributes that might not be strongly tied to the value of the final class. Such attributes (columns) increase the dimensionality of the dataset but may not have an importance. Hence, the dataset become unmanageable and complex because of them. On the other side reduced number of features increases understandability. Those irrelevant attributes (attributes that could not be helpful for some tasks of classification or other purpose) should be removed prior to application of any technique. Removal of those attributes will be required to come up with reduced number of attributes. Hence, from the original dataset containing 82 attributes, 36 (25, 5 and 6) attributes were filtered out as they were software metrics that are calculated at the level of class, package and project respectively. Those attributes are irrelevant to the method-level code smells because they do not contribute to detection of the method level code smells. Additionally, [66] has reported that that Method metrics can cover all the structural information (coupling of other classes, etc.) of the methods.

But, even after the initial removal of attributes, there are still a number of attributes with different level of importance. Attribute ranking is one of those things that can really improve the performance of a machine learning models. The rest 46 attributes and one final class were feed to the WEKA to check their relevance with respect to their contribution towards the class label. All attribute are not equally relevant for predicting the target. So, we need to first evaluate the usefulness of each attribute before conducting any experiment. In order to check the order of relevance of those attributes, this study has proposed the use of one of the most popular attribute selection methods, information gain technique.

Information Gain is an attribute ranking technique that tells how important a given attribute is and helps to decide the order of attributes in the node. It is calculated based on entropy value of an attribute. It is the ID3 algorithm calculates a sample's homogeneity using entropy. The entropy of a perfectly homogenous sample is zero, while the entropy of an evenly divided sample is one. The weighted Entropy of each attribute is subtracted from the original (class) Entropy to calculate Information Gain. Let D be the training data and C the different classes of the dataset D. The general working of this selection method is presented below.

First Calculate the Expected information (entropy) needed to classify an instance in D

$$\textbf{Entropy (E (D))} = \sum_{i=1}^{M} Pi\ Log2\ (Pi)$$

Where pi denotes the probability that each given instance in D (training data) belongs to class C1, as determined by |C1, D|/|D|.

Entropy (E (D)) is the average amount of information required to determine an instance's class label in D.

Then Calculate the information needed of an attribute. Assume that attribute A can be utilized to divide D into n partitions or subsets, D1, D2,..., Dn, with Dj containing those occurrences in D that have A result aj. To classify D, the following information is needed (after using A to split D):

$$\text{Info}_A\text{ (D)} = \sum_{J=1}^{V} \frac{Dj}{D} X\ Info(DJ)$$

The purity of the partitions increases as the predicted information required decreases. The following is the information obtained by branching on attribute A:

$$\text{Gain (A)} = \text{E (D)-Info}_A\text{(D)}$$

Generally, information gain increases with the increase in the average purity of the subsets. As the splitting attribute, the attribute with the highest information gain among the attributes is chosen. After the information gain for each attribute is calculated, the one with the highest information gain will be selected as the root node, and the calculation will be continued recursively until the data is completely classified. Indicating that, when utilizing these measures (metrics) to train a decision tree, the best split is determined by maximizing Information Gain. Here is the information gain value of the attributes.

Table 5. 4 Attribute ranking using Information Gain

No	**Attribute Name**	**Information Gain Value**
1	NOAV_method	1.00707
2	LOC_method	0.98359
3	CYCLO_method	0.97812
4	ATFD_method	0.86311
5	MAXNESTING_method	0.8582
6	NOLV_method	0.82567
7	FDP_method	0.73726
8	CINT_method	0.72594
9	LAA_method	0.69413
10	CDISP_method	0.60157
11	CFNAMM_method	0.59797
12	FANOUT_method	0.54763
13	CLNAMM_method	0.25539

14	NOP_method	0.23748
15	ATLD_method	0.18219
16	MaMCL_method	0.11764
17	NMCS_method	0.11222
18	MeMCL_method	0.11222
19	number_standard_design_methods	0.10896
20	number_not_abstract_not_final_methods	0.10888
21	number_private_visibility_attributes	0.10485
22	number_not_final_not_static_methods	0.1039
23	num_final_attributes	0.10337
24	number_private_visibility_methods	0.08788
25	num_static_attributes	0.08565
26	num_final_static_attributes	0.07049
27	num_not_final_not_static_attributes	0.07001
28	num_final_not_static_attributes	0.06751
29	number_protected_visibility_methods	0.05871
30	number_static_methods	0.04515
31	number_package_visibility_methods	0.04278
32	number_package_visibility_attributes	0.04027
33	CC_methods	0

34	CM_methods	0
35	is_static_methods	0
36	is_static_type	0
37	number_not_final_static_methods	0
38	number_constructor_Defaultconstructor_methods	0
39	number_final_static_methods	0
40	number_final_not_static_methods	0
41	number_abstract_methods	0
42	number_constructor_NotDefaultconstructor_methods	0
43	number_static_not_final_attributes	0
44	number_public_visibility_methods	0
45	number_final_methods	0
46	number_protected_visibility_attributes	0

The attributes with higher information gain value are believed to have a better contribution to the prediction. This study has used a default threshold value to discriminate the attributes and selected the most important attributes because, it doesn't mean all attributes are equally relevant to the classification task. So, the study has removed the attributes with information gain value "0" Hence this study uses a total of 32 attributes after the removal of fourteen attributes. Then the task of classification is held according to the significance level (ranking) of the attributes.

5.6.1.3 Data Balancing

The other most important data preparation process is Data balancing. Data balancing can be applied to the datasets where there is an imbalanced proportion among the class labels. Data balancing is a very essential step prior to any classification task. The process of data balancing enables a better proportion among the final classes. This helps the minority classes from being mistreated by the classification algorithm. As described earlier, there is a higher-class imbalance among the classes of the adopted dataset. This is because the code smell dataset contains a higher proportion of Non-smelly instances.

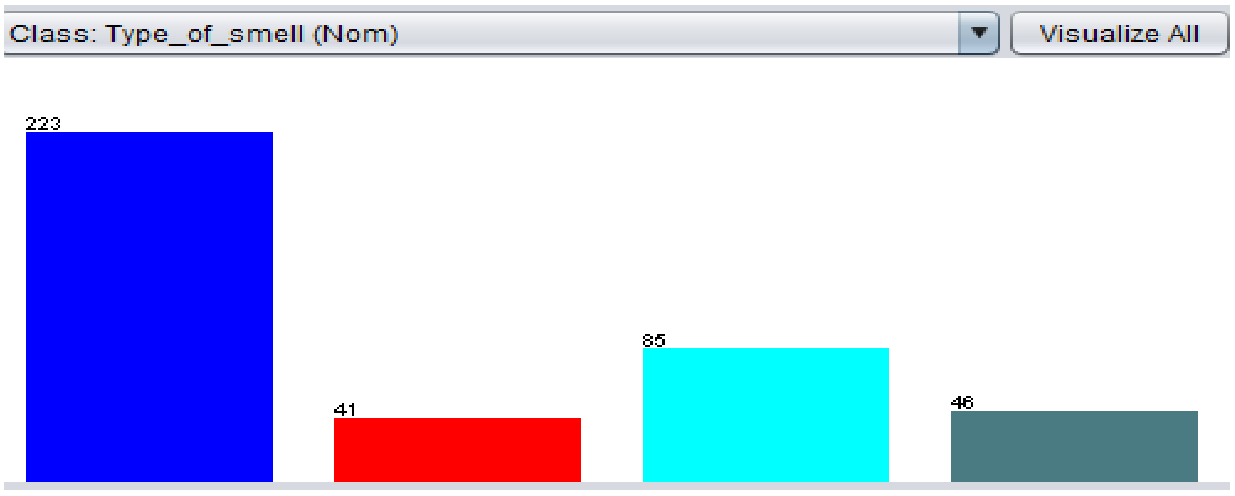


Figure 5. 2 Class proportion of the original dataset

In order to alleviate this data imbalance problem, the study has proposed the use of SMOTE technique. SMOTE is an oversampling technique that generates synthetic samples from the minority class [91]. SMOTE gives attention to minority classes and increases their proportion in the dataset helping them to be equally treated with majority class. All the classes except the Non-smelly are considered as minority classes in this study. So, in the first stage of sampling the study has applied all the default parameters to the three class labels resulting in doubling of the instances. The class “Both” seemed balanced with the first class “Non-smelly”. But, the rest two classes are still minor compared to the others.

The balancing shows improvement compared to the original one. But, the sampling process still requires additional SMOTE to the second and forth classes. Hence the study undergoes the process again to balance those two with the others with all default parameters.

It is required for the result after SMOTE to not interfere with the original appearance of classes in the dataset. So, even if the classes seem quite balanced, the number of instances in the class “Long Method” were less than that of the instances in class “Both”. In order to maintain the consistency of appearance of classes in the original data, the study apples SMOTE again to the class “Both”. But, this time with the percentage value (parameter) degraded to 10 so that it won’t have larger value than the first class “Non-smelly”.

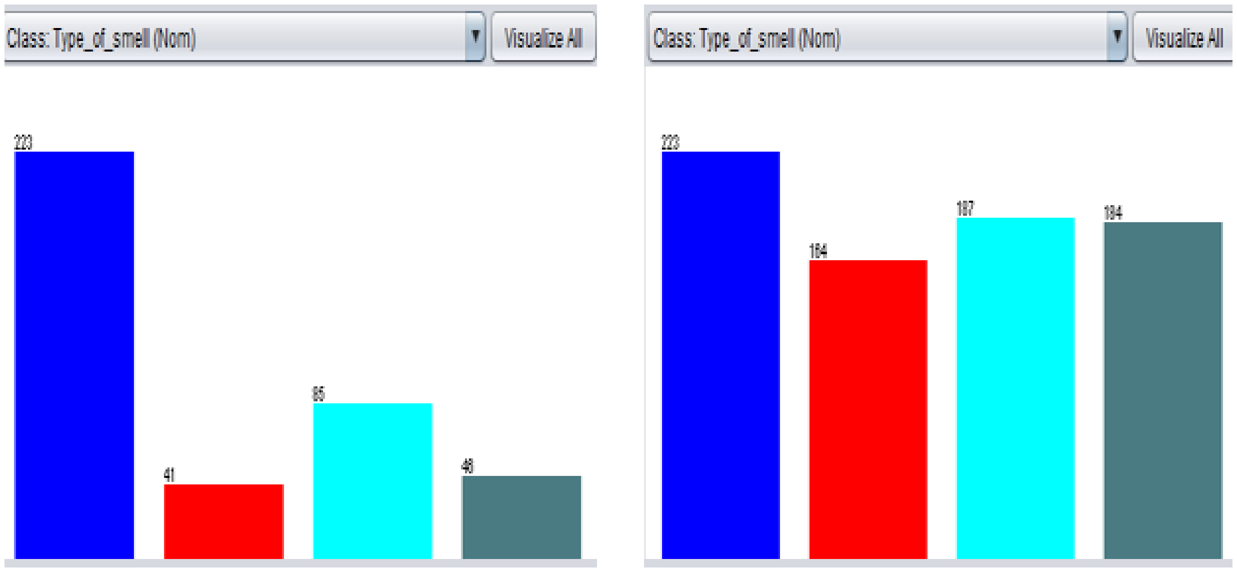


Figure 5. 3 Original Class distribution Vs Class distribution after SMOTE

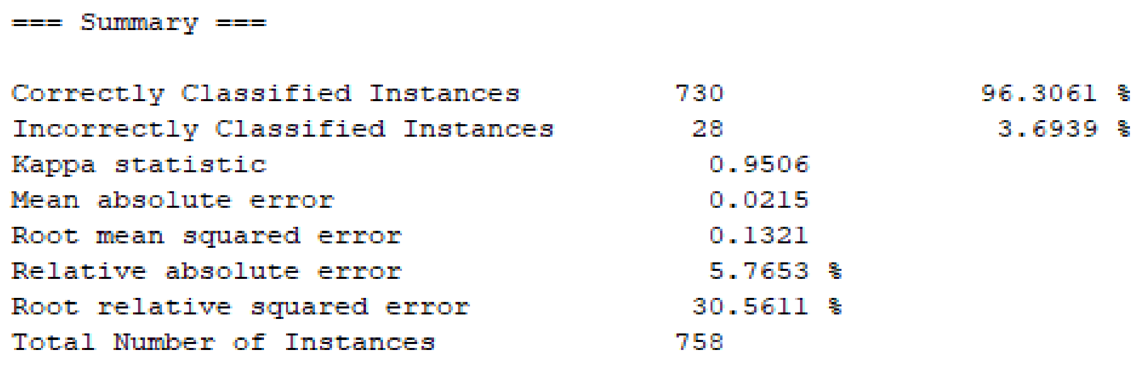
The above figure depicts that the classes are now balanced and also maintain their original appearance. A well-balanced data provides a more concrete result. The task of classification then will be applied on the new balanced dataset.

