

**SOFTWARE BAD SMELLS PREDICTION USING
ADVANCED MACHINE LEARNING
TECHNIQUES**

BY

AHMED ALI HAMDAN AL-SHAABY

**A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES**

**KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA**

**In Partial Fulfillment of the
Requirements for the Degree of**

**MASTER OF SCIENCE
In
SOFTWARE ENGINEERING**

DECEMBER, 2019

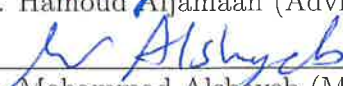
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

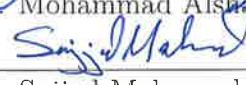
DEANSHIP OF GRADUATE STUDIES

This thesis, written by **AHMED ALI HAMDAN AL-SHAABY** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN SOFTWARE ENGINEERING**.

Thesis Committee

 2/3/2020
Dr. Hamoud Aljamaan (Adviser)

 2/3/2020
Dr. Mohammad Alshayeb (Member)

 02/03/2020
Dr. Sajjad Mahmood (Member)


Dr. Hamoud Aljamaan
Department Chairman


Dr. Salam A. Zummo
Dean of Graduate Studies

9/3/2020
Date



©AHMED Ali Hamdan AL-SHAABY
2019

Dedicated To
KFUPM

ACKNOWLEDGMENTS

*In the name of Allah. First and foremost thanks to **Allah** who grant me patience, strength to complete this thesis. My deep Thanks to the last messenger*

***Mohammed**, who guided us to the right path.*

*My deep appreciation and heartfelt thanks go to my thesis advisor **Dr.Hamoud Aljamaan** for his guidance and patience through my thesis, his continuous encouragement, and support can never be forgotten. His helpful guidance made this thesis exciting and knowledgeable for me.*

*Deep thanks to my thesis committee members **Prof. Mohammad Alshayeb** and **Dr. Sajjad Mahmood** for their suggestions, cooperation, valuable comments and support.*

*I would like to thank my **father, mother**, whose prayers are always a great source of strength for me and my deep thanks to my brother **Abdullah** for his continuous support and encouragement and my **wife** and daughters: **Rawan, Raheeq**, and **Renad** who took all the pain to give me time to finish this work.*

I am also grateful to all faculty members for their encouragement and for imparting quality learning and knowledge with their valuable support and able guidance that has led my way through this point of undertaking my research work.

*I owe thanks to my colleagues and my friends for their help, motivation and pivotal support. Special thanks to my brother **Sadam Al-Azani**.*

*Finally, deep thanks to King Fahd University of Petroleum & Minerals (**KFUPM**) that granted me the opportunity for doing my MSC. degree.*

TABLE OF CONTENTS

ACKNOWLEDGMENTS	vi
LIST OF TABLES	xi
LIST OF FIGURES	xiii
LIST OF ABBREVIATIONS	xiv
ABSTRACT (ENGLISH)	xvii
ABSTRACT (ARABIC)	xix
CHAPTER 1 INTRODUCTION	1
1.1 Problem Statement and Motivation	2
1.2 Research Objectives	3
1.3 Research Methodology	5
1.4 Thesis Contributions	7
1.5 Thesis Outline	7
CHAPTER 2 BACKGROUND	9
2.1 Code Bad Smells	9
2.1.1 Code smell categories	10
2.2 Machine learning techniques	15
CHAPTER 3 SYSTEMATIC LITERATURE REVIEW	19
3.1 Research Methodology	22

3.1.1	Research questions	24
3.1.2	Search strategy	26
3.1.3	Study selection	29
3.1.4	Study quality assessments	32
3.1.5	Data extraction	32
3.1.6	Data synthesis	37
3.2	Results	38
3.2.1	Overview of selected studies	38
3.2.2	Types of machine learning techniques used (RQ1)	39
3.2.3	Predicted smells(RQ2)	42
3.2.4	The accuracy of machine learning techniques (RQ3)	43
3.2.5	The used datasets (RQ4)	51
3.2.6	The tools used to implement the machine learning algorithms (RQ5)	55
3.3	Discussion	56
3.3.1	Types of machine learning techniques used (RQ1)	56
3.3.2	Predicted smells (RQ2)	57
3.3.3	The accuracy of machine learning techniques (RQ3)	58
3.3.4	The used datasets (RQ4)	59
3.3.5	The tools used to implement the machine learning algorithms (RQ5)	62
3.4	Threats to validity	63
3.5	Chapter Summary	64
CHAPTER 4 EXPERIMENTS		66
4.1	Research hypothesis	66
4.1.1	The proposed ensemble learning models	68
4.2	Experiment setting	69
4.2.1	Experimental setup and Tool	69
4.2.2	Evaluation Measures	70

4.2.3	Evaluation method	70
4.2.4	Datasets Description and Pre-processing	70
4.3	Results and analysis	73
4.3.1	Performance evaluation of single classifiers	73
4.3.2	Performance evaluation of the proposed models	76
4.3.3	Impacts of the number of classifiers on the Average probability ensemble (APE) classification performance	80
4.4	Threats to validity	80
4.5	Chapter Summary	81
CHAPTER 5 CONCLUSIONS AND FUTURE WORK		83
5.1	Conclusion	83
5.2	Summary of the contributions	86
5.3	Future Works	87
REFERENCES		88
APPENDIX A SLR APPENDIX		106
APPENDIX B EXPERIMENT APPENDIX		112
B.1	Comparison between applied models	112
B.2	Normal APE models Results	113
VITAE		118

LIST OF TABLES

3.1	Online Databases	27
3.2	Number of collected studies	28
3.3	Quality assessment questions	33
3.4	Quality scores of selected studies.	33
3.5	Selected primary studies.	34
3.6	Data extraction card.	35
3.7	Addressed questions	36
3.8	Distribution of publication venues.	38
3.9	Distribution of machine learning techniques in the primary studies (PSs)	40
3.10	Machine learning family	40
3.11	The ML techniques used	41
3.12	Distribution of predicted code smells over studies	43
3.13	Validation methods	44
3.14	Validation of the learning mechanism	44
3.15	Standalone machine learning Accuracy Comparison	45
3.16	Standalone machine learning accuracy comparison, where GC: God Class, DC: Data Class, LM: Long Method, FE: Feature Envy, SS: Switch Statement, LP: Long Parameter List, MC: Message Chains, MM: Middle Man, LC: Lazy Class.	46
3.17	Standalone machine learning Accuracy Comparison continued	46
3.18	Suitable machine learning technique to predict a specific code smell . .	47
3.19	Standalone machine learning Accuracy Comparison	48

3.20	Application of boosting techniques	49
3.21	Comparison of accuracy of boosting techniques	49
3.22	Suitable machine learning algorithm to predict a specific Code smell . .	50
3.23	A summary of previous studies	51
3.24	Dataset size	52
3.25	Used systems	52
3.26	Software Metrics Used in Previous Studies	54
3.27	Used tools to compute the metric	55
3.28	Tools used to implement experiments	55
3.29	Bad Smell prediction Tools	61
3.30	Sampling Procedure	61
4.1	Used datasets	71
4.2	The obtained results of the single classifiers	75
4.3	Best Classifier for each dataset	76
4.4	Classification results of APE, APE weighted support vector machine (W-SVMs), Gradient Boosting (GB) and Random forests (RF) models.	78
4.5	Comparison between the proposed models and the related work	79
A.1	Used systems	106
B.1	Comparison between applied models	112

LIST OF FIGURES

1.1	Proposed models structure	6
2.1	Bad Smells prediction from Source Code	10
2.2	Bad Smells prediction from Software Design Model	10
2.3	Code smells categories	11
3.1	Systematic literature review (SLR) Process Phases	23
3.2	Selection Process Phase	30
3.3	Number of PSs published per year.	39
3.4	Machine learning taxonomy	42
4.1	Hypothesis H_a evaluation algorithm	67
4.2	APE structure	69
4.3	Best Classifier per dataset	75
B.1	Performance evaluation of the normal APE models	113
B.2	Performance evaluation of the normal APE models continued	114
B.3	Performance evaluation of the normal APE models continued	115
B.4	Performance evaluation of the normal APE models continued	116
B.5	Performance evaluation of the normal APE models continued	117

LIST OF ABBREVIATIONS

AIS	Artificial Immune Systems
ANN	artificial neural network
APE	Average probability ensemble
ASG	averaged stochastic gradient
AUC	Area Under the Curve
BBN	Bayesian Belief Network
BLR	Binary Logistic Regression
BNB	Bernoulli naive Bayes
CPU	Central Processing Unit
DFMC4J	Design Features and Metrics for Java
DFT	Decision Tree Forest
DPDG	Design Program Dependence Graph
DT	Decision Tree
FPR	false positive rate
GB	Gradient Boosting
GPC	Gaussian Process Classification

IBK	Instance-Bases Learning with Parameter k
IBL	Instance-based learning
LR	Linear Regression
MAE	Mean Absolute Error
ML	Machine Learning
MLP	Multilayer Perceptron
MNB	Multinomial Naive Bayes
NB	Naïve Bayes
NLP	Natural language processing
PS	primary studie
QMOOD	Quality Model for Object Oriented Design
RAM	Random-Access Memory
RBF	Radial Basis Function Network
RF	Random forests
RMSE	Root Mean Square Error
ROC	Receiver Operating Characteristics
SGD	Stochastic Gradient Descent
SLR	Systematic literature review
SMO	Sequential Minimal Optimization
SVM	Support Vector Machine
TPR	true positive rate

VFI	Voting Feature Intervals
W-SVMs	weighted support vector machine

THESIS ABSTRACT

NAME: Ahmed Ali Hamdan AL-Shaaby

TITLE OF STUDY: Software Bad Smells Prediction Using Advanced Machine Learning Techniques

MAJOR FIELD: Software Engineering

DATE OF DEGREE: December, 2019

Code smells are indicators of potential problems in software. They occur for many reasons, such as ignoring software development principles, writing code in a complex way, and incorrect integration of new modules into the system. They tend to have a negative impact on software quality. The main objectives of this thesis are to systematically review and analyze machine learning techniques used to detect code smells to provide interested research community with knowledge about the adopted techniques and practices for code smells detection, and to perform an empirical study to investigate the advantages of ensemble prediction models over standalone machine learning techniques in code smell prediction. In this thesis, we systematically review and analyze machine learning-based methods to predict software smells. The conducted SLR provides knowledge about the adopted techniques and identifies the gaps to be addressed.

Consequently, an empirical study is conducted to fill these gaps and to investigate the advantages of advanced machine learning approaches (ensemble learning) over standalone machine learning approaches in code smell prediction. The empirical study is carried out on 20 collected datasets. Eight single learning approaches are applied to predict the code smells and are evaluated. Then, we proposed heterogeneous and homogeneous average probability-based ensemble classification models to predict code smells. The results indicate that, the application of machine learning techniques to predict code smells in systems is still in its infancy and needs more research to facilitate the employment of machine learning approaches in predicting code smells. It also indicate that, the proposed technique APE outperformed other single classifiers in most cases. The highest results of 100% in terms of AUC is reported on most used datasets.

ملخص الرسالة

الاسم: احمد علي حمدان الشعبي

عنوان الدراسة: التنبؤ بالعيوب البرمجية باستخدام تقنيات التعليم الآلي المتقدمة

التخصص: هندسة البرمجيات

تاريخ الدرجة العلمية: ديسمبر, 2019

(Code smell) هي عبارة عن المؤشرات للمشاكل المحتملة في البرنامج. قد تحدث لعدة أسباب, مثلاً تجاهل مبادئ تطوير البرمجيات ، كتابة التعليمات البرمجية بطريقة معقدة، الإضافات الخاطيفة للوحدات النمطية الجديدة في النظام. هذه المشاكل تميل إلى أن يكون لها تأثير سلبي على جودة البرمجيات. في السابق تم إجراء عملية الكشف عن رائحة الكود بطريقة يدوية, لكنها كانت طريقة مجتهده وتحتاج الكثير من الوقت وفي نفس الوقت مكلفة. لا يزال تطبيق خوارزميات التعلم الآلي (Machine learning) للكشف عن روائح الكود في النظم في مراحله الأولى ويحتاج إلى مزيد من البحث لتسهيل استخدام خوارزميات التعلم الآلي في اكتشاف روائح الكود. في هذا البحث، قمنا بمراجعة وتحليل المنهجيات القائمة على التعلم الآلي للكشف عن روائح الأكواد. في هذا البحث، قمنا بمراجعة وتحليل المنهجيات القائمة على التعلم الآلي للكشف عن روائح الأكواد. توفر مراجعة الأدبيات المنهجية (SLR) التي أجريت معرفه حول التقنيات المستخدمة وتحدد الفجوات التي يجب دراستها ومعالجتها. ونتيجة لذلك ، أجريت دراسة تجريبية لملء هذه الفجوات وللتحقق من مزايا تقنيات التعلم الآلي المتقدمة (التعلم الجماعي) (Ensemble learning) على تقنيات التعلم الآلي المستقلة (Standalone machine learning) في الكشف عن مشكلة رائحة الكود البرمجي. تم في هذه الرسالة إجراء الدراسة التجريبية على عدد 20 مجموعة بيانات مجمعة من دراسات سابقة. تم تطبيق ثمان تقنيات من تقنيات التعليم الآلي المستقلة للكشف عن روائح الكود ومن ثم تقييمها. بعد ذلك، قمنا باقتراح استخدام نماذج تصنيف التعليم الجماعي في كلتا الحالتين متجانسة (homogeneous) وغير متجانسة (heterogenous). هذه النماذج تعتمد على حساب المتوسط للقيمة الاحتمالية للمصنفات لكشف عن رائحة الكود. تشير النتائج إلى

أنه في معظم الحالات ، تفوقه النماذج المقترحة للكشف والتنبؤ عن الروائح الكريهة في البرامج على بقية النماذج الأخرى في معظم حالات مجموعة البيانات المستخدمة, حيث حققت النماذج المحسنة على نتائج متميزة بنسبة 100 % من حيث AUC على مجموعات البيانات المستخدمة.

CHAPTER 1

INTRODUCTION

Code smells are indicators of potential problems in software. Code smells occur due to programming and design mistakes caused by software developers during the software designing and programming state [1]. They can also occur for other reasons such as an incorrect analysis, incorrect integration of new models into the system, ignoring software development principles, writing codes in a complex way, etc. [2], [1]. These smells may have a negative influence on the overall quality of the system, for example, maintainability and understandability [3], [4], [5], [6], [7]. Therefore, the code smell prediction process has motivated many researchers to propose different methods to deal with the occurrence of code smells in systems. Refactoring is proposed to alleviate and overcome code-smell-related issues. Refactoring leads to high quality, high performance, low cost, reusability, implementation and the easy development of software [2], [8]. Code smell prediction can be defined as the task of identifying potential code or design problems in a system [2], [1], [9], [10], [11].

Undertaking the code smell prediction process in a manual manner is considered to be subjective. Most of these techniques mainly depend on object-oriented metrics

that result in various outcomes [4]. Therefore, automated tools are proposed [12], [13], [14], [15], [16], [17], [18], [19], [20]. Nowadays, machine learning approaches are utilized to address code smell issues with promising results. A machine learning classifier needs first to be trained using a set of code smell examples to generate a model. The generated models are then used to identify or predict code smells in unseen or new instances. The power of the generated model relies on various criteria related to the dataset, the machine learning classifiers, the parameters of the classifier itself, etc.

Research on code smells prediction indicated the success of many techniques such as, Linear Regression [21], Decision Trees [22], Multilayer Perceptron [23], Bayesian Belief Networks, Naive Bayes [24], [4], [25], [26], Support Vector Machines [24], [27], [4], [21], [25], [28]. These algorithms are considered as suboptimal in the case of irrelevant, redundancy features datasets. The classification performance of these techniques goes worse if the dataset includes irrelevant or incomplete features [29]. In fact, ensemble learning techniques presented an adequate efficiency to overcome the dataset issues stated formerly. For example, random forests Random Forest outperforms the single algorithms in predicting software code smells [24], [4], [25], [26].

1.1 Problem Statement and Motivation

predicting software code smells is one of the essential software quality assurance activities. Because code smells may cause a negative influence on the overall quality of the system, for example, maintainability, reusability, and understandability [2], [1], [9].

Code smells prediction process was performed in a manual manner, yet it is tedious and expensive and Considered to be subjective. Next, it performed in an automatic using prediction tools, metric-based. However, these tools have several limitations, such as, the current detectors require the specification of thresholds that allow them to distinguish smelly and non-smelly instances: the selection of thresholds, hence, strongly impact the predictors' performance, and the results provided by predictors are usually different.

As a consequence, the code smell prediction using machine learning techniques got more attention from many researchers to propose different approaches to overcome code smells in software systems.

Unfortunately, there has been a lack of effort in this filed. Our thesis aims at filling this gap. In this thesis, we conducted research in code smells prediction. A prototype system of code smells prediction using machine learning approaches has been implemented to assess the performance of the proposed models in our thesis.

1.2 Research Objectives

The main objective of this thesis is to provide software engineers with the ability to predict code smells utilizing machine learning techniques. Conducting this thesis will face many challenges that can be outlined as a series of research questions. These questions differ in their difficulty level and type of contribution. Each research objective will be associated with the appropriate research questions. Each of the following objectives can be accomplished by answering the associated research questions that

will act as our guide through this research:

- **Objective 1: Provide a comprehensive empirical study of machine learning techniques applied to code smells prediction.**

The research questions related to this objective can be formulated as follows:

- **Research Question 1:** Which machine learning techniques have been applied to predict code smells?
- **Research Question 2:** Which code smells are most commonly predicted using machine learning techniques?
- **Research Question 3:** What are the accuracy measures of the machine learning techniques that have been used for code smell prediction?
- **Research Question 4:** What are the most commonly used machine learning tools in the context of bad smell prediction?
- **Research Question 5:** What datasets have been used for code smell prediction?
- **Research Question 6:** Investigate how existing code smells datasets were constructed?

- **Objective 2: Perform an empirical study to investigate the advantages of ensemble prediction models over standalone machine learning techniques in code smell prediction.**

The research question related to this objective can be formulated as follows:

- **Research Question 7:** To which extent do standalone machine learning techniques predict code smells.
- **Research Question 8:** To which extent do ensemble prediction models provide better prediction performance over standalone machine learning techniques?

1.3 Research Methodology

The code smells prediction task in this thesis is considered as a classification problem. Consequently, the task of predicting code smells is divided into several steps, namely: dataset collection, pre-processing, feature/attribute selection or reduction, training code smells models, classification, and evaluation. The following phases are followed in this thesis in order to answer our research objectives.

- **Phase 1: Systematic Literature Review.** An SLR is conducted to analyze previous machine learning studies in code smells prediction, and find the gap between these studies.
- **Phase 2: Datasets collection, Data analysis, and pre-processing.** Conducting a descriptive and statistical analysis on the collected datasets. This will help us examine and study our datasets. In addition, applying pr-processing techniques on the collected datasets.
- **Phase 3: Code Smells prediction using Single classifiers** Several machine learning techniques are applied to generate different models that are able to

predict the code smells. Particularly, we applied eight single learning approaches and evaluated them.

- **Phase 4: Code Smells prediction using ensemble learning approaches**

We proposed heterogeneous and homogeneous APE based ensemble classification models to predict code smells.

- **Phase 5: Models evaluations** K-fold evaluation method is applied to evaluate

the generated models. The generated models are evaluated using AUC metrics.

The classification results of the proposed models are reported and compared.

We also compare the proposed methods with related works. Figure 1.1 depicts

the high-level architecture of the proposed method.

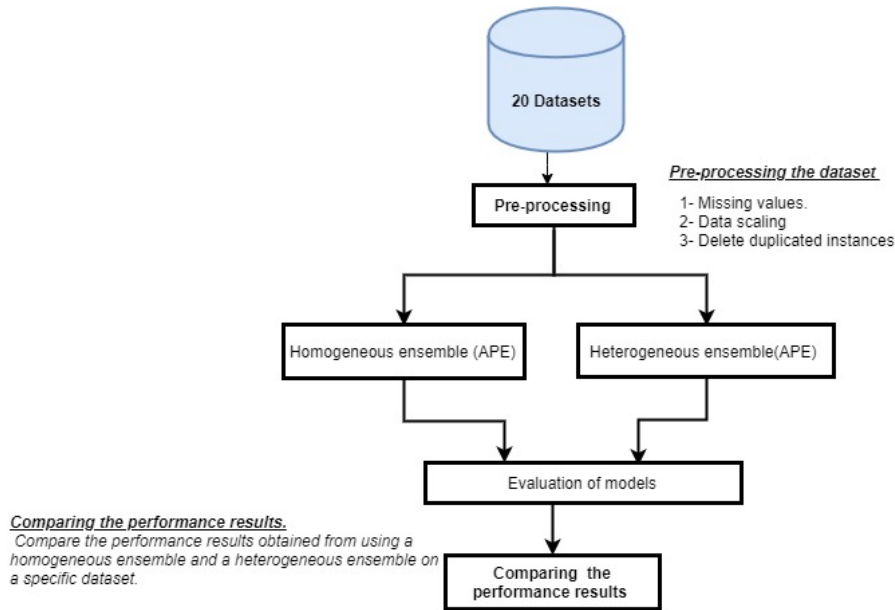


Figure (1.1) Proposed models structure

1.4 Thesis Contributions

The main contributions of this study are:

1. Conducting an SLR on code smells prediction using machine learning approaches to find the gaps. We have provided researchers with guidelines on how to conduct their studies or where the research gaps in this field. We Conducted different analyses on these primary studies to provide knowledge about them and evaluate them in terms of: (1) the applied techniques, (2) the predicted code smells, (3) the accuracy measures, (4) the used datasets, and (5) the most commonly used tools.
2. We have performed a comprehensive empirical study for standalone machine learning models in code smells prediction. We used numerous datasets.
3. We have proposed using ensemble learning models in code smells prediction and compared their performance results with standalone machine learning models. The proposed methods were compared favorably with the most related work.

1.5 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 presents a background of code smells, and machine learning techniques. Chapter 3 provides an SLR on studies carried out to predict code smells using machine learning algorithms.

Chapter 4 presents the experiments and results. It defines the hypothesis, describes

the proposed models, the experimental settings and describes the datasets. The pre-processing operations are also presented in this chapter. It also analyzes and discusses the obtained results. Chapter 5 presents the main findings of this thesis. Future directions are also presented in this chapter.

CHAPTER 2

BACKGROUND

In this chapter, we provide a brief background of code smells, and machine learning techniques.

2.1 Code Bad Smells

Code smells indicate potential code or design problems in a system [2], [1]. Code smells, also known as design flaws, refer to design situations that negatively influence the maintainability of the software [30], therefore, they may impact the maintenance processes [26]. A number of code smells were presented in Fowler’s book [31]. Fowler also suggested guidelines to eliminate these smells from the system [32].

It is helpful if we identify a bad smell as early as possible in the development lifecycle [26]. predicting bad smells in code or in the design and then performing the appropriate refactoring procedures when necessary is very useful to enhance the quality of the code. These smells make the system more difficult to maintain, probably also increasing its fault-proneness [33]. Bad smells are unlikely to result in failure directly

but might do so indirectly which still negatively impacts software quality [34].

Code metrics are used by different code smell prediction tools to predict code smells. These tools either compute the metrics from the code itself or utilize the extracted metrics from the external party's tools [4]. The bad smell prediction process can either be manual or automatic, using prediction strategies. These smells can be predicted in source code or system design as shown in Figure 2.1 or from system design in Figure 2.2.

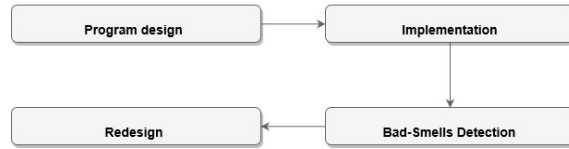


Figure (2.1) Bad Smells prediction from Source Code

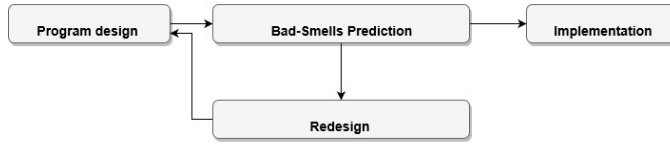


Figure (2.2) Bad Smells prediction from Software Design Model

2.1.1 Code smell categories

There are several categories of bad smells and each category contains several types. In our study, we selected the category proposed in [31], [35], [36], which is based on criteria that are related to the effects of these smells in each category, it also involves the most common code smells types that have the highest negative impact on software quality. The code bad smells are very closely relevant. Consequently, we

consider that the taxonomy makes the smells more recognizable and understandable.