5.7 Experiments and Experimental Results

After the dataset has been properly preprocessed, the next step will be feeding the data to the classification algorithms. In the third chapter, this study was able to point out which techniques are most commonly used in the detection of the most widely studied code smells. Since the experimental work of this study focuses on the two method-level smells (Long Method and Feature Envy), it considers the most commonly applied machine learning techniques to detect them. Accordingly, by referring to the earlier chapter, Random Forest, Tree based, Naive Bayes are the most commonly used ML techniques while the higher result for the Long Method smell is gained through RF and Tree based classifiers. This study has presented the experiments under different scenarios. All experiments are conducted under 10-fold cross validation. The main reason is that the size of the dataset is quite small in size to perform further split. Therefore, it is believed that this splitting technique can represent the dataset well.

5.7.1 Experiment 1 Application of J48 Technique

The first experiment is run using the J48 algorithm. This experiment is conducted under the 10-fold cross validation and has used all the default values of J48 algorithm in WEKA. The result of the experiment shows that from the total 758 instances, the algorithm was able to classify 730 (96.3061 %) instances accurately. The number of incorrectly classified instances is 28 (3.6939 %).



```
=== Summary ===

Correctly Classified Instances         730               96.3061 %
Incorrectly Classified Instances        28                3.6939 %
Kappa statistic                          0.9506
Mean absolute error                      0.0215
Root mean squared error                  0.1321
Relative absolute error                  5.7653 %
Root relative squared error             30.5611 %
Total Number of Instances              758
```

Figure 5. 4 Summary of performance of J48

5.7.1.1 The detailed information by class of J48 algorithm

Table 5. 5 The detailed information by class of J48 algorithm

Detailed information by class						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class
	0.982	0.004	0.991	0.982	0.986	Non smelly
	0.963	0.012	0.958	0.963	0.960	Feature Envy
	0.941	0.014	0.957	0.941	0.949	Both
	0.962	0.019	0.941	0.962	0.952	Long Method
Weighted Avg.	0.963	0.012	0.963	0.963	0.963	

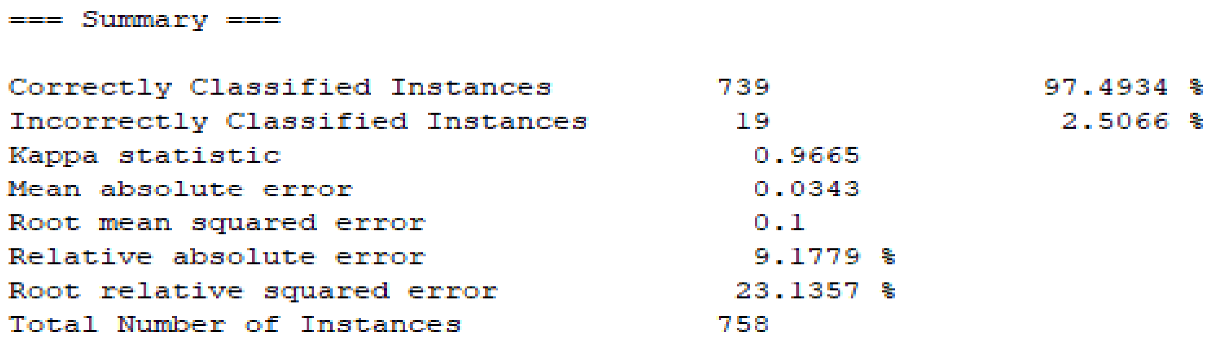
5.7.1.2 Confusion Matrix for J48 algorithm

Table 5. 6 Confusion Matrix for J48 algorithm

Confusion Matrix				
a	B	C	D	Classified as
219	3	0	1	a = Non smelly
1	158	3	2	b = Feature Envy
0	3	176	8	c = Both
1	1	5	177	d = Long Method

5.7.2 Experiment 2 Application of Random Forest Technique

The second experiment applies Random Forest algorithm. From the total 758 instances, the algorithm was able to detect 739 instances correctly. The rest 19 instances were incorrectly classified. The algorithm was able to classify 97.4934 % of instances accurately from the total.



```
=== Summary ===

Correctly Classified Instances         739               97.4934 %
Incorrectly Classified Instances        19                2.5066 %
Kappa statistic                          0.9665
Mean absolute error                      0.0343
Root mean squared error                  0.1
Relative absolute error                  9.1779 %
Root relative squared error             23.1357 %
Total Number of Instances              758
```

Figure 5. 5 Summary of performance of Random Forest

5.7.2.1 The detailed information by class of Random Forest algorithm

Table 5. 7 The detailed information by class of Random Forest algorithm

Detailed information by class						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class
	0.964	0.002	0.995	0.964	0.979	Non smelly
	0.982	0.010	0.964	0.982	0.973	Feature Envy
	0.973	0.009	0.973	0.973	0.973	Both
	0.984	0.012	0.963	0.984	0.973	Long Method
Weighted Avg.	0.975	0.008	0.975	0.975	0.975	

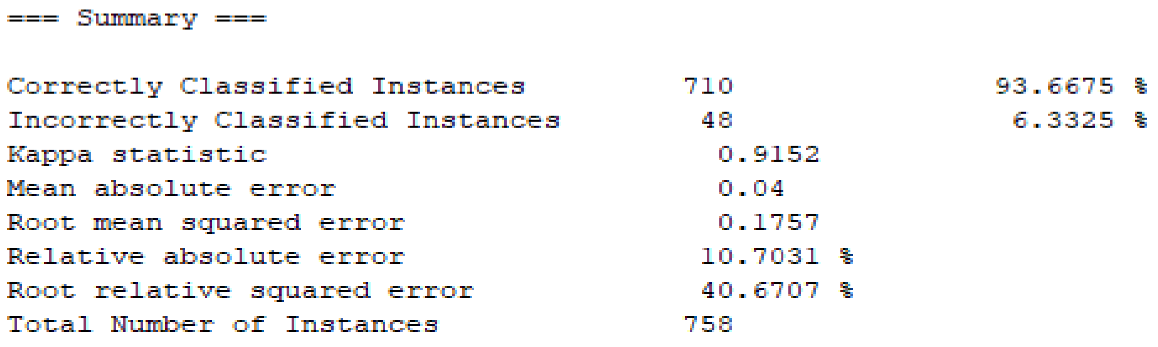
5.7.2.2 Confusion Matrix for Random Forest algorithm

Table 5. 8 Confusion Matrix for Random Forest algorithm

Confusion Matrix				
A	B	C	D	Classified as
215	6	0	2	a = Non smelly
0	161	3	0	b = Feature Envy
0	0	182	5	c = Both
1	0	2	181	d = Long Method

5.7.3 Experiment 3 Application of JRip Technique

The third experiment applies JRip algorithm with all the default parameters. The algorithm was able to distinguish 710 of out of 758 instances correctly. In terms of accuracy the algorithm was able to score an accuracy rate of 93.6675 %. The rest 48 instances (6.3325 %) were incorrectly classified by the algorithm.



```
=== Summary ===

Correctly Classified Instances         710               93.6675 %
Incorrectly Classified Instances        48                6.3325 %
Kappa statistic                          0.9152
Mean absolute error                      0.04
Root mean squared error                  0.1757
Relative absolute error                 10.7031 %
Root relative squared error             40.6707 %
Total Number of Instances              758
```

Figure 5. 6 Summary of performance of JRip

5.7.3.1 The detailed information by class of JRip algorithm

Table 5. 9 The detailed information by class of JRip algorithm

Detailed information by class						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class
	0.960	0.024	0.943	0.960	0.951	Non smelly
	0.915	0.012	0.955	0.915	0.935	Feature Envy
	0.936	0.028	0.916	0.936	0.926	Both
	0.929	0.021	0.934	0.929	0.932	Long Method
Weighted Avg.	0.937	0.022	0.937	0.937	0.937	

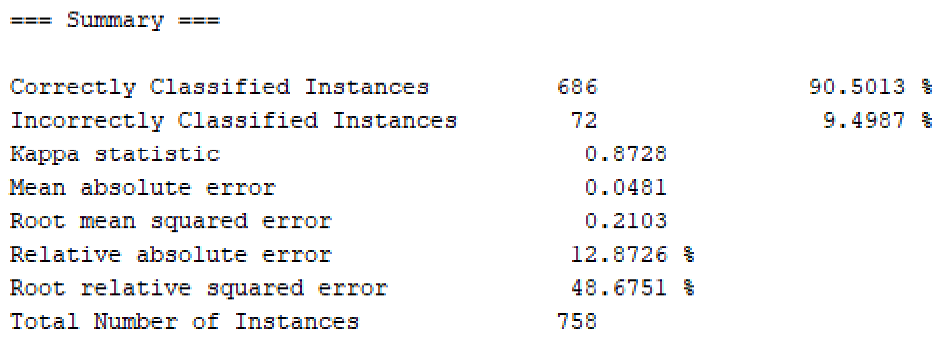
5.7.3.2 Confusion Matrix for JRip algorithm

Table 5. 10 Confusion Matrix for JRip algorithm

Confusion Matrix				
A	B	C	D	Classified as
214	5	1	3	a = Non smelly
9	150	5	0	b = Feature Envy
2	1	175	9	c = Both
2	1	10	171	d = Long Method

5.7.4 Experiment 4 Application of Naïve Bayes Technique

The fourth experiment applies Naïve Bayes and with this algorithm a percentage of 90.5013 % instances were classified accurately. Indicating that from the total 758 instances, 686 of them were detected correctly. The rest 72 instances were classified incorrectly.



```
=== Summary ===

Correctly Classified Instances         686               90.5013 %
Incorrectly Classified Instances        72                9.4987 %
Kappa statistic                          0.8728
Mean absolute error                      0.0481
Root mean squared error                  0.2103
Relative absolute error                 12.8726 %
Root relative squared error             48.6751 %
Total Number of Instances              758
```

Figure 5. 7 Summary of performance of Naïve Bayes

5.7.4.1 The detailed information by class of Naïve Bayes algorithm

Table 5. 11 The detailed information by class of Naïve Bayes algorithm

Detailed information by class						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class
	0.928	0.015	0.963	0.928	0.945	Non smelly
	0.854	0.019	0.927	0.854	0.889	Feature Envy
	0.893	0.053	0.848	0.893	0.870	Both
	0.935	0.040	0.882	0.935	0.908	Long Method
Weighted Avg.	0.905	0.031	0.907	0.905	0.905	

5.7.4.2 Confusion Matrix for Naïve Bayes algorithm

Table 5. 12 Confusion Matrix for Naïve Bayes algorithm

Confusion Matrix				
A	B	C	D	Classified as
207	6	4	6	a = Non smelly
8	140	14	2	b = Feature Envy
0	5	167	15	c = Both
0	0	12	172	d = Long Method

The confusion matrix of Naïve Bayes shows that 207 out of 223 instances of "Non-smelly" class were classified in their true class. In the second-class label "Feature Envy", 140 instances were classified in their actual class. From the instances of third class "Both", 167 of them were classified accurately. Finally, from the fourth class "Long Method", 172 of them were accurately classified. So, totally 686 instances from the dataset were classified correctly.

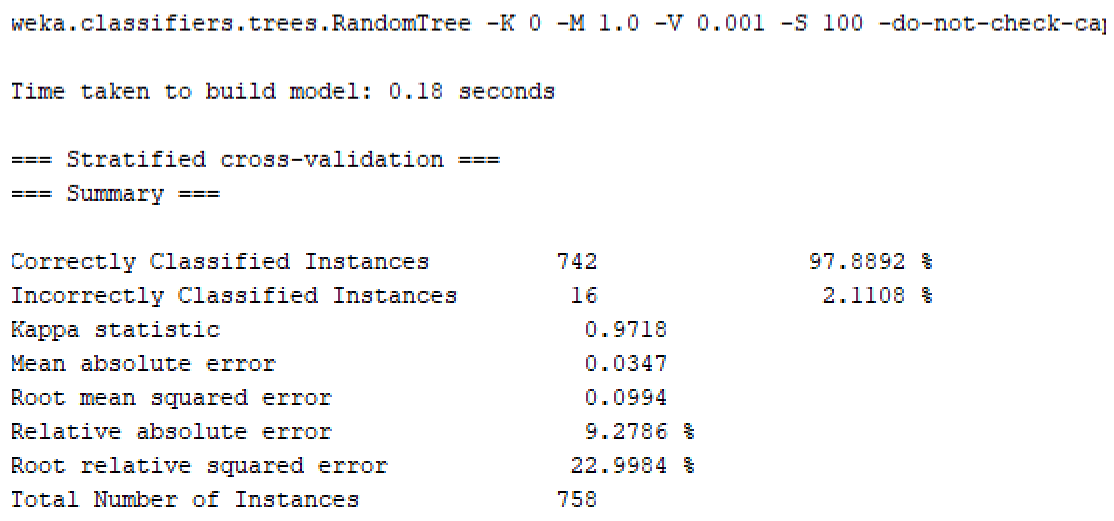
Finally, the result of the experiments show that Random Forest scores best compared to the rest. But the study further proposes to enhance the performance of the best learning technique (Random Forest). Hence, planned to change parameters and check if they can improve the result gained through default parameters. By arranging the seed size to different values, the study checks for better results. Seed size is a random value to be set for a given algorithm. The result of experiments with different seed size is presented below.