Figure 2.3 details these categories.

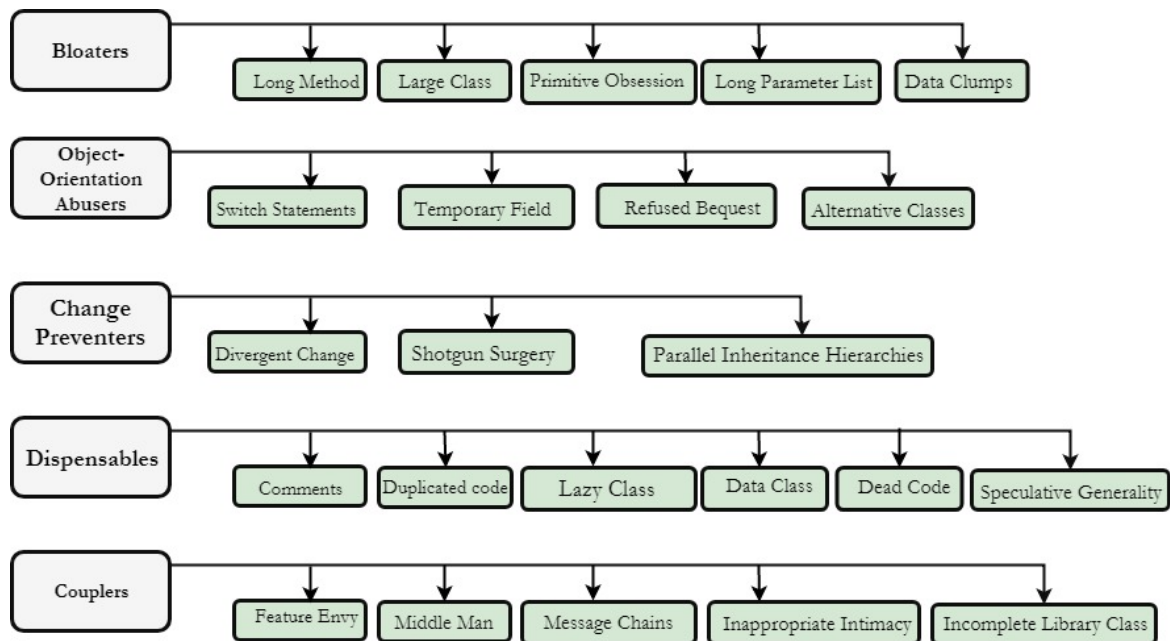


Figure (2.3) Code smells categories

- **Bloaters:** In this category, the code or classes are expanded to such a large extent that they are difficult to work with. These smells do not manifest immediately, rather they aggregate after some time as the program develops, particularly when no one endeavors to eliminate them. The first type of code smell in this category is the Long Method, which contains too many lines of code, making it difficult to reuse, change and understand. The best solution for this smell is to divide this method into separate methods. The second type of code smell in this category is Large Class. This occurs when a single class attempts to do too much, and it usually contains several instances and has various responsibilities. This smell makes the reusability and maintainability of this class more difficult.

The best solution for this smell is splitting this class, by applying extract class.

The third type of code smell in this category is Primitive Obsession used in the software. We should use small classes instead of primitive types in some situations. For instance, primitives are used in place of small objects for simple tasks, for example, special strings for phone numbers, ranges, and currency. The fourth type of code smell in this category is Long Parameter List. This type of smell occurs if any method has more than four parameters, making parameter lists more difficult to understand and use, and also inconsistent. The final type of code smell in this category is Data Clumps. Occasionally, various parts of the code contain identical groups of variables, for example, parameters to connect to a database. These clumps should be turned into their own classes.

- **Object-Orientation Abusers:** All the smells in this category involve the incomplete or incorrect application of the principles of object-oriented programming. The first type of bad smell in this category is the Switch Statements. This smell appears in the code when it contains a sequence of if statements or a complex switch operator. The second type of bad smell in this category is the Temporary Field. Usually, temporary fields are created for use in an algorithm that requires a lot of parameters. Therefore, instead of creating a large number of parameters, the programmer creates fields for this data in the class. These fields are utilized only in the algorithm and go unused the rest of the time. This kind of smell is difficult to discover. Removing this smell enhances code clarity and organization. The third type of bad smell in this category is Refused

Bequest. This smell occurs when programmers create inheritance between two completely different classes, but the subclass uses only a few of the methods and properties inherited from the superclass. The best way to treat this smell is to use Delegation instead of Inheritance. The fourth type of bad smell in this category is Alternative Classes with Different Interfaces. This code smell occurs when programmers create two classes with identical functionality, but these methods have different names.

- **Change Preventers:** These smells occur if you need to modify something in one place in your code, and then you have to make many modifications in other places too. Therefore, the development process of the software is more complicated and costly. In this category, there are three kinds of code smells. The first type of code smell in this category is Divergent Change. This code smell occurs when many changes are made to a single class. The best way to remove this smell is to split the class's behavior. For instance, in the case where different classes have the same behavior, the classes should be combined through inheritance. This will improve the organization of the code as well as reduce code duplication. The second type of bad smell in this category is Shotgun Surgery. This occurs when a single change is made to multiple classes simultaneously. The reason this smell occurs is because one responsibility has been split up amongst a large number of classes. The best way to remove this smell is to move existing class behaviors into a single class. This will improve the organization of the code, reduce code duplication and make it easier to maintain.

The third type of bad smell in this category is Parallel Inheritance Hierarchies.

This smell occurs when you create a subclass for a class and then discover that you need to create a subclass for another class.

- **Dispensable:** These smells occur when part of the code is not needed and where it to be removed, the code would be cleaner, more efficient and easier to understand. There are six types of bad smells in this category. The first type of bad smell in this category is Comments. This smell occurs when the program is filled with explanatory comments. The second type of bad smell in this category is the Duplicate Code. This smell occurs when the same or very similar code appears in several parts of the program, making the program code large. This bad smell can be removed by creating a new method that encapsulates the duplicated code. The third type of bad smell in this category is Lazy Class which is a useless class. Every class which is built takes effort and is time-consuming to understand and maintain. The best way to remove this smell is to eliminate these classes. The fourth type of bad smell in this category is Data Class. This is a class that contains only fields but there is seldom any logic to it. The Data Class has getters and setters methods for fields. The fifth type of bad smell in this category is Dead Code. This occurs when a code is never executed. The sixth type of bad smell in this category is Speculative Generality. This occurs when there is an unused parameter, field, method, or class. The reason this bad smell occurs is because sometimes, code is created to support anticipated future features that are never actually implemented. Consequently,

the code becomes difficult to understand and support.

- **Couplers:** All the smells in this category contribute to excessive coupling between classes or show what happens if the coupling is replaced by excessive delegation. There are six types of bad smells in this category. The first type of bad smell in this category is Feature Envy. This bad smell occurs a method accesses the data of another object more than its own data. It generally occurs when fields are moved to a data class. If this happens, the operations on data should be moved to this class as well. The second type of bad smell in this category is Inappropriate Intimacy. This occurs when one class uses the internal methods and fields of another class to do its work. The third type of bad smell in this category is Message Chains. This occurs when a client requests another object, and that object requests yet another object and so on. The fourth type of bad smell in this category is Middle Man. This occurs if a class performs only one action and delegates work to another class, hence there is little point in its existing. This smell can be the result of the overzealous elimination of Message Chains. The fifth type of bad smell in this category is Incomplete Library Class. This occurs when libraries no longer meet user requirements.

2.2 Machine learning techniques

Machine learning is a discipline where computer systems are able to learn and perform their work even if they were not explicitly programmed [37]. The most commonly used machine learning approaches in the literature on software quality prediction are su-

pervised learning, reinforcement learning, and unsupervised learning. This subsection describes the investigated classification models.

- **Multilayer Perceptron (MLP)** [38], [39] Is an artificial neural network (ANN) model that consists of a layer as an input, at least one hidden layer, and an output layer. Every node is a neuron that utilizes a nonlinear activation function and it associates with other nodes in the next layer with a specific weight. MLP constantly utilizes the backpropagation technique for training.
- **Support Vector Machine s (SVMs)** [40], [41] These are supervised learning models with related learning techniques and these learning algorithms can be used for regression or classification. SVM was defined by Vapnik [42] based on the principle of structured risk minimization. The main objective of SVM is empirical error minimization and geometric margin maximization. Commonly, the extent to which deviations are tolerated, the complexity parameter C , and the kernel are the parameters that are used to define the SVM model. The basic idea of W-SVMs is to add further weights to the training samples concerning with minority class. Hence, this model overcomes the issue of data imbalance [29].
- **Radial Basis Function Network (RBF)** [43], [44] This is a type of neural network that has three layers, one being an input layer, the second being a hidden layer, and the third being a linear output layer. Three types of RBFs are multiquadric, polyharmonic spline and Gaussian. RBF networks are used for classification, function approximation, and system control.
- **Bayesian Belief Networks (BBNs)** [45] This is a convenient graphical model

for representing a collection of variables and their probabilistic independencies. In this model, a random variable is represented by a node in the graph, whereas the probabilistic dependencies among the corresponding random variables are represented by the edges connecting the nodes.

- **Naïve Bayes (NB)** [46], [44] This is a supervised learning algorithm that employs the Bayes algorithm with the “naive” assumption of conditional independence among each pair of attributes.
- **Decision Tree (DT)** [47] This is one of the most successful options for the supervised learning method for regression and classification. The C4.5 algorithm is the most commonly used technique to generate decision trees [48].
- **Random forests (RF)** Is a supervised learning algorithm, contains many unpruned classifications or regression trees. such that every tree based on the values of a random vector experimented individually and with the same distribution for all trees in the forest. it can be employed for both classification and regression problems [49], [50].
- **Gradient Boosting (GB)** Is a machine learning technique for classification and regression problems, it proposed by Friedman to address regression issues by producing a prediction model that takes the form of an ensemble of weak prediction models [50]. Usually, these predictors are a set of decision trees. GB technique solves the classification issues by using an appropriate loss function to transform them into a regression [51].

- **Stochastic Gradient Descent (SGD)** Is a machine learning technique used to solve regression and classification issues, also called incremental gradient descent, this technique has effective methods to solve classification and regression issues that have huge datasets. This method called averaged stochastic gradient (ASG) techniques and second-order stochastic gradient. The idea behind this technique is to minimize a cost function [52].
- **Linear Regression (LR)** is a modeling method that is applied to find the correlation between the target and independent variables in the dataset by utilizing the linear predictor functions [21].
- **Multinomial Naive Bayes (MNB)** Multinomial naive bayes is a version of NB that is introduced for text classification. In MNB, the data samples follow a multinomial distribution [53], [46].
- **Bernoulli naive Bayes (BNB)** Is a version of NB. In BNB, the feature samples are independent booleans (binary variables) representing inputs [53], [46].
- **Gaussian Process Classification (GPC)** Is a popular classification method that is based on Bayesian methodology. It is a Nonparametric classification method that presumes some previous distribution on the underlying probability densities that support some smoothness characteristics [54].

CHAPTER 3

SYSTEMATIC LITERATURE

REVIEW

This chapter provides an SLR on studies carried out to predict code smells using machine learning algorithms.

A number of SLRs have been conducted in the area of bad smells. Zhang et al. [55] conducted an SLR on refactoring and code smells and reviewed papers published during the period of 2000 to 2009. They observed that most of the reviewed studies used a small number of code smells and some of these smells are used by the participants in a bad way (for example message chains). Likewise, Singh and Kaur [56] performed an SLR by reviewing 238 papers. They concentrated on the methods used to predict code smells as well the tools used for refactoring these code smells. Sharma and Spinelis [57] conducted another SLR and presented the existing knowledge associated with code smells, identified the challenges and investigated the definitions of code smells, reasons for their occurrence, their impact, and the available prediction tools. Santos et

al. [9] conducted an SLR to synthesize the existing knowledge on code smells. They concentrated on empirical studies that investigated how code smells affect software development. Mariani and Vergilio [58] presented an SLR of the existing research that motivates or applies search-based methods in the software refactoring activity. Their SLR was conducted by reviewing 71 primary studies. Several mechanisms have been proposed to predict code smells. Rasool and Arshad [32] performed a review on the existing tools that are used to predict code smells and report their associated challenges. Furthermore, Fernandes et al. [59] conducted a comparison between 84 smell prediction tools. Fontana et al. [30] conducted a literature review, they focused on code smells and on automatic tools. They identified 7 code smell prediction tools and evaluate 4 of them in terms of their prediction results. Garcia et al. [60] presented the concept of code smells that are frequently recurring software designs that can have non-clear and important detrimental impacts on system lifecycle properties.