5.7.5 Experiment 5 Applying Random Forest with different seed size

Table 5. 13 Random Forest performance with different seed size

Scenarios for RF	**Seed size**	TP Rate	FP Rate	Precision	Recall	F-Measure	Accuracy
Default	1	0.975	0.008	0.975	0.975	0.975	97.4934
Scenario 1	10	0.978	0.007	0.978	0.978	0.978	97.7573
Scenario 2	**100**	**0.979**	**0.007**	**0.979**	**0.979**	**0.979**	**97.8892**
Scenario 3	1000	0.972	0.009	0.973	0.972	0.972	97.2296

The table above depicts that with different seed sizes the performance of the RF algorithm changes. Accordingly, the best result was gained through the last Scenario (Scenario 2). The detailed information of this Scenario is presented below.



```
weka.classifiers.trees.RandomTree -K 0 -M 1.0 -V 0.001 -S 100 -do-not-check-ca

Time taken to build model: 0.18 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         742               97.8892 %
Incorrectly Classified Instances        16                2.1108 %
Kappa statistic                          0.9718
Mean absolute error                      0.0347
Root mean squared error                  0.0994
Relative absolute error                  9.2786 %
Root relative squared error             22.9984 %
Total Number of Instances              758
```

Figure 5. 8 Summary of performance of Random Forest with seed size 100

5.7.5.1 The detailed information by class of Random Forest with seed size 100

Table 5. 14 The detailed information by class of Random Forest with seed size 100

Detailed information by class						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class
	0.969	0.000	1.000	0.969	0.984	Non smelly
	0.982	0.008	0.970	0.982	0.976	Feature Envy
	0.979	0.009	0.973	0.979	0.976	Both
	0.989	0.010	0.968	0.989	0.978	Long Method
Weighted Avg.	0.980	0.007	0.979	0.980	0.979	

5.7.5.2 Confusion Matrix for Random Forest with seed size 100

Table 5. 15 Confusion Matrix for Random Forest with seed size 100

Confusion Matrix				
A	B	C	D	Classified as
216	5	0	2	a = Non smelly
0	161	3	0	b = Feature Envy
0	0	183	4	c = Both
0	0	2	182	d = Long Method

The Random Forest with seed size 100 was able to classify 742 (216, 161, 183 and 182) instances from Non smelly, Feature Envy, Both and Long Method classes respectively. The number of instances that are incorrectly classified is 16. So, 97.8892 % of instances has been correctly classified. The time taken to build the model is 0.18 sec.

5.8 Experimental Result Comparison

The table below presents the detailed and overall performance of each of the above algorithms in terms of the metrics defined in the earlier chapter.

Table 5. 16 Summary of average performance of all the algorithms

	Performance Metrics								
Classification Algorithm	Correctly classified instances	Incorrectly classified instances	Time taken to build the model	TP Rate	FP Rate	Precision	Recall	F-Measure	Accuracy
J48	730	28	0.14 sec	0.963	0.012	0.963	0.963	0.963	96.3061 %
Random Forest	739	19	0.49 sec	0.975	0.008	0.975	0.975	0.975	97.4934 %
JRip	710	48	0.33 sec	0.937	0.022	0.937	0.937	0.937	93.6675 %
Naive Bayes	686	72	**0.01 sec**	0.905	0.031	0.907	0.905	0.905	90.5013 %
Random Forest with seed size 100	**742**	**16**	0.18 sec	**0.979**	**0.007**	**0.979**	**0.979**	**0.979**	**97.8892 %**

According to the above table, the algorithm Random Forest out performs the rest machine learning techniques deployed by the study. With all default values of WEKA, from the table above, the algorithm Random Forest shows a higher result in terms of all the performance measurement metrics except for the time taken to build the model. The algorithm was able to detect 97.4934 % of the instances under their real class label. The percentage of instances that are labeled mistakenly as positive are 2.5066 % which is relatively lower than the other classifiers.

Additionally, the study has conducted another three scenarios under the second experiment to further enhance the performance of the selected algorithm. This results in the improvement of classification accuracy and the algorithm was able to gain higher result with different parameter. The results are the one's highlighted in bold. So, compared to the other classifiers, Random Forest algorithm is well suited given the specific MCD dataset.

CHAPTER SIX

FINDINGS OF THE STUDY

This section describes the major findings discovered through the Systematic literature review (SLR) and experimental work section. It also presents the experimental results of the reference work and the original work. Accordingly, a comparison between the two works will be presented.

6.1 Findings of the Systematic Literature Review

This study has applied a systematic literature review with the aim of exploring the code smells studied and the machine learning techniques deployed in the detection of code smells in recent years. Hence, the systematic Literature Review carried out in this study was able answer the first three research questions of the study. The result of the SLR is discussed as follows.

“Which code smells are most typically discovered using machine learning techniques?” was the study's first research question. Following a thorough review of the literature, it was discovered that the smells Long method, Feature envy, God class, and Data class are the code smells that have been studied the most between 2017 and 2020. Specifically, the Long method and Feature envy are the leading smells to be detected using ML techniques in recent years. Each of them was detected by 11 papers out of the selected 16 papers. The others God class and Data class were detected by 8 and 6 papers respectively. So, the answer to the first research question can be summarized as follows.

Research Question 1: Which code smells are most commonly detected using machine learning techniques?

Hence, the answer to the first research question of this study is, the code smells Long method, Feature envy, God class and Data class are the most widely studied code smells in the studies from 2017-2020. Additionally, according to the code smell category, Bloaters, couplers and Dispensable are the leading smell categories to be detected by the studies from 2017 to 2020.

The second research questions of this study were “Which machine learning techniques better applicable to detect code smells?”. This study conducts a full literature analysis on publications published between 2017 and 2020, in the same way as the first research question was answered. The result of the SLR shows that Random Forest, Decision tree, Naïve Bayes and SVM are the most widely used and the leading machine learning techniques in code smell detection. They were detected by 9, 7, 6 and 5 papers respectively. So, the answer to the first research question can be summarized as follows.

Research Question 2: Which machine learning techniques are most commonly used in the detection of code smells?
Hence, the result shows that Random Forest, Decision tree, Naïve Bayes and SVM are the most widely used machine learning techniques in code smell detection in the studies from 2017-2020.

Then the study further extends the analysis by merging the concepts of question 1 and 2 covered in the SLR. Hence tried to assess the higher results (in terms of the widely used performance measures) gained for the four most commonly studied smells in studies considering them. Accordingly, the study was able to come with the following conclusions.

- Random Forest is the best machine learning technique in the detection of Long method with a very promising result.
- Deep semantic based approach and MLP are the best learning technique in the detection of Feature Envy.
- The machine learning techniques Random Forest and GBT are the best in the detection of the class level code smell Data class.
- JRip and Deep semantic approach are the best in the detection of the class level code smell God class.

Finally, the third and final research question to be addressed by this SLR part of the study is “What datasets have been used for code smell detection?” The purpose of this topic was to learn more about the different types of datasets that are utilized in code smell detection using machine learning techniques. The major reason for this question was that the selection of dataset has a great influence in the performance of a given ML technique. The study has categorized the datasets in to three categories. These categories are Type of dataset used (name of the datasets), the nature of dataset being used (open source (publicly available) or other projects datasets (Industrial) and Nature of dataset (cross project or within project dataset). As a result, the SLR shows that most studies have used an open-source projects to prepare their datasets. Thirteen out of fifteen papers excluding S8 (whose dataset is not explicitly mentioned) adopt an open-source projects. In addition, the result was able to point out that the Qualitas corpus dataset is identified as the most widely used smells dataset used by five papers. Additionally, all the fifteen papers that have explicitly mentioned their datasets used a cross-project platform to build their dataset.

Research Question 3: What datasets have been used for code smell detection?
The datasets Qualities corpus containing java projects, Xerces (with different versions) and other project datasets that are not explicitly mentioned are the most commonly used open-source datasets for training the learning algorithms in code smell detection. Additionally, majority of the studies have used an open-source dataset and almost all of them have used different projects (cross- projects) to build their dataset.

6.2 Findings of the Experimental work

All the previous works mentioned in the literature review part of this study consider the application of machine learning techniques for the detection of a single smell type. None of them have tried to adopt more than one smell type in a single dataset. Except for the study that is undergone by [66]. This study tries to address the issue by introducing the existence of more than one type of code smell for a single instance. They have considered two most frequently studied method level smells. Similarly, this study adopts those two method level datasets used by the reference work. That is the reason why this work is used as a reference work for conducting a study in the same topic.

6.2.1 Results of the reference Work

The reference work as described above, tried to address a new concept which is the adoption a multi label dataset containing the existence of more than one smell type. This study has applied five machine learning techniques on the MLD. The ML techniques used in the reference work and their performance result is summarized as follows: -

Table 6. 1 Result of the Reference work

No	ML techniques	Performance in Accuracy
1	J48 pruned	96.7 %
2	Random Forest	96.7 %
3	B-J48 pruned	97.5 %
4	B-J48 Unpruned	97.2 %
5	B- Random Forest	96.7 %

The researchers have applied tree-based classifiers and were able to get a promising result in which most of them have performed above 95 %. The greatest result (97.5 %) was gained by the B-J48 pruned algorithm. Implying that tree-based classifiers are suitable for the MLD in the same way they work for the single labeled dataset containing single smell type.

6.2.2 Results of the original Work

Similar to the reference work, this study has tried to create a replicable experiment recreating the most commonly applied machine learning techniques applied for the detection of the two selected code smells. These techniques were applied to the multi-class dataset containing the smells that are addressed by the reference work. The techniques applied on the MCD were J48, Random Forest, JRip and Naïve Bayes. These techniques are the most widely used techniques in the detection of code smells according to the SLR result. Additionally, they are selected for a reason that the study believes those techniques are well suited for multi class dataset. The performance of the ML techniques used in the original study is presented as follows:-

Table 6. 2 Results of the Original work

No	**Classification Algorithm**	**Accuracy**
1	J48	96.3061 %
2	Random Forest	97.4934 %
3	JRip	93.6675 %
4	Naive Bayes	90.5.13 %

As a result, most of the applied techniques (common one) performed nearly the same result as the experiments on the reference work. They have scored above 95% of Accuracy. The greatest result in this study, (97.4934 %) was gained by the RF algorithm. This study then further enhances the performance of the best performing ML technique by adjusting the parameter "seed size" to different values. As a result, was able to get a result (97.8892 %) that is even higher than that gained with the default parameter of the classifier.

6.2.3 Comparison of Results of the Reference and Original Work

The table below shows the experimental comparison between the reference work and this study. But, to maintain the consistency, the study considers algorithms that are used in both studies which are J48 and Random Forest. Additionally, the reference work has used the metrics Accuracy, Hamming Loss and Exact Match Ratio to measure the performance of algorithms. On the other side, this study applies other metrics along with the one common metric "Accuracy". So, the comparison is made using Accuracy. The table presents the performance of the considered machine learning techniques in the reference and original work in terms of Accuracy which is common for both studies.

Table 6. 3 Accuracy comparison of common metrics used in both studies

No	Studies	ML techniques	Performance in terms of Accuracy
1	Original Work	J48	96.3061 %
		Random Forest	97.4934 %
		Random Forest (with parameter modification)	**97.8892 %**
2	Reference Work	J48	96.7 %
		Random Forest	96.7 %

The table shows that with the common ML techniques used, the RF algorithm was able to get a higher result in this specific dataset (MCD). But, the J48 performs better in the reference work (MLD). In terms of the best results of the two studies, the researcher in the reference work was able to get a best result with the B-J48 pruned and B-J48 unpruned algorithm. The performance of these algorithm in terms of accuracy were 97.5 % and 97.2 % respectively. But this work was able to get the best result with the RF algorithm. Accuracy rate of this algorithm was 97.8892% which is better than the best result gained by B-J48 pruned (97.5) of the reference work.

From the experimental result, it can be concluded that the tree based and rule-based classifiers are suitable for this specific dataset and performed a good classification of the instances in the same way they did in the reference work. In addition, Random Forest is best performing algorithm for the Multi Class Dataset compared to the other ML techniques applied.

The fourth research question of this study is, What result achieved after developing and evaluating the performance of the proposed machine learning model that would be used for detection of code smells?. Hence, the answer is presented as follows.

Research Question 4: What result achieved after developing and evaluating the performance of the proposed machine learning model that would be used for detection of code smells?
The study used a Multi Class Dataset which is believed to be more representative of the real-world scenario. Hence, according to the experiments undertaken, most of the ML algorithms applied perform a promising result. Especially, the machine learning algorithm random forest was able to give higher result with respect to most of the performance measurement metrics used. Compared to the reference work, the common algorithms show approximately the same results. But this study shows a higher result with random forest (97.8892%) compared to the best result gained in the reference work with B-J48 pruned (97.5%).

Even if Naïve Bayes (NB) works well for high dimensional data [111] , it performs less in this study compared to the other ML techniques. According to [111] in order for NB to perform well it requires a large number of records and the dataset used by this study is very small in size. Additionally, according to [112], NB algorithm performs better for categorical type of data than the numerical ones. The dataset used in this study is composed of numerical data. So, this nature of the dataset influences the performance result of the NB algorithm.

Random forest performs best in this is study. This is because according to [79] , random forest algorithm works well for high dimensional and small datasets.

6.3 Threats to conclusion validity

There are some concepts that might influence the generalizability of the findings in both the SLR and Experimental work.