A recent and related SLR was undertaken by Azeem et al. [61] in which they conducted a study to provide an overview of the use of machine learning approaches for code smell prediction. They identified 15 primary studies that used machine learning approaches. Their study focused on addressing four issues: 1) the code smells considered, 2) the setup of the approach, 3) the design of the evaluation strategies, and 4) an analysis of the performance. However, in this thesis, we further provide a detailed analysis of the datasets used in code smell prediction studies. Furthermore, we investigate the tools used to predict code smells, the tools used to extract and compute software metrics (features) and the tools used to implement machine learning techniques applied to predict code smells. Moreover, we investigate the use of

feature selection techniques and compare the standalone and ensemble-based machine learning techniques.

Another recent SLR was developed by Caram et al. [62], the authors in this study conducted a systematic mapping study of the use of machine learning techniques for code smells identification. 25 primary studies were identified. Their SLR concentrated on studying code smells prediction using machine learning techniques from different perspectives: 1) the predicted code smells, 2) the used machine learning techniques, 3) the most used machine learning techniques for each code smell, and 4) and the performance of each of these techniques for the code smells. These perspectives are covered in this study, also we present an extensive analysis of the used datasets in literature from different viewpoints: 1) the size of each dataset (i.e., number of included systems), 2) dependent variables, 3) independent variables, 4) the used tools to compute or extract metrics (independent variables), 5) the used tools to assign dependent variables(smelly or not smelly), and 6) description analysis of each available dataset (i.e., number of features, instances and smelly and nun-smelly instances in each dataset). Further, we explore the setup of used techniques such as, the used evaluation metrics, the used validation methods, classification type (i.e., binary classification or multi-classification). Furthermore, we investigate the use of ensemble machine learning techniques and compare them with stand-alone machine learning techniques. Moreover, we show the tools used to implement machine learning techniques. In addition, we investigate the use of feature selection techniques.

Unlike the above-related studies, we use a different method to compare the performance of the used machine learning techniques for code smells prediction. This method

called the vote-counting approach [63]. We believe that it is a suitable method in the situation when having studies used different evolution metrics to evaluate the performance of the same techniques for predicting the same code smells, or when having studies applied their experiments on different datasets. Therefore, in the current case, we can only report the findings of reviewed studies to find those machine learning algorithms that are able to predict code smells more consistently than other standalone machine learning techniques.

3.1 Research Methodology

The main objective of this SLR is to identify and analyze all relevant studies that use machine learning to predict code smells. As previously mentioned, we followed the SLR guidelines suggested by Kitchenham and Charters [64]. An SLR is a well-defined and systematic way of finding, assessing, and analyzing published primary research. The SLR gives a strong basis on which to make claims on research questions, but it needs considerably more effort than a traditional literature review [65]. The SLR process involves six phases, followed in sequence as shown in Figure 3.1:

- Research questions
- Search strategy design
- Study selection
- Quality assessment
- Data extraction

- Data synthesis

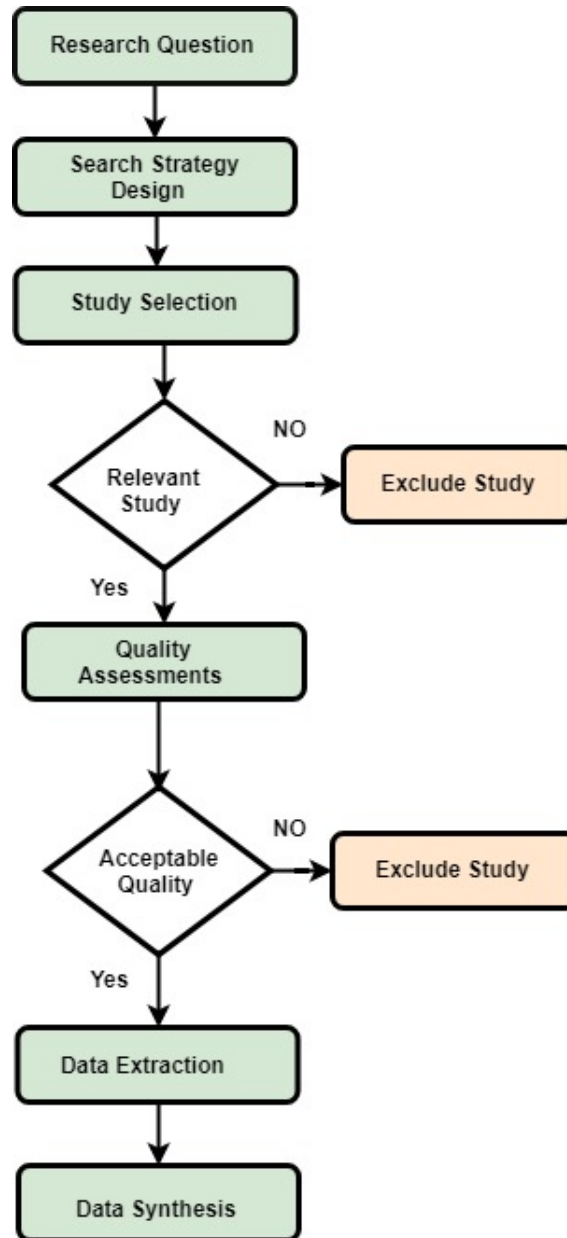


Figure (3.1) SLR Process Phases

In the first phase, a number of research questions based on the goal of the current SLR are defined. Then, a search strategy is built to find all the studies relevant to the research questions. Next, the search string, the digital libraries and the inclusion and the exclusion criteria are identified and in the fourth phase, a set of quality assessment

criteria are identified and applied to the selected papers. In the data extraction phase, the data extraction cards are created and employed to obtain data from the selected papers. Finally, in the data synthesis phase, appropriate methodologies are defined to synthesize the extracted data.

3.1.1 Research questions

Constructing the research questions is a significant step in the SLR process. Five research questions are defined to achieve our research objective:

1. **RQ1: Which machine learning techniques have been applied to predict code smells?** The objective of RQ1 is to identify the machine learning techniques that have been applied to predict code smells. Researchers can use the outcomes of this question to identify the most applied machine learning techniques for code smells prediction and to investigate the possibility of implementing unused techniques.
2. **RQ2: Which code smells are most commonly predicted using machine learning techniques?** Several studies have been conducted to address the issue of code smells, so our objective in relation to this question is to identify the code smells that have been predicted using machine learning and why researchers have chosen these code smells. The findings of this question can be used to determine the code smells that have not been investigated yet or got less attention by current studies. Hence, researchers can address them in future work.
3. **RQ3: What are the accuracy measures of the machine learning tech-**

niques that have been used for code smell prediction? To answer this question, we identify the performance metrics used to evaluate the machine learning techniques in terms of the prediction of code smells. Then, based on these performance metrics, we identify the accuracy of the machine learning techniques that have been used for code smell prediction. Next, we compare the machine learning techniques that have been used to predict code smells in order to find the most efficient. The findings of this question can be useful to identify the accuracy measures used to evaluate the performance of the used machine learning techniques. Such knowledge enables researches to use the most appropriate accuracy measure in their studies.

4. **RQ4: What datasets have been used for code smell prediction?** The objective of this question is to investigate the attributes of these datasets, such as: the dataset name, size (number of systems in each dataset, the size of each system), type (commercial, student, open-source), availability of the datasets (available online or not), the language of the selected systems, the inputs of the datasets, the tools used to obtain the values of dataset inputs, and an analysis of the available datasets. Researchers can use the outcomes of this question to build their datasets in a good way.
5. **RQ5: What are the most commonly used machine learning tools in the context of bad smell prediction?** Many tools are used to implement machine learning algorithms. To answer this question, we investigate the tools that are used to implement machine learning algorithms and explain why they

have been selected. The findings of this question can be useful to identify the most used tool; thus, researchers can select the most suitable tool for their needs.

3.1.2 Search strategy

This involves three phases: search terms, online databases, and the search process.

These are described in the following subsections.

Search terms

The common strategies applied to construct the search terms are described in this subsection as follows:

1. Obtaining the main terms from the research questions.
2. Finding the alternate synonyms and spelling for the obtained main terms.
3. Verifying the above steps with matching the keywords from any relevant research paper.
4. Managing Boolean operator "OR" for the link of alternative synonyms and spellings "AND" for the link of the major term and Boolean operator.

The search terms are built based on population, intervention, outcome and experimental design.

- **Population:** Code Smells.
- **Intervention:** The existing machine learning techniques for predicting Code Smells.

- **Outcomes:** Improve software quality.
- **Experimental Design:** Empirical studies, Case studies, Experiments studies.

After applying the previous steps along with several tests results, the following complete search terms are employed in our research.

(((((Code OR Bad) AND Smell*) OR “Antipatterns” OR “Refactoring”) AND (Detect* OR Predict* OR Estimat* OR Forecast*) AND (“Machine learning” AND (Model OR Technique OR Algorithm OR Method OR approach)) OR “artificial intelligence” OR “Ensemble learning”))

Research resources

Five online databases are used to find relevant conference and journal papers using our defined search terms. Table 3.1 presents these online databases. These online databases are considered to be important indexers, and well-known publishers [9]. Furthermore, they are well accepted by the software engineering community and they are employed in many other SLRs since they hold a huge amount of studies.

Table (3.1) Online Databases

Number	Name	URL
1	IEEE Explor	http://ieeexplore.ieee.org
2	Springer Link	http://link.springer.com
3	Science Direct	http://www.sciencedirect.com
4	Scopus	https://www.scopus.com
5	ACM Digital Library	http://dl.acm.org

The search terms are modified to be compatible with each online database since each has its own search engine syntax. Furthermore, the grammars for searching differ

from one to another. The start date of the searching process was open until December 2018.

In order to collect relevant material avoiding bias, a broad range of research databases are considered which includes all journal papers and conference papers.

Search process

In general, the SLR process involves a comprehensive search of all the studies that fit the selection criteria. The search process comprises the following two phases:

- **phase 1:** The five online databases are searched separately using the constructed search string. The identified studies are then collected to establish a set of candidate studies. Table 3.2 presents the results of this phase.
- **phase 2:** For each relevant study, the reference lists are scanned to find other relevant studies to be included into our candidate list.

Table (3.2) Number of collected studies

Name	# Collected studies	Initial selection	Final selection
IEEE Explor	147	20	10
Springer Link	247	5	2
Science Direct	166	5	3
Scopus	169	20	10
ACM Digital Library	700	6	2
Total	1429	56	27

It is expected that many references will be collected during the whole SLR process. EndNote is used to keep a record of all the references. Consequently, 1429 relevant studies are retrieved.

3.1.3 Study selection

The inclusion and exclusion criteria are defined in this subsection. 1429 candidate studies are retrieved as described in Table 3.2 and Figure 3.2. Most of them do not provide valuable information to fit the defined research questions. Therefore, more filtering is recommended in order to identify relevant studies. The study selection process is illustrated in Figure 3.2. The selection process is performed by one researcher and another two researchers verify the selection process at random. The selection process is carried out in two phases:

- **Initial selection:** Studies found during the initial search are assessed for relevance. This is done by analyzing the title and abstract as recommended by [66] and by applying the inclusion and exclusion criteria (defined below) to identify the relevant studies, which provide information to answer the research questions.
- **Final Selection:** Studies found in the initial selection underwent further analysis (i.e., reading full text) and the quality assessment criteria were applied. This is done so that no important information or data with respect to the research question is missed. These studies will be eventually used for data extraction.

Inclusion criteria

For a paper to be included in the SLR, it needs to meet various inclusion criteria.

- Studies that propose and discuss the use of machine learning techniques to predict code smells.

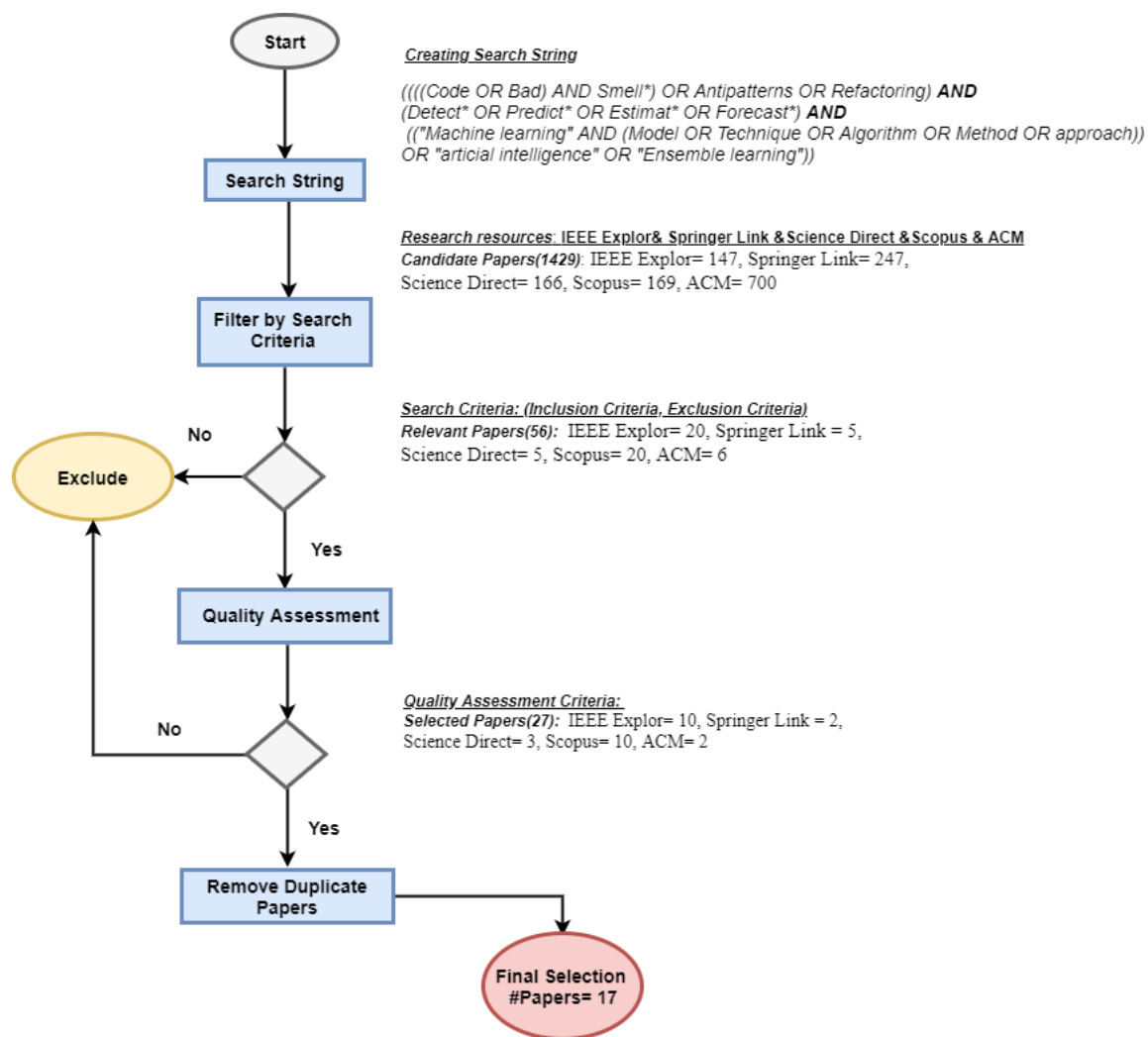


Figure (3.2) Selection Process Phase

- Studies that motivate and describe the benefits of using machine learning techniques to predict code smells.
- Studies that provide an empirical basis for their findings.
- In the case of duplicate studies, only the most complete and recent will be selected.
- Papers which have been published in a journal or in conference proceedings.

Exclusion criteria

Several exclusion criteria were established to make certain studies ineligible for inclusion in the SLR.

- Studies that are not applicable to the research questions.
- Studies that do not use or propose machine learning techniques to predict code smells.
- Studies that are not written in the English language.
- Publications that do not have an empirical analysis or findings from applying machine learning models to predict code smells.

As a result, 56 relevant studies were obtained, as described in Table 3.2. Next, we scanned the references in these relevant studies, but we did not find any additional relevant studies.

3.1.4 Study quality assessments

After applying the selection criteria, we define the quality assessment criteria for the selected papers according to the objectives of this research. These quality assessments are used to weight the selected studies. In this subsection, we list the quality assessment questions used for quantitative assessment of the quality of the selected papers where Yes=1, No=0 and partly=0.5. The quality assessment questions are listed in Table 3.3. The final score is computed by summing the values assigned to each question, where the higher the score, the higher the quality of the study. In our research, we selected only those papers with a quality score greater than 4.5. We used the same threshold (50%) that was used in [63]. Two researchers conducted these quality assessments of the selected studies separately. In the case of if there was disagreement between the researchers, this was discussed and resolved. Table 3.4 shows the quality scores of the selected studies. Seventeen studies were identified as the final papers for the data extraction process. These 17 studies are described in Table 3.5.

3.1.5 Data extraction

The selected papers are used to gather data that address our research questions. In order to make this process easier, cards are created with the form described in Table 3.6. This form is refined by pilot data extraction with many of the selected papers. To make the process of data synthesis as easy as possible, we group the items based on the association between the research questions, as shown in Table 3.6.

Table 3.6 demonstrates that the data extracted to address **RQ3**, **RQ4**, and **RQ5**

Table (3.3) Quality assessment questions

Q#	Quality questions	Yes	No	Partly
Q1	Are the research objectives clearly stated and well-motivated?			
Q2	Is the experiment design clearly defined?			
Q3	Are the performance measures used to measure the machine learning models clearly explained?			
Q4	Is the dataset size sufficiently stated?			
Q5	Are the predicted code smells clearly defined?			
Q6	Are the machine Learning models employed sufficiently defined?			
Q7	Are the independent variables clearly defined?			
Q8	Are the findings of the study sufficiently defined well-motivated?			
Q9	Does the study discuss threats to validity or limitations?			

Table (3.4) Quality scores of selected studies.

Study	Reference	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Score
S1	[24]	1	1	1	1	1	0.5	1	1	1	8.5
S2	[23]	1	1	0.5	1	1	0.5	1	0.5	0	6.5
S3	[27]	1	1	1	0.5	1	1	0.5	0.5	0.5	7
S4	[4]	1	1	1	1	1	1	0.5	1	1	8.5
S5	[21]	1	1	0.5	0.5	0	1	1	0.5	0	5.5
S6	[22]	1	1	1	0.5	1	1	1	0.5	0	7
S7	[25]	1	1	1	0.5	1	1	1	0.5	0	7
S8	[28]	1	1	1	1	1	1	1	0.5	1	8.5
S9	[67]	1	1	1	1	1	1	0.5	0.5	0.5	7.5
S10	[26]	1	1	1	1	1	1	0.5	0	0	6.5
S11	[68]	1	1	0.5	0	1	0.5	0	0.5	0.5	5
S12	[3]	1	1	1	0.5	1	0.5	0.5	0.5	0	6
S13	[69]	1	1	0.5	0.5	1	0.5	0.5	0.5	0	5.5
S14	[70]	1	1	1	0.5	1	1	0.5	0.5	0.5	7
S15	[71]	1	1	0.5	0.5	1	1	1	0.5	0.5	7
S16	[72]	1	0.5	1	0.5	1	1	0.5	0.5	0.5	6.5
S17	[73]	1	1	0.5	0.5	0	1	1	0.5	0	5.5

Table (3.5) Selected primary studies.

#	TYPE	Online Database	Year	Reference
S1	Journal	Scopus& Science Direct	2018	[24]
S2	Journal	Scopus	2017	[23]
S3	Conference	Scopus & IEEE	2017	[27]
S4	Journal	Springer Link	2016	[4]
S5	Conference	Scopus & IEEE& Science Direct	2016	[21]
S6	Conference	Scopus & IEEE	2015	[22]
S7	Conference	Scopus & IEEE	2013	[25]
S8	Conference	IEEE & ACM& Digital Library	2012	[28]
S9	Journal	Science Direct	2011	[67]
S10	Conference	Scopus & IEEE	2011	[26]
S11	Conference	Scopus & IEEE	2010	[68]
S12	Conference	Scopus & IEEE	2009	[3]
S13	Conference	Scopus	2005	[69]
S14	Journal	Springer Link	2015	[70]
S15	Conference	ACM	2012	[71]
S16	Conference	IEEE	2010	[72]
S13	Conference	IEEE	2009	[73]

are associated with the experiments conducted in the chosen papers. Generally, in the software engineering field, the word experiment has diverse meanings. Consequently, to avoid confusion, we explicitly define the word “experiment” as a process in which an algorithm or a model is assessed with suitable performance metrics depending on a specific dataset.

The data extraction cards were employed to obtain data from the chosen papers. Three researchers undertook this process. One extracted the data and then inserted these data into the cards, while the others examined the collected data. If there was disagreement between the researchers about the results, a discussion was conducted to resolve the conflict. Finally, the examined extracted data is documented in a file, to be utilized in the next process.

As shown in Table 3.7, all our research questions were answered by the primary

Table (3.6) Data extraction card.

Name of Extractor Date of Extraction Data Checker Paper ID Title of Paper Authors Name Publication Year Source Reference type(Journal/Conference) Study Type (experiment, case study) Publisher
RQ1: Which machine learning techniques have been applied to predict code smells? • Machine learning techniques applied to predict code smells.
RQ2: Which code smells are most commonly predicted using machine learning techniques? • The selected code smells in the field of using machine learning techniques to predict code smells, and why they have selected these smells?
RQ3: What are the accuracy measures of machine learning techniques that have been used for code smells prediction? • Metrics employed to estimate the prediction performance, prediction accuracy values and validation mechanism.
RQ4: What datasets have been used for code smell prediction? • Datasets used in the experiments. Dataset name, size, type (commercial, student, open source), availability, language, metrics used, and tools used to compute metric value.
RQ5: What are the most commonly used machine learning tools used in the context of bad smell prediction? • Tools used to implement machine learning techniques to predict code smells.

Table (3.7) Addressed questions

ID	Authors	The addressed research questions				
S1	D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia.	1	2	3	4	5
S2	D. K. Kim.	1	2	3	4	5
S3	A. Kaur, S. Jain, and S. Goel.	1	2	3	4	5
S4	F.A. Fontana, M. V. Mantyla, M. Zanoni, and A. Marino.	1	2	3	4	5
S5	Tarwani, Sandhya, and Anuradha Chug.	1	2	3	4	-
S6	L. Amorim, E. Costa, N. Antunes, B.Fonseca, and M. Ribeiro.	1	2	3	4	5
S7	F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla.	1	2	3	4	5
S8	A. Maiga, N. Ali, N. Bhattacharya, A. Sabane.	1	2	3	4	5
S9	F. Khomh, S. Vaucher, Y.-G. Gueheneuc, and H. Sahraoui.	1	2	3	4	5
S10	N. Maneerat and P. Muenchaisri.	1	2	3	4	5
S11	S. Bryton, F. B. e Abreu, and M. Monteiro.	1	2	3	4	-
S12	F. Khomh, S. Vaucher, Y.-G. Gueheneuc, and H. Sahraoui.	1	2	3	4	5
S13	J. Kreimer.	1	2	3	4	5
S14	Yang et al.	1	2	3	4	-
S15	Wang et al.	1	2	3	4	5
S16	Hassaine et al.	1	2	3	4	-
S17	Vaucher et al.	1	2	3	4	-

studies except five studies (**S5**, **S11**, **S14**, **S16**, **S17**) which did not address **RQ5** because they did not mention the tools used to implement the machine learning techniques. To facilitate the tracking of the extracted data, we marked every paper with an ID, Table 3.7 presents more detail.

3.1.6 Data synthesis

Data synthesis aims at aggregating the extracted data from selected papers to answer our research questions. In this study, the data extracted compose of qualitative data (e.g., a list of employed machine learning algorithms, size of each dataset, and a list of predicted code smell) and quantitative data (e.g., values of predicting accuracy). The number of selected papers was small only 17 studies. Therefore, we employed the canonical meta-analysis [64] as a synthesis methodology because of its solid theoretical background. Moreover, employing the standard meta-analysis with this number of selected studies inappropriate [64].