The papers in the SLR use different projects to assess their results and even the studies that use the same project use different annotations to train the data, decreasing the reliability of the comparisons of performance. Additionally, with respect to the generalizability of the findings, this study used two code smell datasets which are constructed from 74 open-source Java projects for experimentation. However, it’s hard to conclude that the results can be generalized to other coding languages and industrial practice. So, future replications of this study are necessary to confirm the generalizability of the findings.

6.4 Contribution of the original work

Different researches were undergone in the detection of code smells using machine learning algorithms. Even if all those works have tried to fill gaps in this area, this study was able to notice spaces of improvement. The basic gaps addressed by this study are:

- Giving insight on the application of machine learning techniques in the detection of code smells by performing a through SLR. With this, future researchers will have an initial understanding on the code smells detected and machine learning techniques deployed to detect them in recent years.
- The representation of more than one smell in a single dataset by introducing a Multi class dataset containing two code smells which is a more realistic than the binary datasets adopted by previous works.
- A balanced proportion of class labels which is one of the problems in most of the smell’s dataset due to the fact that they are composed of an unrealistically unbalanced proportion between the smelly and non-smelly instances.
- An impure dataset containing an irrelevant feature, which do not have importance rather than increasing the complexity. Those attributes were taken into consideration by the previous researchers even when they know their significance is low.
- The performance of algorithms in a multi label dataset will suffer from the specific problem transformation method adopted as indicated by the reference work. In order to alleviate this problem, the study has followed the idea of merging the dataset in a better way. This way the study believes label chaining (multi class problem transformation) considers the label correlation effectively than the binary relevance method.

So, this study has tried to fill the gaps of all the literatures reviewed. Hence, tried to come up with a more realistic model to rely on.

CHAPTER SEVEN
CONCLUSION AND RECOMMENDATION

7.1 Conclusion

In this research, an attempt has been made to perform SLR and apply ML techniques for the detection of method level code smells. This study has tried to systematically review studies undergone on the same topic to identify the most commonly studied smells and the most widely used ML techniques in the detection of code smells. It also tried to create a replicable experiment recreating the most widely used and best performing machine learning techniques applied for code smells identification on two selected smells. This study's key contribution can be summarized as follows.

- This study has undergone the Systematic Literature Review in more recent papers from 2017 to recent year 2020. Hence, was able to detect which smells and ML techniques have been given attention recently. Additionally, it was able to point out the nature of the datasets being used by these studies.
- This paper has adopted two datasets from other researcher and merged those datasets in order to form a more realistic dataset that represent real use case scenario. Implying that a given instance can be affected by more than one smell type.
- The study has tried to discriminate the attributes according to their information gain value. Hence, the classification task is conducted according to the contribution of the attributes in detecting the final class.
- There is a data imbalance problem in most smell datasets. Similarly, the dataset used by this study suffers from an imbalanced proportion among the classes. To alleviate this data imbalance problem, different scholars have proposed a set of balancing techniques (over and under sampling). This study uses one of the most commonly applied balancing technique which is SMOTE. Finally, the experimental work then was conducted on a more balanced environment.

In general, the results from this study are very promising. Most of the applied classifiers have performed best with respect to most of the performance metrics applied. Especially the tree and rule-based classifiers show a higher result the same way they performed in the reference work.

The study has also shown that it is possible to identify more than one smell and the algorithms used for a multi class smell datasets have relatively the same performance with the ones with single class dataset.

7.2 Recommendation

7.2.1 Recommendation for Practical use of ML techniques in smells detection

Application of Machine Learning techniques in the area of code smell detection is a very recent trend according to different literatures. Hence, many researchers now a days are focusing on addressing the open issues in this area. As a result, a continuous work is being done and a promising result is being achieved. But this detection mechanism should be applied for real use case scenario. Software developers should give attention on addressing code smells and apply this new trend in the examination of the quality of their code in the ongoing process of development. So that they can decide on the appropriate refactoring method to be used if a smell exists. By doing so, they can guarantee a software product that is of high quality and open/easy to extend/improve. This way they can reduce poor quality software systems.

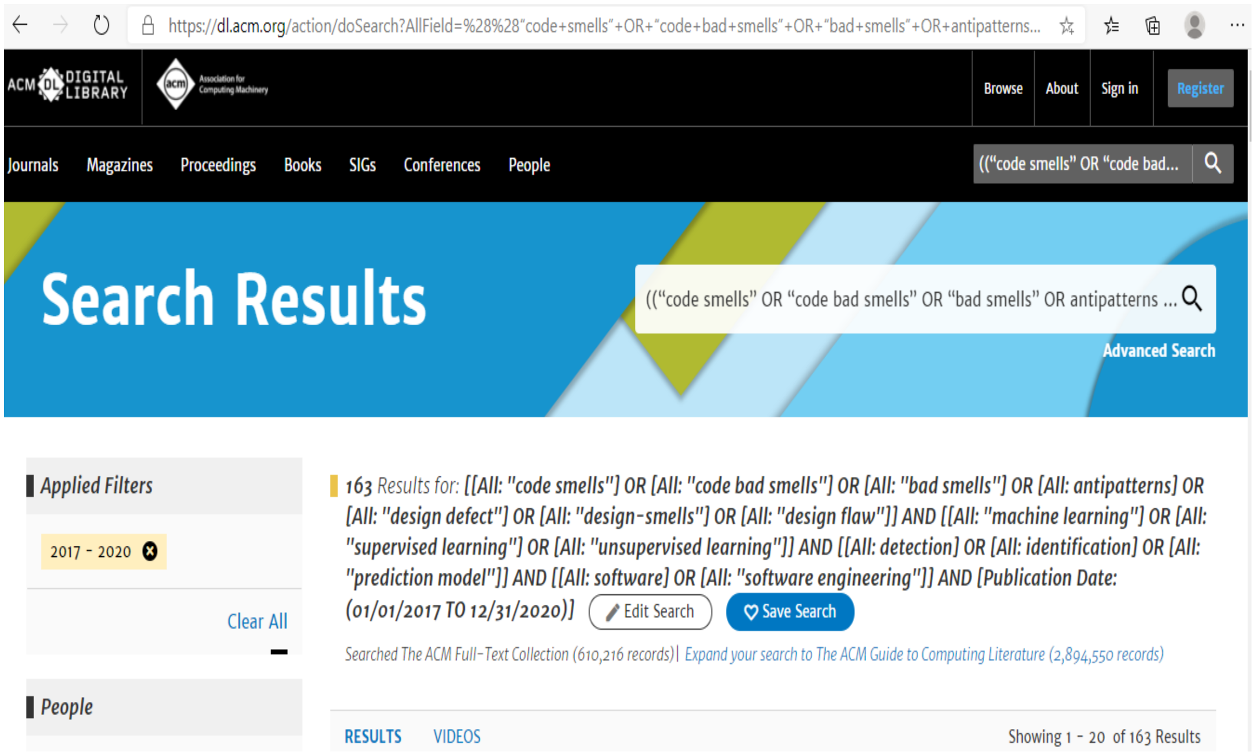
7.2.2 Recommendation for Future Work

The proposed approach tried to cover some gaps in the previous studies by introducing the detection of two code smells Long Method and Feature Envy. But, still there is a plenty of room for improvement. Other researchers can improve this work in the following ways.

- ➢ The dataset used by the study is quite small in size. Hence, the representativeness of the dataset is low. The experimental results and conclusions can benefit from a considerably large amount of data. So, other researchers can use a dataset that is richer in size.
- ➢ Additionally, this study has introduced the detection of two code smells in a single instance. But, an instance of a similar method may be affected by code smells other than those specified by the study. So, future studies may consider the existence of multiple smell types in a single instance by introducing other techniques to handle them.
- ➢ This study has applied four ML techniques. But, according to the works of previous studies there are a number of techniques that can be used in the detection of code smells. So, future studies should incorporate the application of different ML techniques and check if other techniques can handle the problem better.

- The other improvement area can be the application of ML techniques for the detection of class level code smells. This study has considered the detection of method level smells only. Other studies can replicate the study for the detection of class level smells.
- Future studies may also consider the application of hybrid approach like an integration of clustering and knowledge-based approach to explore the possibility of multiple code smells in a single Dataset.

APPENDIX I Total number of papers retrieved from ACM digital library

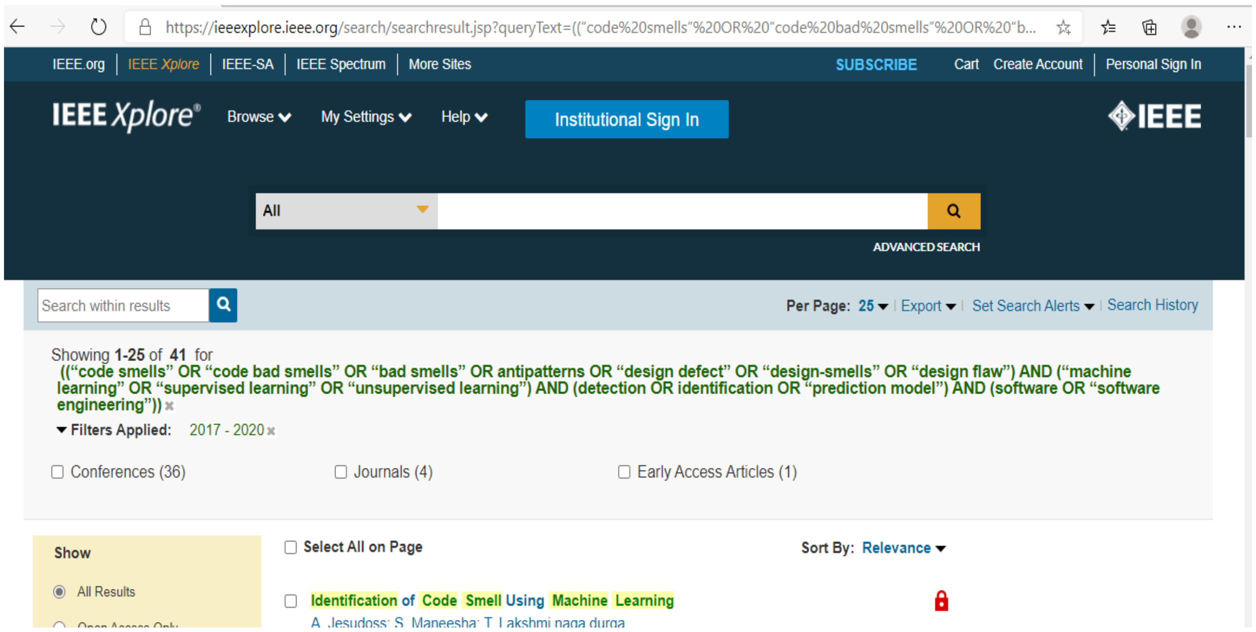


163 Results for: [[All: "code smells"] OR [All: "code bad smells"] OR [All: "bad smells"] OR [All: antipatterns] OR [All: "design defect"] OR [All: "design-smells"] OR [All: "design flaw"]] AND [[All: "machine learning"] OR [All: "supervised learning"] OR [All: "unsupervised learning"]] AND [[All: detection] OR [All: identification] OR [All: "prediction model"]] AND [[All: software] OR [All: "software engineering"]] AND [Publication Date: (01/01/2017 TO 12/31/2020)]

Searched The ACM Full-Text Collection (610,216 records) | Expand your search to The ACM Guide to Computing Literature (2,894,550 records)

Showing 1 - 20 of 163 Results

APPENDIX II Total number of papers retrieved from IEEE digital library



Showing 1-25 of 41 for
(("code smells" OR "code bad smells" OR "bad smells" OR antipatterns OR "design defect" OR "design-smells" OR "design flaw") AND ("machine learning" OR "supervised learning" OR "unsupervised learning") AND (detection OR identification OR "prediction model") AND (software OR "software engineering"))