Different procedures were used to synthesize the extracted data relating to various research questions. The synthesis procedures are as follows.

- The narrative synthesis strategy was employed for **RQ1**, **RQ2**, **RQ4**, and **RQ5**.

We tabulated the data in a style compatible with the questions.

- For data concerning **RQ3** in which the authors compared the prediction accuracy of all the algorithms, the vote-counting approach was used [63]. For example, to synthesize the comparisons between algorithm X and algorithm Y, the number of studies in which the X algorithm outperformed the Y algorithm were counted, then the number of studies in which the Y algorithm outperformed the X algorithm were counted. Finally, we used these two numbers as the basis for analyzing the comparison between algorithm X and algorithm Y. Consequently, we can find whether one prediction algorithm outperforms another or not.

3.2 Results

In this section, we present the main results of this review. First, a summary of the selected papers is presented. Then, we document the main findings of our work based on the research questions. We explain the study outcomes not just within the research question context but also in a comprehensive manner relevant to the research questions.

3.2.1 Overview of selected studies

The issue of predicting code smells [74] has drawn the attention of many researchers over recent years [59]. The research literature can be categorized into two: empirical studies and prediction models. Empirical studies conduct research with the intention of knowing code smell growth [75], [76], [77], [78], their perception [6], [79], [80] and their influence on the quality attributes of source code [81], [82], [83]. Studies on prediction models depend on the analysis of structural information of the code itself [84], [85], [86]. The recent studies focus on the analysis of other sources of information [87], [88] or the utilization of software engineering techniques which are search-based [55], [34]. In the context of our research, we concentrate on machine learning techniques for predicting bad smells. Therefore, we review studies which utilize machine-learning techniques to predict bad smells.

Table (3.8) Distribution of publication venues.

Publication Type	# Studies
Conference	12
Journal	5

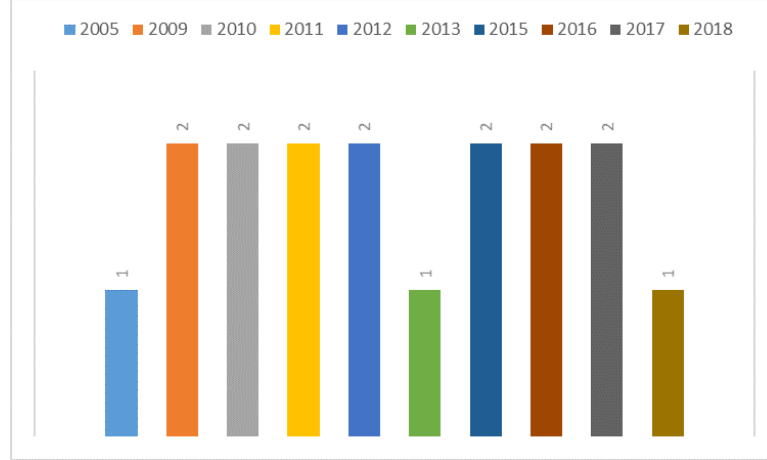


Figure (3.3) Number of PSs published per year.

In our study, we identified 17 primary studies (PSs), listed in Table 3.5, that relate to bad smell prediction using machine learning techniques, published between 2005 and 2018. As shown in 3.8 five of the PSs were published as journal articles, and 12 were published in conference proceedings. The distribution of the publication venues is presented in Tables 3.5, and 3.8. All the selected studies are experiment papers. The number of PSs published per year is shown in Figure 3.3. It can be observed that the number of studies published between 2009 and 2012, and between 2015 and 2017 remained stable with two studies published per year, whereas in 2005, 2013, and 2018, one study was published each year. Furthermore, there is a slight increase in the number of published studies in this area in recent years, which indicates that research attention in this area is growing.

3.2.2 Types of machine learning techniques used (RQ1)

Eighteen types of machine learning algorithms were employed to predict code smells.

Table 3.9 presents our findings. We found that 12 of our PSs applied a single machine

learning technique per code smell prediction while the remaining 5 studies applied more than one machine learning technique. The table presents the machine learning techniques used in descending order from the highest frequency to the lowest. Study **S10** used 7 machine learning techniques which was the highest.

Table (3.9) Distribution of machine learning techniques in the PSs

Used ML/Studies	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	Count	Frequency%
SVMs	✓		✓	✓	✓		✓	✓										6	35.29
J48	✓			✓			✓			✓								4	23.52
NB	✓			✓			✓			✓								4	23.52
RF	✓			✓			✓			✓								4	23.52
BBN									✓			✓			✓		✓	4	23.52
JRIP	✓			✓			✓											3	17.64
SMO	✓			✓			✓											3	17.64
DT C5.0						✓							✓					2	11.76
BLR										✓	✓							2	11.76
MLP		✓			✓													2	11.76
RBF					✓													1	5.88
DFT					✓													1	5.88
LR					✓													1	5.88
IBK										✓								1	5.88
VFI										✓								1	5.88
IBL										✓								1	5.88
NLP														✓				1	5.88
AIS																✓		1	5.88
# Of used ML techniques	6	1	1	6	5	1	6	1	1	7	1	1	1	1	1	1	1		

We observe that some of the applied machine learning techniques are related to the same family, for example, J48 and Decision Trees C5.0 are from the same decision tree family, as shown in Table 3.10 the family of the decision tree is the most commonly applied to predict code smells. In addition, we found that the authors of studies **S1**, **S4**, **S7** used the same machine learning techniques. The authors of **S4**, **S7** are the same whereas the authors of study S1 replicated study **S4**.

Table (3.10) Machine learning family

Technique Family	Techniques
ANN	RBF, MLP
DT	J48, JRIP, Decision Trees C5.0, DFT, RF
SVMs	SVM, SMO
Instant Based Learning	IBL, IBK

The process of identifying the machine learning methods that were used to pre-

dict code smells is based on the general name of the machine learning technique. For example, in the studies of [4], [24], they used SVM and Sequential Minimal Optimization (SMO) with different kernels and they also used boosting techniques for all the used techniques, as shown in Table 3.11. We identify only the general name of the techniques, so we identify six machine learning techniques which are SVM, J48, NB, RF, SMO and JRIP.

SVM was the most commonly used, being applied in 6 of the 17 studies (35.29%) and the second most commonly used machine learning techniques are J48, NB, BBNs, and RF, being applied in 4 of the 17 studies (30.77%). Table 3.9 shows the detailed information.

Table (3.11) The ML techniques used

Used machine learning techniques	Study
B-J48 Pruned, B-J48 Unpruned, B-J48 Reduce Error Pruning, B-JRip, RF, B-RF, NB, B-NB, B-SMO, RBF Kernel, B-SMO Poly Kernel, B-LibSVM C-SVC Poly Kernel, B-LibSVM C-SVC Linear Kernel, B-LibSVM C-SVC Radial Kernel, B-LibSVM C-SVC Sigmoid Kernel, J48 Pruned, J48 Unpruned, J48 Reduce Error Pruning, JRip, SMO Polynomial, SMO, RBF, LibSVM C-SVC Poly Kernel, LibSVM C-SVC Linear Kernel, LibSVM C-SVC Radial Kernel, LibSVM C-SVC Sigmoid Kernel.	S1, S4
B-LibSVM ν -SVC Linear Kernel, B-LibSVM ν -SVC Poly Kernel, B-LibSVM ν -SVC Radial Kernel, B-LibSVM ν -SVC Sigmoid Kernel, LibSVM ν -SVC Linear Kernel, LibSVM ν -SVC Polynomial Kernel, LibSVM ν -SVC Radial Kernel, LibSVM ν -SVC Sigmoid Kernel.	S4

We can observe that SVM, J48, RF, NB, SMO, JRIP received the most attention in 2013, 2016, and 2018 while in 2017, only Neural Networks received attention. We further observe that the authors of the primary studies applied the most commonly used machine learning techniques.

In general, the machine learning technique is categorized into two types: supervised, unsupervised. Figure 3.4 shows this taxonomy [89], [90]. According to our findings, we observe that all the applied techniques in the field of code smell prediction were supervised machine learning techniques. The findings were expected since

the application of supervised techniques is the most often applied in studies [91], [92].

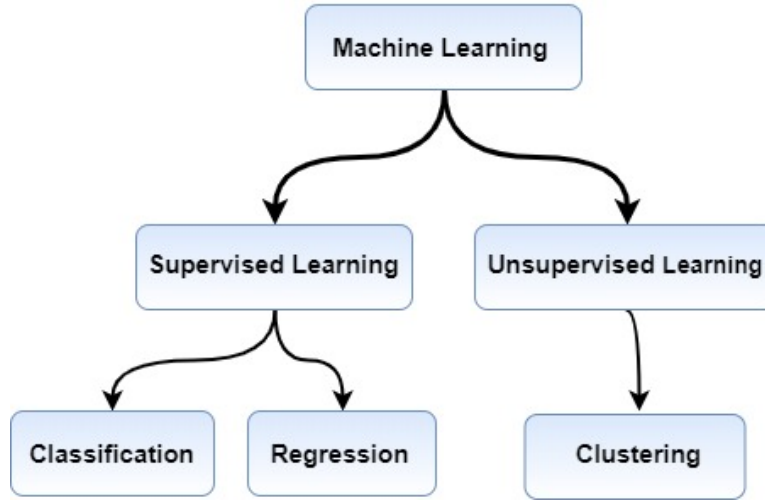


Figure (3.4) Machine learning taxonomy

3.2.3 Predicted smells(RQ2)

Twenty-eight types of code smells were used in the PSs. Table 3.12 presents our findings. We found that 12 of the PSs used more than one code smell while the other five studies used one code smell. The table lists the used code smells in descending order from the highest frequency to the lowest.

We found that the authors of studies **S1**, **S3**, **S4**, **S7** not only used the same code smells, they also used the same number of code smells. The authors of **S4**, **S7** are the same whereas the authors of study **S1** replicated study **S4**. **S5** used the highest number of code smells, using 11 code smells.

We observe that the God Class was the most used bad smell, being used in 13 studies (76.47%). The second most used bad smell was the Long Method which was used in nine studies (52.94%). The Feature Envy code smell was used in seven studies

(41.17%) and the Data Class was used in five studies (29.41%). The remaining code smells were used in a smaller number of studies ranging from one to three.

Table (3.12) Distribution of predicted code smells over studies

predicted Smells/Studies	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	Total	Frequency%
God Class	✓	✓	✓	✓	✓	✓	✓	✓	✓							✓	✓	13	76.47
Long Method	✓		✓	✓	✓	✓	✓			✓	✓		✓					9	52.94
Feature Envy	✓	✓	✓	✓	✓		✓			✓								7	41.17
Data Class	✓	✓	✓	✓			✓											5	29.41
Lazy Class		✓				✓				✓								3	17.64
Spaghetti Code instances								✓	✓							✓		3	17.64
Functional Decomposition (FD)								✓	✓							✓		3	17.64
Large Class		✓				✓												2	11.76
Message Chains						✓				✓								2	11.76
Long Parameter List						✓				✓								2	11.76
Swiss Army Knife(SAK)						✓		✓										2	11.76
Duplicated Code														✓	✓			2	11.76
Parallel Inheritance Hierarchies		✓												✓	✓			1	5.88
Type checking					✓													1	5.88
AntiSingleton						✓												1	5.88
Refused Parent Request						✓												1	5.88
Speculative Generality						✓												1	5.88
private class data						✓												1	5.88
Middle Man										✓								1	5.88
Complex Class						✓												1	5.88
Empty catch block					✓													1	5.88
Careless cleanup					✓													1	5.88
Dummy handler					✓													1	5.88
Unprotected main					✓													1	5.88
Nested try statement					✓													1	5.88
Exception thrown in finally block					✓													1	5.88
Switch Statement										✓								1	5.88
Over logging					✓													1	5.88
# Of predicted smells	4	6	4	4	11	12	4	4	3	7	1	1	2	1	1	3	1		

3.2.4 The accuracy of machine learning techniques (RQ3)

To answer this question, we explore the evaluation metrics used in the PSs to evaluate the performance of the machine learning techniques used in code smell prediction. Then, we compare the machine learning techniques that were used in code smell prediction to identify the techniques that achieved the highest prediction performance.

Several evaluation metrics were used by the PSs to evaluate the performance of the machine learning techniques used in code smell prediction. These evaluation metrics are presented in Table 3.13 in descending order according to their frequency in the PSs. We notice that the recall metric was the most frequently used metric, being used in 9 of the 17 studies, followed by the accuracy metric, which was used in 8 of

the 17 studies. Different configurations were used in the PSs to validate the learning mechanism. We present these mechanisms in Table 3.14. We observe that, 7 studies (**S1**, **S4**, **S5**, **S7**, **S10**, **S15**, **S16**) adapted K-fold cross-validation method. This method guarantees that each fold includes the same proportions of the smelly and non-smelly classes. Our findings show that only 2 studies (**S1**, **S4**) used the Grid-search algorithm.

Table (3.13) Validation methods

Validation Method	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	Frequency
Recall			✓			✓		✓	✓	✓		✓			✓	✓	✓	9
Accuracy	✓	✓		✓			✓			✓	✓		✓	✓				8
Precision			✓			✓		✓	✓			✓				✓	✓	7
F-measure	✓			✓		✓	✓											4
ROC Area	✓			✓			✓											3
Specificity										✓								1
Mean Absolute Error (MAE)					✓													1
Root Mean Square Error (RMSE)					✓													1

Table (3.14) Validation of the learning mechanism

Validation mechanism	Studies
<i>K-fold cross-validation</i>	S1, S4, S5, S7, S10, S15, S16
<i>leave-one-out method</i>	S13

To find the accuracy of the machine learning techniques that were used for code smell prediction, we performed an analysis of standalone machine learning techniques and ensemble techniques. **S5** [21] is the only study that ranks the used machine learning algorithms in terms of their ability to predict all the selected code smells in that study (see Table 3.15). In contrast, the other studies evaluated the used standalone machine learning algorithms based on their ability to predict a specific code smell. For example, machine learning *X* is better than machine learning *Y* in predicting code smell *Z*. Therefore, we extract these evaluations from the selected studies to find those standalone machine learning algorithms which are able to predict code smells more consistently than other standalone machine learning techniques.

In **S2** [23], Kim presented a bad smell prediction approach with the neural network model which presents the correlation among object-oriented metrics and bad smells using a collection of projects written in Java as an experimental dataset. This approach was developed to predict six kinds of bad smells namely Data Class, Feature Envy, Lazy Class, Large Class, God Class, and Parallel Inheritance Hierarchies. Kim concluded that applying many hidden layers improves accuracy.

The strategy which was followed to answer this question is presented in Tables 3.16 and 3.17. The main attributes of our evaluation are:

- The used machine learning algorithm.
- The considered code smell.
- The study source that mentioned the evaluation.
- A lower or a higher Prediction (The sign of ”+” means higher while ”-” is lower).

Table (3.15) Standalone machine learning Accuracy Comparison

Model	Rank
SVM	1
Decision Tree Forest (DFT)	2
MLP	3
LR	4
BBNs	5

For example, **S4** [4], the J48 algorithm performs better than NB to predict the code smell of Data Class, Long method and Feature Envy. However, this study found that the NB algorithm is better than the J48 algorithm in terms of predicting the God Class code smell. Therefore, the algorithm in the row of the table is better than the algorithm in the column if the sign is “+” and the opposite in case the sign is “-”.

Table (3.16) Standalone machine learning accuracy comparison, where GC: God Class, DC: Data Class, LM: Long Method, FE: Feature Envy, SS: Switch Statement, LP: Long Parameter List, MC: Message Chains, MM: Middle Man, LC: Lazy Class.

	NB		RF		JRIP		SVM		SMO	
	+	-	+	-	+	-	+	-	+	-
J48	(S1, S4) DC, FE, (S4) LM	(S1, S4) GC, (S1) LM	(S1, S4) GC, (S1) DC, GC, FE, (S1) LM	(S4) DC, LM, FE	(S1, S4) DC, GC, FE, (S1) LM	(S4) LM, FE	(S1,S4) DC, GC, FE, LM		(S1,S4) DC, GC, FE, LM	
NB			(S1),DC, LM, FE, (S1, S4) GC	(S4),DC, LM, FE	(S1, S4) GC, (S1) LM, FE	(S1, S4) DC , (S4) LM, FE	(S1,S4) DC, GC (S1) LM, FE	(S4) LM, FE	(S1,S4) GC, (S1) DC, LM, FE	(S4) DC, LM, FE
RF					(S4) DC, LM, FE, GC	(S1) DC, LM, FE, GC	(S4) DC, LM, FE, GC	(S1) DC, LM, FE, GC	(S4) DC, LM, FE, GC	(S1) DC, LM, FE, GC
JRIP							(S1, S4) DC, LM, FE, GC		(S1, S4) DC, LM, FE, GC	
SVM									(S1, S4) GC, (S1) DC, LM, FE	(S4) DC, LM, FE

Table (3.17) Standalone machine learning Accuracy Comparison continued

	NB		RF		Binary Logistic Regression (BLR)		IBK		VFI		IBL	
	+	-	+	-	+	-	+	-	+	-	+	-
J48	(S1, S4) DC,(S10) FE, (S10) LM, (S10) LC, SS, LP, MM	(S10) MC	(S10) LM, LC, SS, MM	(S10) FE, LP, MC, SS, MM	(S10) LM, FE, LC, SS, MM	(S10) LP, MC	(S10) LC, SS, MM	(S10) LM, FE, LP, MC	(S10) LC, SS, MM, LM, FE, LP, MC		(S10) LC, SS, MM	(S10) LM, FE, LP, MC
NB				(S10) LM, FE, LC, SS, MM, LP, MC		(S10) LC, SS, MM, LM, FE, LP, MC		(S10) LC, SS, MM, LM, FE, LP, MC	(S10) LC, SS, MM, LM, FE, LP, MC		(S10) LC, SS, MM, LM, FE, LP, MC	
RF					(S10) LC, SS, MM, LM, FE, LP	(S10), MC	(S10) LC, FE, LP	(S10) LM, SS, MM, LM, FE, LP, MC	(S10) LC, SS, MM, LM, FE, LP, MC		(S10) SS, LC , FE, LP	(S10) LM MM, MC
BLR							(S10) FE	(S10) LM, LC, LP, SS,MM	(S10) FE, LM, LC, LP, SS, MM		(S10) FE, LM, LC, LP, SS, MM	
IBK									(S10) FE, LM, LC, LP, SS, MM		(S10) LM, LC, LP, SS, MM	(S10) FE
IBL									(S10) FE, LM, LC, LP, SS, MM			

For more clarification, Table 3.18 compares the used machine learning techniques per the considered code smell. The best machine learning algorithm for predicting the Data Class smell is the J48 algorithm, followed by JRip. For the Long Method smell, the JRip algorithm is the best followed by J48. RF is the best at predicting the Feature Envy code smell, whereas the NB algorithm is the best at predicting God Class. In conclusion, the standalone machine learning algorithm that consistently exceeds the other standalone machine learning algorithms was the J48 algorithm, whereas the worst algorithm was Voting Feature Intervals (VFI). Table 3.18 sorts these algorithms for each code smell based on the reported accuracy.

Table (3.18) Suitable machine learning technique to predict a specific code smell

Code Smell	Suitable machine learning technique
Data Class	J48, JRIP, RF, NB, SMO, SVM.
Long Method	J48, JRIP, IBK, NB, RF, IBL, BLR, SMO, SVM, VFI.
Feature Envy	RF, J48, IBK, NB, JRIP, IBL, SMO, SVM, BLR, VFI.
God Class	NB, J48, RF, JRIP, IBK, SMO, SVM, IBL, BLR, VFI.
Long Parameter List	RF, IBK, BLR, J48, IBL, NB, VFI.
Middle Man	J48, IBK, IBL, RF, BLR, NB, VFI.
Meassage Chains	IBL, IBK, BLR, RF, NB, J48, VFI.
lazy class	J48, RF, IBK, BLR, IBL, NB, VFI.
Switch Statement	J48, IBK, RF, BLR, IBL, NB, VFI.

The remainder of the studies applied one standalone machine learning technique to predict the type of code smell. Table 3.19 depicts the evaluation conducted by these studies. In study **S3** [27], the authors applied SVM to predict God Class, Data

Class, Long Method, and Feature Envy. The results of this study show that the SVM achieved the highest accuracy in predicting the God Class code smell. Additionally, **S8** [28] applied SVM to predict four types of code smells namely: Spaghetti Code, Functional Decomposition, Swiss Army Knife. Their results support **S3** [27] in the case of predicting God Class. Consequently, the SVM algorithm has a good estimation of the God Class code smell. In **S13** [69], the Decision Tree Forest is more accurate for predicting God Class than the Long Method.

Table (3.19) Standalone machine learning Accuracy Comparison

Model	Suitable machine learning
SVM	(S3, God Class, Data Class, Long Method, Feature Envy), (S8, God Class, Spaghetti Code, Swiss Army Knife, Functional Decomposition)
DFT	(S13, God Class, Long Method)
BLR	(S11, Long Method)
BBNs	(S11, God Class), (S9, God Class, Swiss Army Knife, Functional Decomposition, Spaghetti Code), (S15, Duplicated Code), (S17, God Class)
Natural language processing (NLP)	(S14, Duplicated Code)

Some studies used ensemble techniques. Therefore, we need to compare the performance of these techniques to find the machine learning technique that performs better than the others. We follow the same strategy to compare the standalone machine learning techniques. Table 3.20 shows the studies which used the boosting techniques.

Three of the 17 selected studies used boosting techniques (**S1** [24], **S4** [4], **S7** [25]). They applied the same algorithm and predicted the same types of code smells and there is also a high similarity in the datasets used, especially **S1** [24], **S4** [4]. The

authors in study **S1** [24] replicated the experiment conducted in study **S4** [4] and made some modifications to the datasets that were built in **S4** [4].

Table 3.21 shows which strategy was applied in each of the studies. The main attributes of our evaluation are the name of the machine learning algorithm used, the predicted code smell, the study source that mentioned the evaluation, and a lower or a higher prediction (“+” indicates higher while “-” indicates lower).

Table (3.20) Application of boosting techniques

Techniques	Studies
Standalone machine learning	S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17
Boosting	S1, S4, S7

Table (3.21) Comparison of accuracy of boosting techniques

	B-NB		B-RF		B-JRIP		B-SVM		B-SMO	
	+	-	+	-	+	-	+	-	+	-
B-J48	(S1, S4, S7) DC, GC, (S4, S7) LM, FE	(S1) LM, FE	(S1, S4) DC, FE, GC, (S1) LM	(S7, S4) LM, (S7) DC, FE, GC	(S1, S4, S7) DC, GC, LM, FE		(S1, S4, S7) DC, GC, LM, FE		(S1, S4, S7) DC, GC, LM, FE	
B-NB			(S1) DC, GC, LM, FE	(S4, S7) DC, GC, LM, FE	(S1) GC, FE	(S1, S4, S7) DC, LM, (S4,S7) GC, FE	(S1, S4, S7) DC, (S1, S4) GC LM, (S4) GC, (S1) FE.	(S7) LM, (S1, S7) GC (S4, S7) FE, (S1)DC.	(S1,S4) LM, FE, (S1) GC	(S1, S4, S7) DC (S4, S7) GC, (S7) LM, FM
B-RF					(S4, S7) GC, (S4) LM, (S7) DC	(S1, S7) LM, (S1, S4, S7) FE ,(S1, S4) DC, (S1) GC	(S4, S7) DC, LM, GC, FE	(S1) DC, LM, GC,FE	(S4, S7) DC, LM, GC,FE	(S1) DC, LM, GC, FE
B-JRIP							(S1, S4, S7) DC, LM, FE, (S4, S7) GC	(S1) GC	(S1, S4, S7) DC, LM, FE, (S4, S7) GC	(S1) GC
B-SVM									(S1) LM, FE, GC	(S1, S4, S7) DC, (S4, S7) GC, LM, FE

Table 3.22 presents the code smell and which machine learning algorithm has performed better than the others in relation to predicting the code smell. Table 3.22

shows that the best machine learning algorithm to predict Data Class smell is the B-J48 pruned algorithm. The second most effective algorithm is the B-JRip, followed by the B-RF algorithm. The B-J48 unpruned algorithm was the best at predicting the Long Method smell, followed by B-JRip. The B-J48 algorithm and the B-JRip algorithm were the two most effective in predicting all code smells except for God Class where B-RF was the second most effective algorithm. We conclude that the family of J48 techniques and B-RF are the top techniques for all types of code smells while the SVM techniques are the worst. Furthermore, we notice that, in some cases, using the boosting technique on the models does not always enhance their performance, and in fact, it could make the model worse. For instance, in **S4** [4], using NB to predict the God Class code smell without using boosting techniques was the best algorithm but after applying boosting techniques, its performance declined. This was also the case when using RF with boosting techniques to predict God Class.