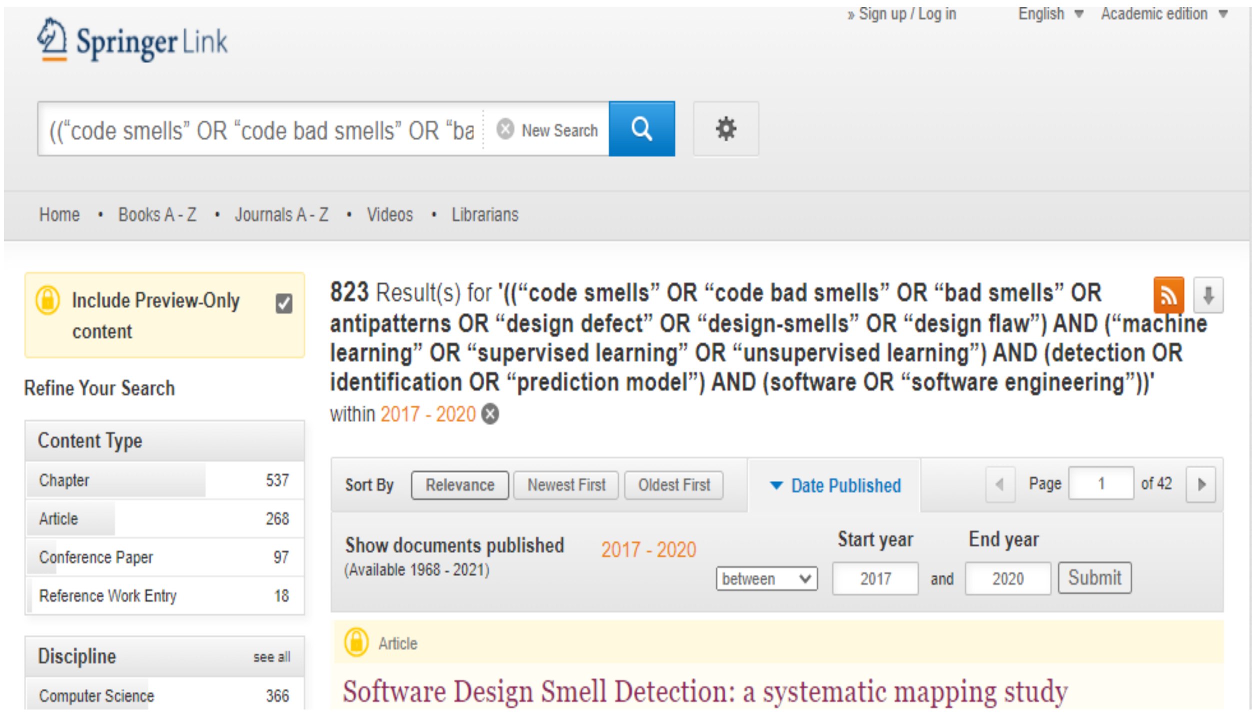
Filters Applied: 2017 - 2020

Conferences (36) Journals (4) Early Access Articles (1)

Identification of Code Smell Using Machine Learning

A Jesudoss; S Maneesha; T Lakshmi naga durga

APPENDIX III Total number of papers retrieved from Springer digital library


Springer Link
» Sign up / Log in
English
Academic edition
New Search
Home • Books A - Z • Journals A - Z • Videos • Librarians
Include Preview-Only content
Refine Your Search
Sort By Relevance Newest First Oldest First
Date Published
Page 1 of 42
Submit

((“code smells” OR “code bad smells” OR “ba

823 Result(s) for '((“code smells” OR “code bad smells” OR “bad smells” OR antipatterns OR “design defect” OR “design-smells” OR “design flaw”) AND (“machine learning” OR “supervised learning” OR “unsupervised learning”) AND (detection OR identification OR “prediction model”) AND (software OR “software engineering”))' within 2017 - 2020

Content Type	
Chapter	537
Article	268
Conference Paper	97
Reference Work Entry	18

Discipline	see all
Computer Science	366

Show documents published 2017 - 2020 (Available 1968 - 2021)

Start year between 2017 and End year 2020

Article

Software Design Smell Detection: a systematic mapping study

APPENDIX IV Questioner form to accomplish the Quality assessment task.

Dear Respondents please fill the form properly. Put **X** mark in the Yes, Partial or No alternatives under each question. Here are the contents of each questions read them properly and put the mark accordingly.

Question 1 Is the dataset being used by the researcher mentioned clearly?

Question 2 Is the machine learning classifier used clearly defined?

Question 3 Are the code smells detected by the proposed technique clearly defined?

No	Author of the paper	Question 1			Question 2			Question 3		
		Yes	**Partial**	**No**	**Yes**	**Partial**	**No**	**Yes**	**Partial**	**No**
1	Mhawish, M.Y. et al., 2020 [89]									
2	Guggulothu, T. et al., 2020 [66]									
3	Cruz, D. et al., 2020 [90]									
4	Pecorelli, F. et al., 2019 [91]									
5	Luiz, F.C., 2019 [49]									
6	Rubin, J. et al., 2019 [92]									
7	Oliveira, D. et al., 2020 [93]									
8	Liu, H. et al., 2019 [94]									
9	Azadi, U. et al., 2018 [95]									
10	Guo, X. et al., 2019 [96]									
11	Pecorelli, F. et al., 2019 [97]									
12	Kaur, A. et al., 2017 [98]									
13	Gupta, H. et al., 2019 [99]									
14	Karađuzović-Hadžiabdić, K. et al., 2018 [100]									
15	Das, A.K. et al., 2019 [101]									
16	Kiyak, E.O. et al., 2019 [102]									
17	Singh, R. et al., 2020 [103]									
18	Di Nucci, D. et al., 2018 [86]									
19	Jesoudoss, A. et al., 2019 [104]									
20	Yang, Y. et al., 2018 [105]									
21	Thongkum, P. et al., 2020 [106]									
22	Chen, D. et al., 2019 [107]									

APPENDIX V Design Metrics Definition and their Computation Details

1. **Lines of Codes (LOC)**:- An operation's or a class's total amount of lines of code, including all blank lines and comments. LOC of a method is calculated by counting the LOC from the method signature to the last curly bracket. The value for this metric becomes worse for greater value.
2. **Lines of Codes without Accessor or Mutator Methods (LOCNAMM)**:- A class's total amount of lines of code, including blank lines and comments, but excluding accessor and mutator methods and their related comments. A method's LOCNAMM is determined by counting the number of LOCs from the class declaration to the last curly bracket. For higher numbers, the metric's value deteriorates [66].
3. **Number of Packages (NOPK)**:- A system's total number of packages.
4. **Number of Classes (NOCS)**:- A system's, a package's, or a class's total number of classes.
5. **Number of Methods (NOM)**:- The number of methods declared locally in a class, including both public and private methods, is represented by NOM. Methods that have been overridden are ignored. The number of methods declared in class is added together to get the NOM of a method for a class. The NOM of a method for package method is calculated by adding the NOMs of all the classes in the package. The NOM of a project method is calculated by adding the NOMs of all the packages in the project [86] [22].
6. **Number of Not Accessor or Mutator Methods (NOMNAMM)**:- NOMNAMM is the number of methods declared locally in a class, including both public and private methods but excluding accessor and mutator methods. The number of non-accessor or mutator methods declared in a class is added together to get NOMNAMM. The NOMNAMM for a package is determined by adding the NOMNAMM for all of the classes in the package. The NOMNAMM for a project is computed by adding the NOMNAMM for all of the project's packages [66].
7. **Number of Attributes (NOA)**:- A class's number of attributes.
8. **Cyclomatic Complexity (CYCLO)**:- The greatest number of directly unrestricted pathways in a method is known as cyclomatic complexity. The path is linear when the relevant code's execution flow does not branch. This metric indicates how complex the code is, which has an impact on maintenance and modularization efforts. Because it's

difficult to understand, code with a lot of "break", "continue", "goto", or "return" clauses is tough to simplify and divide into simpler functions [66]. The precise Cyclomatic Complexity has been computed in the dataset used. The Cyclomatic Complexity adds 1 to the complexity for each occurrence of "logical conjunction" and "logical and in conditional expressions." i.e., the statement if (a && b || c) would have a Cyclomatic Complexity of one but the strict Cyclomatic Complexity of the expression will be three. The minimum Cyclomatic Complexity is one [66]. The value for this metric becomes worse for greater values.

9. **Weighted Methods Count (WMC)**:- The total complexity of the methods defined in the class is WMC. WMC is computed with the Cyclomatic Complexity metric (CYCLO) [22].
10. **Weighted Methods Count of Not Accessor or Mutator Methods (WMCNAMM)**:- WMCNAMM is the sum of the complexity of the class's methods, which are not accessor or mutator methods. The WMCNAMM, like the WMC, is calculated using the Cyclomatic Complexity metric (CYCLO).
11. **Average Methods Weight (AMW)**:- The average static complexity of a class's methods [86]. The Cyclomatic Complexity (CYCLO) measure is used to calculate AMW complexity. According to [66], AMW can be calculated with the formula:-

$$f(x) = \{ \text{ AMW} = \frac{\text{WMC}}{\text{NOM}} \text{ where } NOM \neq 0$$

12. **Average Methods Weight of Not Accessor or Mutator Methods (AMWNAMM)**:- The average static complexity of a class's non-accessor and mutator methods is referred to as AAN. The Cyclomatic Complexity measure is used to calculate AMWNAMM complexity (CYCLO). According to the researcher at [66], AMWNAMM can be calculated with the formula:-

$$f(x) = \{\text{AMWNAMM} = \frac{\text{WMCNAMM}}{\text{NOMNAMM}} \text{ where } NOMNAMM \neq 0$$

13. **Maximum Nesting Level (MAXNESTING)**:- The maximum number of control structures that can be nestled within a single operation [66]. For higher values, the metric's value deteriorates.

14. **Weight of Class (WOC)**:- The number of in-service public methods divided by the total number of public members. This metric measures the weight of functionalities offered by a class through its public interface.

$$\frac{\text{Number of Non Abstract Public Non Accessor or Mutator Methods}}{\text{Total Number of Public Methods and Attributes.}}$$

15. **Called Local Not Accessor or Mutator Methods (CLNAMM)**:- The total number of called non accessor or mutator methods in the same class as the measured method. It's calculated by adding up the number of Intra Methods that aren't accessors or mutators.
16. **Number of Parameters (NOP)**:- A method's number of arguments. The more parameters there are, the more difficult it is to grasp the method signature. As a result, the value of this metric becomes worse as the amount increases.
17. **Number of Accessed Variables (NOAV)**:- The total number of variables from the observed operation that were accessed directly or via accessor methods. Parameters, local variables, as well as instance variables and global variables stated in system classes, are among these variables. The Used Variables defined within the system, not in external libraries, are counted in the dataset. The list of Called Methods was used to count the variables accessible through accessor methods, and then the Used Intra Variables by each accessor method in the set of Called Methods were counted. For higher numbers, the metric's value worsens.
18. **Access to Local Data (ATLD)**:- The number of attributes declared by the current classes accessed directly or via accessor methods by the measured method. It's obtained by adding up the number of Used Intra Variables defined within the system rather than in external libraries. The list of Called Intra Methods was utilized to count the variables accessed using accessor methods, and then the Used Intra Variables by each accessor method in the set of Called Methods were counted. For higher values, the metric's value deteriorates.
19. **Number of Local Variable (NOLV)**:- Is the number of declared local variables in a method. A method's parameters are referred to as local variables [86].
20. **Tight Class Cohesion (TCC)**:- TCC is the normalized ratio of the total number of possible connections between methods to the number of methods directly associated with other methods through an instance variable. When two methods access the same instance

variable directly or indirectly through a method call, they have a direct link. TCC has a value between 0 and 1. According to the researcher at [66], given N, where N is the number of visible methods, NP is calculated by the formula

$$\mathrm{N}P = N * (N - 1)\ 2$$

The other Number of direct connections NDC, computed using a connectivity matrix that records all direct connected methods, making attention to cyclic calls among methods. Then TCC is calculated by the formula below.

$$\mathrm{TCC} = \frac{\mathrm{NDC}}{\mathrm{NP}} \text{ where NP} \neq 0$$

TCC only considers visible methods, which aren't private, don't implement an interface, or handle an event. Constructors aren't taken into account. Because of the divergent links with attributes, constructors are a difficulty. They boost cohesiveness, by creating diverging relationships between techniques that employ distinct properties, which isn't true [22].

21. **Lack of Cohesion in Methods (LCOM5)**:- LCOM5 is calculated by the following formula

$$\frac{\mathrm{NOM} - \frac{\sum \mathrm{m\epsilon M\ NOAcc(m)}}{NOA}}{NOM - 1}$$

Where M is the set of methods of the class, NOM the number of methods, NOA, the number of attributes and NOAcc(m) is the number of attributes of the class accessed by method m. The value for this metric becomes worse for lower values.

22. **FANOUT**: - Number of classes that have been called. It's calculated by adding up all of the system's Called Classes. For higher numbers, the metric's value deteriorates.
23. **Access to Foreign Data (ATFD)**:- The number of attributes accessible directly or by invoking accessor methods or methods from unrelated classes in the system. First, the Used Inter Variables belonging to the system, also through not Constructor, Public, and not Abstract Called Inter Methods of the system were summed up. For class, the Used Inter Variable belonging to the system, also through not Constructor, Public and not

Abstract Called Inter Methods from the field declaration class of the methods and from all the not Constructor and not Abstract Methods Declared in Class were summed up [22]. For higher numbers, the metric's value deteriorates.

24. **Foreign Data Providers (FDP)**:- The number of classes that specify the attributes that are accessible in accordance with the ATFD metric. It's calculated by adding up all the classes that have foreign data declared, and only counting each class once. [66]. For higher numbers, the metric's value deteriorates.
25. **Response for a Class (RFC)**:- The size of a class's response set is measured using RFC. "All methods that can be invoked in response to a message to an object of the class" are included in a class's response set. It comprises both native and inherited methods, as well as methods from other classes. This statistic shows the complexity of the class as well as the amount of communication it has with other classes. The complexity of a class increases as the number of methods that can be triggered from it via messages grows [22]. It is computed by adding the Inherited Methods by the Called Inter Methods and Called Hierarchy Methods, counting each method only once. Only call to classes belonging to the system [66]. For higher numbers, the metric's value deteriorates.
26. **Coupling Between Objects classes (CBO)**:- Two classes are tied if one of them calls or accesses a method or an attribute of the other, i.e., one class calls or accesses another class's method or attribute. The use of inheritance and polymorphically called methods in couplings is considered. The number of classes to which a class is related is referred to as its CBO. It's calculated by adding all of the system's unrelated classes that define the Used Inter Variables, Used Hierarchy Variables, Used Inter Types, Used Hierarchy Types, Called Inter Methods, and Called Hierarchy Methods by the measured class and its Ancestor Classes, as well as the methods the measured class methods declare and inherit. For higher numbers, the metric's value degrades.
27. **Called Foreign Not Accessor or Mutator Methods (CFNAMM)**:- is the number of called not accessor or mutator methods declared in unrelated classes respect to the one that declares the measured method. Only calls to classes belonging to the system were considered when calculating the number of called not accessor or mutator methods declared in unrelated classes in relation to the measured one. The number of non-accessor

or mutator Called Inter Methods and Called Hierarchy Methods in the system is added up. The call to class's default constructor is not counted [66].

28. **Coupling Intensity (CINT)**:- The number of unique operations called by the measured operation is denoted by CINT. Called Inter Methods corresponding to system classes are added together to calculate it [66]. For higher numbers, the metric's value continues to deteriorate.
29. **Coupling Dispersion (CDISP):-** The number of classes in which the operations called from the measured operation are defined, divided by CINT. For higher numbers, the metric's value worsens.

$$\text{CDISP} = \frac{\text{FANOUT}}{\text{CINT}} \text{ where CINT} \neq 0$$

30. **Maximum Message Chain Length (MaMCL)**:- is the maximum number of chained calls for a method. The value for this metric becomes worse for greater values.
31. **Number of Message Chain Statements (NMCS)**:- is a method's number of independent chained calls. For higher numbers, the metric's value degrades.
32. **Mean Message Chain Length (MeMCL)**:- is the average length of a method's linked calls. In Message Chains Info, the rounded average length of a chain has been calculated. If NMCS is zero, then MeMCL is zero too. The value for this metric becomes worse for greater values.
33. **Changing Classes (CC)**:- is the number of classes that define the methods that call the measured method. The total number of Calling Classes was calculated. For higher numbers, the metric's value declines.
34. **Changing Methods (CM)**:- The total number of methods that call the measured method. The number of Calling Methods were summed up. For higher numbers, the metric's value deteriorates.
35. **Number of Accessor Methods (NOAM)**:- A class's number of accessor (getter and setter) methods. The total number of getter and setter methods specified in Class was added together.
36. **Number of Public Attributes (NOPA)**:- A class's total amount of public attributes. For higher numbers, the metric's value deteriorates.
37. **Locality of Attribute Accesses (LAA)**:- The number of attributes from the method's definition class, divided by the total number of variables accessed (including attributes

used via accessor methods), where the number of local attributes accessed is computed in accordance with the ATLD standards. Variables specified in system classes were the only ones taken into account. Each attribute is only counted once, regardless of how the class accesses it (directly or through an accessor and/or mutator) or how many times it is accessed. For lower values, the metric's value deteriorates.

38. **Depth of Inheritance Tree (DIT)**:- The depth of a class, measured by DIT, within the inheritance hierarchy is the maximum distance between the class node and the tree's root, as determined by the number of ancestor classes. For classes without ancestors, DIT has a minimum value of one. The deeper in the hierarchy a class is, the more methods it will inherit, making it more difficult to predict its behavior [11]. Only the system's hierarchy classes were taken into account. In fact, the Ancestor Classes were visited in order from the bottom up, with the counter stopping at the first class that did not belong to the system [11]. The value for this metric becomes worse for greater values.
39. **Number of Interfaces (NOI)**:- The number of declared interfaces in a package or system. The Interface Declared Classes have been added up.
40. **Number of Children (NOC)**:- The number of children in a class hierarchy is the number of immediate subclasses that are subordinated to that class. The Children Classes were summed up. For higher numbers, the metric's value degrades.
41. **Number of Methods Overridden (NMO)**:- The number of overridden methods is represented by NMO, i.e., the class declares the superclass, and the superclass redefines the class. This criterion applies to methods that call their parent method several times. For classes that don't have a superclass, NMO isn't specified. The Overridden Methods have been enumerated. For lower values, the metric's value deteriorates.
42. **Number of Inherited Methods (NIM)**:- NIM is an easy metric that indicates how much behavior a certain class can reuse. It keeps track of how many methods a class can call from its super classes. For classes that don't have a superclass, NIM isn't defined. For higher numbers, the metric's value worsens.
43. **Number of Implemented Interfaces (NOII)**:- The number of interfaces that a class has implemented. It's calculated by adding all of the implemented interfaces together.

APPENDIX VI Additional Metrics used in the dataset

Here is the definition of some major keywords that is necessary prior to the definition of the listed additional metrics.

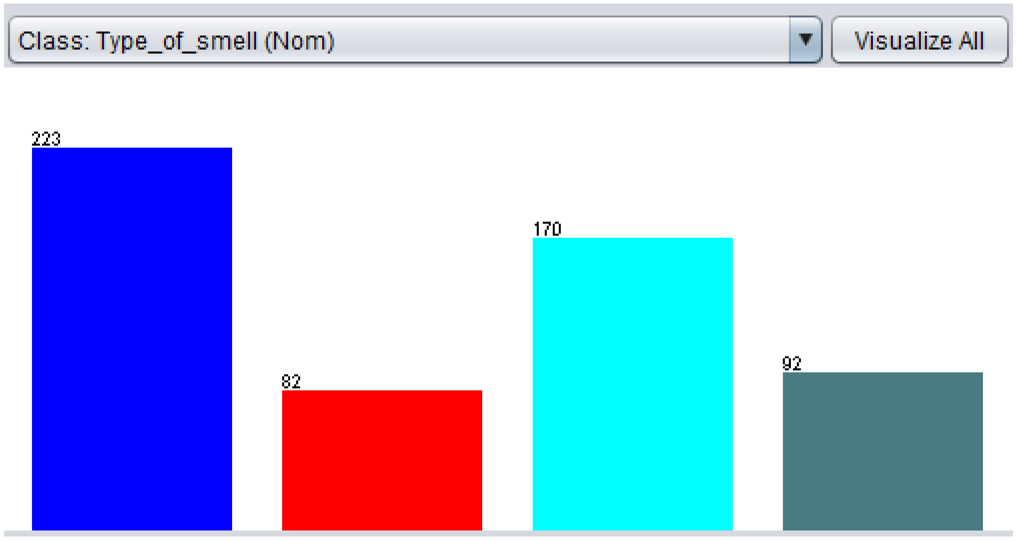
- **An Abstract method** is one that has no body (no implementation). A method must always be defined in an abstract class, or, to put it another way, if a class has an abstract method, the class must also be declared abstract.
- **Final method** is a method that cannot be overridden by subclasses.
- **Static method** is also called static function. It is a member of an object that may be accessed directly from the constructor of an API (Application Program Interface) object rather than from an object instance produced via the constructor. Any static member of a class can be accessed without requiring a reference to any object in the class.
- **Protected** indicates that a data member and method are only accessible by the classes of the same package and the subclasses present in any package.
- **Private** are those methods and members which can't be accessed in other class except the class in which they are declared. Only the class in which they are declared can perform this functionality. That is, the member or method is only visible within the class, not from any other class (including subclasses). Private members are visible to nested classes as well.
- **Default Constructor** In the absence of any programmer-defined constructors, the word default constructor refers to a constructor that is automatically created by the compiler and is usually a null constructor. A default constructor is one that either has no parameters or contains default values for all of the parameters if it does have parameters. The compiler offers an implicit generic parameterless constructor if no user-defined constructor for class A exists and one is required. If a class's constructors are all non-default, the compiler will not define a default constructor implicitly, resulting in a class with no default constructor.

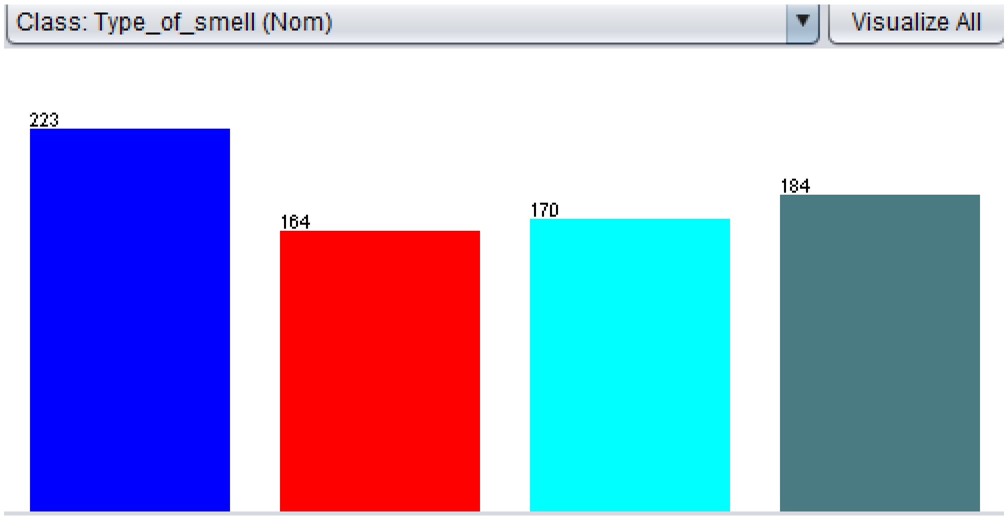
1. **Not Default Constructor** These are user defined constructors.
2. **number_not_abstract_not_final_methods:-** is the number of methods that are neither abstract nor final. In other words it refers to the number of methods that can implement some function and at the same time they can be overridden.
3. **number_private_visibility_attributes:** is the number of attributes that have a private visibility and can not be accessed in other classes.

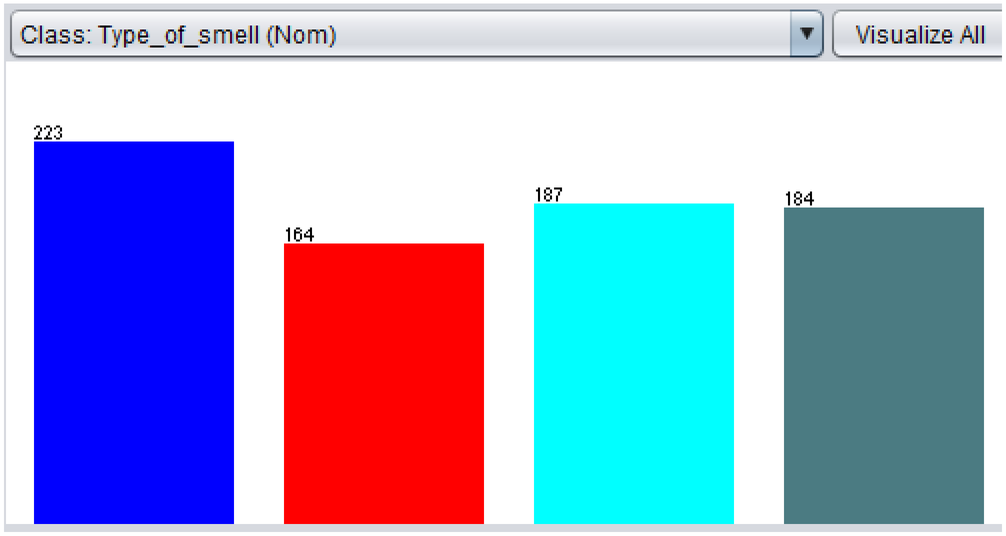
4. **number_not_final_not_static_methods:-** is the total number of methods that are neither final nor static one. These are methods that can be overridden and can be accessed a reference to any object in the class.
5. **num_final_attributes:-** is the total number of attributes that can be overridden.
6. **number_private_visibility_methods:-** is the total number of methods that have a private visibility and can not be accessed in other classes.
7. **num_static_attributes:-** is the total number of attributes that belong to the static class.
8. **num_final_static_attributes:-** is the total number of attributes that belong to final and static class.
9. **num_not_final_not_static_attributes:-** is the total number of attributes that are neither final class nor static.
10. **num_final_not_static_attributes:-** is the total number of attributes that are final but not static.
11. **number_protected_visibility_methods:-** refers to the total number of methods that are protected and can not be directly accessed by other classes.
12. **number_static_methods:-** is the total number of methods that can be directly accessed from the constructor of API object.
13. **number_package_visibility_methods:-** is the total number of methods that are visible to the package containing them.
14. **number_package_visibility_attributes:-** is the total number of attributes that are visible to the package they are contained in.
15. **is_static_methods:-** specifies whether a given method is directly accessible from the constructor of API.
16. **is_static_type:-** refers to the class being static. It specifies whether a given class is accessible directly from the constructor of API.
17. **number_not_final_static_methods:-** refers to the total number of methods that are not final (can be overridden) at the same time can be accessed directly through an object of API constructor.
18. **number_constructor_Defaultconstructor_methods:-** is the total number of methods that are default constructors and are not user defined one.

19. **number_final_static_methods:-** refers to the total number of methods that are final (can not be overridden) and can be accessed directly through an object of API constructor.
20. **number_final_not_static_methods:-** refers to the total number of methods that are final (can not be overridden) and those methods can not be accessed directly without the requirement of the instance of an object created through the API constructor.
21. **number_abstract_methods**:- Is the number of methods that are abstract (methods without any implementation).
22. **number_constructor_NotDefaultconstructor_methods:-** is the total number of methods that are not default constructors and are defined by the user.
23. **number_static_not_final_attributes:-** the total number of attributes that are static but not final.
24. **number_public_visibility_methods:-** refers to the total number of methods whose visibility is public and can be accessed publicly.
25. **number_standard_design_methods:-** is the total number of methods intended for duplication or repetitive manufacture.
26. **number_final_methods:-** is the total number of methods that can not be overridden by their subclass.
27. **number_protected_visibility_attributes:-** refers to the total number of attributes that are protected and their access is restricted to the class they belong to.

APPENDIX VII SMOTE stages







References

[1] M. Lehman, "Programs, life cycles, and laws of software evolution," in *Proceedings of the IEEE*, 1980.

[2] D. Parnas, "Software aging," in *Proceedings of 16th International Conference on Software Engineering*, 1994.

[3] Tamburri, D.A., Palomba, F., Serebrenik, A. and Zaidman, A., "Discovering community patterns in open-source: a systematic approach and its evaluation," *Empirical Software Engineering,* vol. 3, pp. 1369-1417, 2019.

[4] Avgeriou, P., Kruchten, P., Ozkaya, I. and Seaman, C., "Managing technical debt in software engineering," vol. 6(4), 2016.

[5] W. Cunningham, "The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger," 1992, pp. 29-30.

[6] Kruchten, P., Nord, R.L. and Ozkaya, I., "Technical debt: From metaphor to theory and practice," *Ieee software,* vol. 29(6), pp. 18-21, 2012.

[7] Seaman, C. and Guo, Y., "Measuring and monitoring technical debt. In Advances in Computers," *Elsevier,* vol. 82, pp. 25-46, 2011.

[8] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D., "Refactoring: Improving the Design of Existing Code Addison-Wesley Professional," Berkeley,, CA, USA., 1999.

[9] Güzel, A. and Aktas, Ö., "A survey on bad smells in codes and usage of algorithm analysis," *International Journal of Computer Science and Software Engineering,* vol. 5(6), p. 114, 2016.

[10] Danphitsanuphan, P. and Suwantada, "Code smell detecting tool and code smell-structure bug relationship," in *2012 Spring Congress on Engineering and Technology*, 2012.

[11] Fontana, F.A., Braione, P. and Zanoni, M, "Automatic detection of bad smells in code: An experimental assessment," vol. 11(2), pp. 1-5, 2012.

[12] Maneerat, N. and Muenchaisri, P., "Bad-smell prediction from software design model using machine learning techniques," in *Eighth international joint conference on computer science and software engineering*, 2011.

[13] Khomh, F., Vaucher, S., Guéhéneuc, Y.G. and Sahraoui, H., "A bayesian approach for the detection of code and design smells," in *Ninth International Conference on Quality Software*, 2009.

[14] Banker, R.D., Datar, S.M., Kemerer, C.F. and Zweig, D., vol. 36(11), pp. 81-95, 1993.

[15] Marticorena, R., López, C. and Crespo, Y., "Extending a taxonomy of bad code smells with metrics," in *International Workshop on Object-Oriented Reengineering*, 2006.

[16] T. a. T. T. Mens, "A survey of software refactoring.," *IEEE Transactions on software engineering,* vol. 30(2), pp. 126-139, 2004.

[17] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A. and Poshyvanyk, D, "When and why your code starts to smell bad," in *International Conference on Software Engineering*, 2015.

[18] Kessentini, W., Kessentini, M., Sahraoui, H., Bechikh, S. and Ouni, A., "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering,* vol. 40(9), pp. 841-861, 2014.

[19] Olbrich, S., Cruzes, D.S., Basili, V. and Zazworka, N., "The evolution and impact of code smells: A case study of two open source systems. What are code smells?," *In proceedings of the 2009 3rd international symposium on empirical software engineering and measurement,* pp. 390-400, 2009.

[20] E. P. C. a. B. A. P. Murphy-hill, "How We Refactor, and How We Know It.," *IEEE Transactions on Software Engineering,* vol. 38(1), p. 55–57, 2012.

[21] Bryton, S. and e Abreu, F.B., "Strengthening refactoring: Towards software evolution with quantitative and experimental grounds," in *In 2009 Fourth International Conference on Software Engineering Advances*, 2009.

[22] Fontana, F.A., Mäntylä, M.V., Zanoni, M. and Marino, A., "Comparing and experimenting machine learning techniques for code smell detection," in *Empirical Software Engineering*, 2016, pp. 1143-1191.

[23] Fokaefs, M., Tsantalis, N. and Chatzigeorgiou, A., "Identification and removal of feature envy bad smells," in *International conference on software maintenance*, 2007.

[24] M. V. J. a. L. C. Mantyla, "Bad smells-humans as code critics," in *International Conference on Software Maintenance,* 2004.

[25] Rasool, G. and Arshad, Z., "A review of code smell mining techniques," *Journal of software: Evolution and process,* vol. 27(11), pp. 867-895, 2015.

[26] Bryton, S., e Abreu, F.B. and Monteiro, M., "Reducing subjectivity in code smells detection: Experimenting with the long method.," in *International Conference on the Quality of Information and Communications Technology*, 2010.

[27] S. H. R. M. H. H. B. S. a. D. M. Counsell, "Is a strategy for code smell assessment long overdue?," in *In Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics,* Cape Town, South Africa., 2010.

[28] R. Marinescu, "Metrics-based rules for detecting design flaws," in *International Conference on Software Maintenance*, chigao, Illinoiss, USA, 2004.

[29] Moha, N., Guéhéneuc, Y.G., Duchien, L. and Le Meur, A.F., "DECOR: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering,* vol. 36(1), pp. 20-36, 2010.

[30] V. M. A. a. F. F. A. Ferme, "Is it a Real Code Smell to be Removed or not?," in *International Workshop on Refactoring & Testing (RefTest),* Wien, Austria., 2013.

[31] Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D. and De Lucia, "Recommending move method refactorings via relational topic models," vol. 40(7), pp. 671-694, 2013.

[32] Morales, R., Soh, Z., Khomh, F., Antoniol, G. and Chicano, F., "On the use of developers' context for automatic refactoring of software anti-patterns," *Journal of systems and software,* vol. 128, pp. 235-251, 2017.

[33] Panichella, A., Oliveto, R. and De Lucia, A., "Cross-project defect prediction models: L'union fait la force," in *Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, 2014.

[34] Zimmermann, T., Nagappan, N., Gall, H., Giger, E. and Murphy, B., "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009.

[35] Abbes, M., Khomh, F., Gueheneuc, Y.G. and Antoniol, G., "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension.," in *European conference on software maintenance and reengineering,* 2011.

[36] Khomh, F., Di Penta, M., Guéhéneuc, Y.G. and Antoniol, G., "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering,* vol. 17(3), pp. 243-275, 2012.

[37] Pascarella, L., Palomba, F. and Bacchelli, A., "Fine-grained just-in-time defect prediction," *Journal of Systems and Software,* pp. 22-36, 2019.

[38] Sjøberg, D.I., Yamashita, A., Anda, B.C., Mockus, A. and Dybå, T., "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering,* vol. 39(8), pp. 1144-1156, 2012.

[39] Fernandes, E., Oliveira, J., Vale, G., Paiva, T. and Figueiredo, E., "A review-based comparative study of bad smell detection tools," in *20th International Conference on Evaluation and Assessment in Software Engineering*, 2016.

[40] Zhang, M., Hall, T. and Baddoo, N., 2011, "Code bad smells: a review of current knowledge," in *Journal of Software Maintenance and Evolution: research and practice*, 2011.

[41] Fontana, F.A., Dietrich, J., Walter, B., Yamashita, A. and Zanoni, M, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *international conference on software analysis, evolution, and reengineering (SANER)*.

[42] Mäntylä, M.V. and Lassenius, C., "Subjective evaluation of software evolvability using code smells," *Empirical Software Engineering,* vol. 11(3), pp. 395-431, 2006.

[43] J. L. S. L. Z. H. Y. a. H. C. Wen, "Systematic literature review of machine learning based software development effort estimation models," *Information and Software Technology,* vol. 54(1), p. 41–59, 2012.

[44] Kotsiantis, S.B., Zaharakis, I. and Pintelas, P., "Supervised machine learning: A review of classification techniques," in *Emerging artificial intelligence applications in computer engineering*, 2007.

[45] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *International Conference on Software Maintenance*, 2004.

[46] M. Munro, "Product metrics for automatic identification of" bad smell" design problems in java source-code," in *International Software Metrics Symposium*, 2008.

[47] Hozano, M., Garcia, A., Fonseca, B. and Costa, E., "Are you smelling it? Investigating how similar developers detect code smells," *Information and Software Technology,* vol. 93, pp. 1130-146, 2018.

[48] Hadj-Kacem, M. and Bouassida, N., A Hybrid Approach To Detect Code Smells using Deep Learning, 2018, pp. 137-146.

[49] F. Luiz, "Identifying Code Smells with Machine Learning Techniques," 2018.

[50] Bennett, K. H. and Rajlich, V. T. V. T., "Software maintenance and evolution," in *a roadmap. In Proceedings of the Conference on the Future of Software Engineering*, 2000.

[51] Liu, H., Ma, Z., Shao, W. and Niu, Z., "Schedule of bad smell detection and resolution: A new way to save effort," *IEEE transactions on Software Engineering,* vol. 38(1), pp. 220-235, 2011.

[52] Abran, A. and Nguyenkim, H, "Measurement of the maintenance process from a demand-based perspective," *Journal of Software Maintenance: Research and Practice,* vol. 5(2), pp. 63-90, 1993.

[53] Seng, O., Stammel, J. and Burkhart, D., "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *8th annual conference on Genetic and evolutionary computation*, 2006.

[54] Van Emden, E. and Moonen, L., "Java quality assurance by detecting code smells," in *Ninth Working Conference on Reverse Engineering*, 2002.

[55] G. N. H. K. A. a. N. V. Saranya, "Model level code smell detection using egapso based on similarity measures," *Alexandria engineering journal,* vol. 57(3), pp. 1631-1642, 2018.

[56] Azeem, M.I., Palomba, F., Shi, L. and Wang, Q., "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," 2019.

[57] Mathur, N. and Reddy, Y.R., "Correctness of Semantic Code Smell Detection Tools.," in *In QuASoQ/WAWSE/CMCE@ APSEC*, 2015.

[58] Al-Shaaby, A., Aljamaan, H. and Alshayeb, M., "Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review.," *Arabian Journal for Science and Engineering,* vol. 45(4), pp. 2341-2369, 2020.

[59] Chatzigeorgiou, A. and Manakos, A., "Investigating the evolution of code smells in object-oriented systems," Innovations in Systems and Software Engineering, 2014.

[60] Roy, R., Stark, R., Tracht, K., Takata, S. and Mori, M., Continuous maintenance and the future–Foundations and technological challenges, 2016, pp. 667-688.

[61] Hamid, A., Ilyas, M., Hummayun, M. and Nawaz, A., "A comparative study on code smell detection tools.," *International Journal of Advanced Science and Technology,* vol. 60, pp. 25-32, 2013.

[62] Liu, X. and Zhang, C., "The detection of code smell on software development: a mapping study," in *5th International Conference on Machinery, Materials and Computing Technology (ICMMCT 2017),* Atlantis , 2017.

[63] B. a. A. T. Walter, "The relationship between design patterns and code smells," *Information and Software Technology,* vol. 74, pp. 127-142, 2016.

[64] Mantyla, M., Vanhanen, J. and Lassenius, C., "A taxonomy and an initial empirical study of bad smells in code.," in *International Conference on Software Maintenance,* 2003.

[65] M. Mantyla, "Bad smells in software-a taxonomy and an empirical study," Helsinki University of Technology, 2003.

[66] Guggulothu, T. and Moiz, S.A., "Code smell detection using multi-label classification approach," in *Software Quality Journal*, 2020.

[67] A. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of research and development*, vol. 3(3), pp. 210-229, 1959.

[68] I. a. M. M. El Naqa, "What is machine learning?," *In machine learning in radiation oncology Springer*, *Cham.*, pp. 3-11, 2015..

[69] D. Ruck, S. Rogers, M. Kabrisky, M. Oxley and Suter, "The multilayer perceptron as an approximation to a Bayes," vol. 1(4), p. 296–298, 1990.

[70] F. P. o. n. Rosenblatt, perceptrons and the theory of brain mechanisms, 1961.

[71] C. Cortes and V. Vapnik, Support-vector networks. Mach. Learn., vol. 20(3), 1995, p. 273–297 .

[72] H. Aljamaan and M. Elish, "An empirical study of bagging and boosting ensembles for identifying faulty classes in object-oriented software.," 2009.

[73] D. Broomhead and D. Lowe, Radial basis functions, multi-variable functional interpolation and adaptive networks, 1988.

[74] N. Friedman, D. Geiger and M. Goldszmidt, Bayesian networkclassifers. Mach. Learn., vol. 29(2–3), 1997, p. 131–163 .

[75] E. Castillo, J. Gutierrez and A. Hadi, Expert Systems and Probabiistic Network Models., Berlin: Springer, 1996.

[76] I. e. a. Rish, "An empirical study of the naive Bayes classifer In: IJCAI 2001 Workshop," *Empirical Methods In ArtifcialIntelligence*, vol. 3, p. 41–46, 2001.

[77] G. a. L. A. Seber, Linear Regression Analysis, vol. 329, John Wiley & Sons., 2012.

[78] S. Weisberg, Applied Linear Regression., Wiley, Hoboken, 2005.

[79] L. Breiman, Random forests. Mach. Learn., vol. 45(1), 2001, p. 5–32.

[80] Y. Yuan and M. Shaw, Induction of fuzzy decision trees. FuzzySets Syst., vol. 69(2), 1995, p. 125–139 .

[81] A. Jain, "Data clustering: 50 years beyond K-means. Pattern recognition letters," vol. 31(8), pp. 651-666, 2010.

[82] Kitchenham, B., Pretorius, R., Budgen, D., Brereton, O.P., Turner, M., Niazi, M. and Linkman, S., "Systematic literature reviews in software engineering–a tertiary study," *Information and software technology,* vol. 52(8), pp. 792-805, 2010.

[83] Kaur, A. and Singh, S., "Detecting software bad smells from software design patterns using machine learning algorithms," *International Journal of Applied Engineering Research,* vol. 13(11), pp. 10005-10010, 2018.

[84] Y. Bengio, Learning deep architectures for AI, Now Publishers Inc., 2009.

[85] Kitchenham, B. and Charters, S., Guidelines for performing systematic literature reviews in software engineering, 2007.

[86] Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A. and De Lucia, A., "Detecting code smells using machine learning techniques: are we there yet?," in *international conference on software analysis, evolution and reengineering (saner)*, 2018.

[87] W. a. V. V. Kuechler, "A framework for theory development in design science research: multiple perspectives.," *Journal of the Association for Information systems,* vol. 13, no. 6, p. 3, 2012.

[88] J. a. B. R. 2. Venable, "Eating our own cooking: Toward a more rigorous design science of research methods.," *Electronic Journal of Business Research Methods,* vol. 10, no. 2, pp. 141-153.

[89] Mhawish, M.Y. and Gupta, M., "Predicting Code Smells and Analysis of Predictions: Using Machine Learning Techniques and Software Metrics," *Journal of Computer Science and Technology,* vol. 35(6), pp. 1428-1445, 2020.

[90] D. S. A. a. F. E. Cruz, "Detecting bad smells with machine learning algorithms: an empirical study," in *3rd International Conference on Technical Debt*, 2020.

[91] F. D. N. D. D. R. C. a. D. L. A. Pecorelli, "On the role of data balancing for machine learning-based code smell detection," in *3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation*, 2019.

[92] J. H. A. M. N. B. M. a. B. N. Rubin, "Sniffing android code smells: an association rules mining-based approach," in *6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2019.

[93] D. A. W. S. L. O. W. G. A. a. F. B. Oliveira, "Applying Machine Learning to Customized Smell Detection: A Multi-Project Study," in *34th Brazilian Symposium on Software Engineering,* 2020.

[94] H. X. Z. a. Z. Y. Liu, "Deep learning based feature envy detection," in *33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[95] U. F. F. a. Z. M. Azadi, "Poster: machine learning based code smell detection through WekaNose," in *40th International Conference on Software Engineering*, 2018.

[96] X. S. C. a. J. H. Guo, "Deep semantic-Based Feature Envy Identification," in *11th Asia-Pacific Symposium on Internetware*, 2019.

[97] F. P. F. D. N. D. a. D. L. A. Pecorelli, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *27th International Conference on Program Comprehension (ICPC)*, 2019.

[98] A. J. S. a. G. S. Kaur, "A support vector machine based approach for code smell detection," in *International Conference on Machine Learning and Data Science (MLDS)*, 2017.

[99] H. K. L. a. N. L. Gupta, "An Empirical Framework for Code Smell Prediction using Extreme Learning Machine.," in *9th Annual Information Technology, Electromechanical Engineering and Microelectronics Conference (IEMECON)*, 2019.

[100] K. a. S. R. Karađuzović-Hadžiabdić, "Comparison of machine learning methods for code smell detection using reduced features," in *3rd International Conference on Computer Science and Engineering (UBMK)*, 2018.

[101] A. Y. S. a. D. S. Das, "Detecting Code Smells using Deep Learning," in *TENCON 2019-2019 IEEE Region 10 Conference (TENCON)*, 2019.

[102] E. B. D. a. B. K. Kiyak, "Comparison of Multi-Label Classification Algorithms for Code Smell Detection," in *3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, 2019.

[103] R. S. J. G. M. a. M. R. Singh, "Transfer Learning Code Vectorizer based Machine Learning Models for Software Defect Prediction," in *International Conference on Computational Performance Evaluation (ComPE)*, 2020.

[104] A. a. M. S. Jesudoss, "Identification of Code Smell Using Machine Learning," in *International Conference on Intelligent Computing and Control Systems (ICCS)*, 2019.

[105] Yang, Y., Ai, J. and Wang, F., "Defect prediction based on the characteristics of multilayer structure of software network," in *International Conference on Software Quality, Reliability and Security Companion (QRS-C),* 2018.

[106] Thongkum, P. and Mekruksavanich, S., "Design Flaws Prediction for Impact on Software Maintainability using Extreme Learning Machine," in *International Conference on Digital Arts, Media and Technology with ECTI Northern Section Conference on Electrical,*

Electronics, Computer and Telecommunications Engineering (ECTI DAMT & NCON), 2020.

[107] D. C. X. L. H. X. J. a. M. Y. Chen, "Deep learning based cross-project defect prediction," in *IEEE Access*, 2019.

[108] E. A. C. D. J. H. T. L. J. L. M. M. H. a. N. J. .. I. 2. Tempero, "The Qualitas Corpus: A curated collection of Java code for empirical studies," in *Software Engineering Conference*, Asia Pacific , 2010.

[109] R. Marinescu, " Measurement and quality in object-oriented design.," in *In Proceedings of the 21st IEEE international conference on software maintenance, . ICSM'05*, 2005.

[110] V. Ferme, "Jcodeodor: a software quality advisor through design flaws detection.Master's thesis University of Milano-Bicocca, Milano, Italy," 2013.

[111] S. D. a. H. P. C. Jadhav, ""Comparative study of K-NN, naive Bayes and decision tree classification techniques."," *International Journal of Science and Research (IJSR),* vol. 5, no. 1, pp. 1842-1845, 2016.

[112] "https://www.dataversity.net/," what-is-naive-bayes-classification-and-how-is-it-used-for-enterprise-analysis. [Online]. [Accessed 28 june 2022].

[113] Sjøberg, D.I., Yamashita, A., Anda, B.C., Mockus, A. and Dybå, T., "Quantifying the effect of code smells on maintenance effort," *Transactions on Software Engineering,* vol. 39{8}, pp. 1144-156, 2012.

[114] Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R. and De Lucia, A., "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation.," *Empirical Software Engineering,* vol. 23{3}, pp. 1188-1221, 2018.

[115] Palomba, F., Tamburri, D.A.A., Fontana, F.A., Oliveto, R., Zaidman, A. and Serebrenik, A, "Beyond technical aspects: How do community smells influence the intensity of code smells?," in *IEEE transactions on software engineering.*, 2018.

[116] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R. and De Lucia, A., "Do they really smell bad? a study on developers' perception of bad code smells.," in *International Conference on Software Maintenance and Evolution*, 2014.

[117] Santos, J.A.M., Rocha-Junior, J.B., Prates, L.C.L., do Nascimento, R.S., Freitas, M.F. and de Mendonça, M.G.,, "A systematic review on the code smell effect," *Journal of Systems and Software*, vol. 144, pp. 450-477, 2018.

[118] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D. and De Lucia, A, "Mining version histories for detecting code smells," *IEEE Trans. Softw.* , vol. 41{5}, p. 462489, 2014.

[119] Rapu, D., Ducasse, S., Gîrba, T. and Marinescu, R., "European Conference on Software Maintenance and Reengineering," in *European Conference on Software Maintenance and Reengineering*, 2004.

[120] Creswell, J.W. and Creswell, Research design: Qualitative, quantitative, and mixed methods approaches, Sage publications, 2017.

[121] Easterbrook, S., Singer, J., Storey, M.A. and Damian, D., "In Guide to advanced empirical software engineering," in *Selecting empirical methods for software engineering research*, London, Springer, 2008, pp. 285-311.

[122] Fabio, P., Gabriele, B., Di Penta, M., Rocco, O., Denys, P. and De Lucia, A, "Mining Version Histories for Detecting Code Smells," 2015.

[123] M. Fowler, "Refactoring: Improving the design of existing code," in *11th European Conference.*, Jyväskylä, Finland., 1997.

[124] Gasparic, M. and Janes, A., "What recommendation systems for software engineering recommend," *Journal of Systems and Software,* vol. 113, pp. 101-113, 2016.

[125] Ghotra, B., McIntosh, S. and Hassan, A.E., "Revisiting the impact of classification techniques on the performance of defect prediction models," in *37th IEEE International Conference on Software Engineering*, 2015.

[126] Y. Guo, "Measuring and monitoring technical debt," University of Maryland, Baltimore County, 2016.

[127] adj-Kacem, M. and Bouassida, N., A Hybrid Approach To Detect Code Smells using Deep Learning, In ENASE, 2018, pp. 137-146.

[128] Khan, S.U. and Azeem, M.I., "Intercultural challenges in offshore software development outsourcing relationships: an exploratory study using a systematic literature review," *IET software,* vol. 8{4}, pp. 161-173, 2014.

[129] C. Kothari, Research methodology: Methods and techniques, New Age International., 2004.

[130] M. Niazi, "Do systematic literature reviews outperform informal literature reviews in the software engineering domain? An initial case study," *Arabian Journal for Science and Engineering,* vol. 40{3}, pp. 845-855, 2015.

[131] Palomba, F., Di Nucci, D., Tufano, M., Bavota, G., Oliveto, R., Poshyvanyk, D. and De Lucia, A., "An open dataset of code smells with public evaluation," in *12th Working Conference on Mining Software Repositories*, 2015.

[132] Palomba, F., Panichella, A., De Lucia, A., Oliveto, R. and Zaidman, A., "A textual-based technique for smell detection," in *international conference on program comprehension (ICPC*, 2016.

[133] Palomba, F., Zaidman, A. and De Lucia, A., "Automatic test smell detection using information retrieval techniques," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2018.

[134] International Conference on Software Maintenance and Evolution (ICSME), "An exploratory study on the relationship between changes and refactoring," in *International Conference on Program Comprehension (ICPC)*, 2017.

[135] Peters, R. and Zaidman, A., "Evaluating the lifespan of code smells using software repository mining," in *European Conference on Software Maintenance and Reengineering*, 2012.

[136] Spadini, D., Palomba, F., Zaidman, A., Bruntink, M. and Bacchelli, A., "On the relation of test smells to software code quality," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2018.

[137] Taibi, D., Janes, A. and Lenarduzzi, V., "How developers perceive smells in source code," *Information and Software Technology,* vol. 92, pp. 223-235, 2017.

[138] Tantithamthavorn, C., McIntosh, S., Hassan, A.E. and Matsumoto, K., "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering,* vol. 43{1}, pp. 1-18, 2016.

[139] Tarhan, A. and Giray, G., "On the use of ontologies in software process assessment: a systematic literature review," in *International Conference on Evaluation and Assessment in Software Engineering*, 2017.

[140] Tsantalis, N. and Chatzigeorgiou, A, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering,* vol. 35{3}, pp. 347-367, 2009.

[141] Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A. and Poshyvanyk, D., "An empirical investigation into the nature of test smells," in *International Conference on Automated Software Engineering*, 2016.

[142] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A. and Poshyvanyk, D., "When and why your code starts to smell bad (and whether the smells go away)," in *Transactions on Software Engineering*, 2017.

[143] Vilela, J., Castro, J., Martins, L.E.G. and Gorschek, T, "Integration between requirements engineering and safety analysis," *A systematic literature review. Journal of Systems and Software,* vol. 125, pp. 68-92, 2017.

[144] Yamashita, A. and Moonen, L., "Do code smells reflect important maintainability aspects?," in *28th IEEE international conference on software maintenance (ICSM)*, 2012.

[145] Zazworka, N., Shaw, M.A., Shull, F. and Seaman, C., "Investigating the impact of design debt on software quality," in *In Proceedings of the 2nd Workshop on Managing Technical Debt*, 2011.

[146] Palomba, F., Zaidman, A., Oliveto, R. and De Lucia, A., "An exploratory study on the relationship between changes and refactoring," in *25th International Conference on Program Comprehension (ICPC)*, 2017.

[147] Mäntylä, M.V. and Lassenius, C.,, "Subjective evaluation of software evolvability using code smells: An empirical study.," in *Empirical Software Engineering*.

[148] S. Tarwani and A. Chug, "Predicting maintainability of open sourcesoftware using Gene Expression Programming and bad smells. In: 2016 5th International Conference on Reliability, InfocomTechnologies and Optimization (Trends and Future Directions) (ICRITO)," 2016.