We observe that the highest performance results in all PSs obtained by B-J48 unpruned as it achieved 99.63% of F-Measure value when it was used to predict Long method code smell, followed by B-J48 pruned which achieved 99.26% of F-Measure value when it was used to predict Data Class code smell.

Table (3.22) Suitable machine learning algorithm to predict a specific Code smell

Code Smell	Suitable machine learning algorithm
Data Class	B-J48, B-JRIP, B-RF, B-SMO, B-SVM, B-NB.
Long Method	B-J48, B-JRIP, B-RF, B-NB, B-SMO, B-SVM.
Feature Envy	B-J48, B-JRIP, B-RF, B-NB, B-SMO, B-SVM.
God Class	B-J48, B-JRIP, B-RF, B-SMO, B-SVM, B-NB.

3.2.5 The used datasets (RQ4)

In this section, we analyze the datasets that were used for code smell prediction and identify several attributes of these datasets, such as: the dataset name, size, type (commercial, student, open source), availability and language.

Table 3.23 presents general information on the datasets used in the 17 studies. 14 studies used open-source systems except for **S10**, **S13**, and **S15** they used industrial systems. All studies used systems are written in the Java language except for **S14**, **S15**, they used systems written in C and C#, respectively. Most studies used software metrics as the independent variables, as shown in Table 26, except for S14 they used textual metrics called string tokenization. Two studies of the 17 made their dataset available. Table 3.23 presents the number of systems used to build their datasets, the dataset input type, dataset type, language, and the number of used metrics.

Table (3.23) A summary of previous studies

Study	Dataset	# Systems	Dataset Inputs	Dataset type	Language	#Used metrics
S1	—	74	Object oriented metrics	Open source	Java	36
S2	—	20	Object oriented metrics	Open source	Java	8
S3	—	2	Object oriented metrics	Open source	Java	—
S4	Available	74	Object oriented metrics	Open source	Java	82
S5	—	4	Object oriented metrics	Open source	Java	17
S6	—	4	Object oriented metrics	Open source	Java	62
S7	—	76	Object oriented metrics	Open source	Java	82
S8	Available	3	Object oriented metrics	Open source	Java	50
S9	—	2	Goal Question Metric (GQM)	Open source	Java	—
S10	—	—	Object oriented metrics	Open source	Java	27
S11	—	1	Expert's knowledge&Object oriented metrics	Industrial	Java	—
S12	—	2	Rule based	Open source	Java	—
S13	—	2	Object oriented metrics	Industrial	Java	9
S14	—	4	Textual	Open source	C	—
S15	—	2	Object oriented metrics	Industrial	C#	21
S16	—	2	Goal Question Metric (GQM)	Open source	Java	—
S17	—	2	Object oriented metrics	Open source	Java	—

Table 3.24 shows the size of each dataset in terms of three attributes: the number of used systems, the number of classes and the number of lines of code. We observed that only three studies used more than 20 systems. Therefore, we can report these as

reliable datasets. Some of the studies did not mention the size (number of classes and the number of lines of code) of their datasets such as (**S3**, **S6**) but they mentioned the number of systems. In **S3**, there are 20 systems and in **S7** there are 76 systems. The authors of **S10** did not mention the attributes (the number of used systems, the number of classes and the number lines of code) as they only collected 7 datasets from the previous literature. We observe that 14 PSs used open source systems. Therefore, they are freely available and can be used to replicate the research. Furthermore, other researchers who utilized the same systems allow comparison. Additionally, previous studies reported that commercial systems may provide higher code quality than open source systems [93].

We found that, 4 studies (**S1**, **S2**, **S4**, **S7**) build their datasets from different systems, while, the other studies build their datasets based on one system. Building datasets from different systems is the best way to avoid the issue of bias.

Table (3.24) Dataset size

# systems	#Classes	#LOC	Study
74	51,826	6,785,568	S1, S4
2		5,594,000	S15
20	53,000	3,000,000	S2
4	13700	4070000	S6
4	1089	—	S5
3	3,192	113,017	S8
2	777	271000	S9, S12,S16
2	688	96889	S13
1	193	20	S11

Table (3.25) Used systems

System	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	Frequency
Xercesv	✓		✓	✓			✓	✓	✓			✓				✓	✓	9
ArgoUML	✓		✓	✓		✓	✓	✓								✓		7
WEKA	✓			✓			✓						✓					4
Junit-4.10	✓	✓		✓			✓											4

Table 3.25 shows the four most used systems used in the 13 PSs. Table A.1 in the Appendix A lists all the systems that were in the PSs.

Table 3.26 presents the software metrics used as independent variables in the datasets. We can see that only seven of the thirteen studies mentioned the software metrics. These metrics include the metrics proposed by Chidamber and Kemerer in 1991 ¹, MOOD metrics proposed by Abreu and Carapuça in 1994 ², and the Quality Model for Object Oriented Design (QMOOD) metrics proposed by Bansiya and Davis in 2002 ³. We observe that the C&K software metrics were used in eight of the seven studies that mentioned software metrics as C&K metrics cover different Object-Oriented properties, while the QMOOD software metrics were used in 2 studies. The MOOD software metrics were used in one study. To obtain the values of these metrics from the systems, several tools were used. These tools are shown in Table 3.27. From Table 3.27, we observe that the tools POM and Design Features and Metrics for Java (DFMC4J) were used in three of the seventeen studies. The DFMC4J tool was used in two studies **S4**, **S7** which have the same authors, and it was also used in **S1** which replicated the experiment in **S4**.

Regarding the dependent variables in each dataset, 12 studies used binary classification (“0” or “1”) or (“smelly” or “non-smelly”), while only two studies **S1**, **S16** used multiclassification. Therefore, future studies should consider multiclassification.

Table (3.26) Software Metrics Used in Previous Studies

Authors	Software Metrics
Kim [23]	LOC, CC, C&K metrics.
Fontana et al. [4]	"C&K metrics except (LCOM), LOC, LOCNAMM, NOPK, NOCS, NOM, NOMNAMM, NOA, CYCLO, WMCNAMM, AMW, AMWNAMM, MAXNESTING, CLNAMM, NOP, NOAV, ATLD, NOLV, FANOUT, FANIN, ATFD, FDP, CFNAMM, CINT, MaMCL, MeMCL, NMCS, CC, CM, NOAM, NOPA (NOAP), LAA, NOI, NMO, NODM, NOPM, NOPRM, NOPLM, NONAM, NOSM, NIM, NOIL, NODA, NOPVA, NOPRA, NOFA, NONCM, NOFM, NOFNSM, NOFSM, NONFNABM, NOFSA, NOFNSA, NONFNSA, NOSA, NONFSA, NONFNSA, NOSA, NONFSA, NOABM, NOCM, NONFNSM, NONFSM."
Tarwani et al. [21]	"LOC, C&K metrics, CC, Cyclomatic complexity (Ocavg), Dependencies (Dcy&Dcy), Javadoc function (jf), Javadoc Line of code (jLOC), Javadoc methods (jm), MHF, AHF."
Amorin et al. [22]	" C&K metrics, QMOOD metrics except (CAM, ANA, CIS), ACAIC, ACMIC, AID, AMC, CA, CAM, CE, CLD, COHESION-ATTRIBUTES, CONNEC- TIVITY, DCAEC, DCMEC, DIT1, IC, ICHCLASS, IR, LCOM1, LCOM2, LCOM3, LCOM5, LOC, LOC1, MCCABE, MLOCSUM, NAD, NADEXTENDED, NCM, NMA, NMD, NMDEXTENDED, NMI, NMO, NOA, NOC 1 , NOD, NOF, NOPARAM, NOPM, NOTC, NOTI, NPM, RFC NEW, SIX, VG-SUM, WMC NEW , WMC1, CBM."
Maiga et al. [28]	C&K metrics, QMOOD metrics, ACAIC, ACMIC, AID, CBOin, CBOout, CLD, DCAEC, DCMEC, ICHClass, IR, LCOM2, LCOM5, LOC, McCabe, NAD, NADExtended, NCM, NMA, NMD, NMDExtended, NMI, NMO, NOA, NOD, NOPM, NOParam, NOTI, NPrM, SIX, USELESS, WMC1, cohesionAttributes, connectivity.
Maneerat et al. [26]	" MOOD metrics, NA, NC, NM, NO, NP, C PARAM, RFC, WAC, WMA, D APPEAR, DIT, NOC, NAI, NOI, ACT, COMP, NS, CBC, ABSTR_R, ASSOC_R, DEPEND_R".
Kreimer [69]	"Number of statements of a method, NOLV, NOP, CYCLO, Number of instance variables of a class, Median of the number of statements of all methods of a class, Number of internal connected components, Median of complexities of all methods of a class, Number of external connected components."
Wang et al. [71]	"Whether it is a Local Clone, Method Name Similarity, Masked File Name Similarity, Sum of Parameter Similarities, Difference on Only Postx Number, Maximal Parameter Similarity, Fine Name Similarity, LOC, Number of Library Invocations Number of Other Invocations, Number of Local Invocations, Number of Invocations, Number of Library Invocations, Number of Invocations, Number of Other Invocations, Number of Field Accesses, Number of Local Invocations, Number of Recent Changes, Number of Changes, Existence Time, File Existence Time, Number of Recent File Changes, Number of File Changes"

Table (3.27) Used tools to compute the metric

Tool	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	Frequency
<i>POM</i>						✓		✓				✓						3
<i>DFMC4J</i>	✓			✓			✓											3
<i>Eclipse Plugin</i>			✓								✓							2
<i>SciTools Understand</i>		✓																1
<i>Analyst 4j</i>			✓															1
<i>IntelliJ IDEA</i>					✓													1
<i>CKJM</i>						✓												1
<i>Design Program Dependence Graph (DPDG)</i>													✓					1

3.2.6 The tools used to implement the machine learning algorithms (RQ5)

We found that two kinds of tools were used to implement the machine learning algorithms to predict code smells: WEKA and Tensor Flow implemented in Python. Table 3.28 shows the tools which were used in the PSs to implement machine learning algorithms. Eleven of the 17 studies used Weka and only one study **S2** utilized Tensor Flow written in Python. Five studies **S5**, **S11**, **S14**, **S16**, and **S17** did not provide details on the tool that was used.

Table (3.28) Tools used to implement experiments

Tool	Studies	#Studies
Weka	S1, S3, S4, S6, S7, S8, S9, S10, S12, S13, S15	10
Tensor Flow	S2	1
Not mentioned	S5, S11, S14, S16, S17	2

We observe that most of the studies used Weka because this tool is the most common and widely used for implementing machine learning algorithms [4], [26]. In addition, this tool contains 37 algorithms for classification problems [26] and is considered a reliable tool [24]. Furthermore, it is freely available.

3.3 Discussion

In this section, we discuss the main findings of the conducted SLR based on the research questions. Furthermore, we report some recommendations that can be employed as a starting point for future research.

3.3.1 Types of machine learning techniques used (RQ1)

Our findings show that 18 of the most commonly used machine learning approaches were used to predict software smells. The reasons behind using these techniques in many studies are: 1) the high performance these techniques provide, 2) and the availability of these techniques in the tools. Unfortunately, only 5 out of 17 PSs used more than one machine learning algorithms, this reveals a lack in the application of machine learning to address the issue of smells in the software. Therefore, more studies are needed in this field. Future studies most used more than one machine learning technique to explore the capability of each technique to predict code smells on the same dataset.

Ensemble learning is proposed to improve the performance of weak classifiers by combining several single classifiers using different methods. However, most of the reviewed studies applied a single classifier whereas three studies out of the seventeen used boosting-based ensemble techniques (homogenous ensemble techniques). Ensemble techniques were found to outperform single learning techniques in predicting software defects [29], thus, more studies that use ensemble techniques to predict code

smells should be conducted.

3.3.2 Predicted smells (RQ2)

Looking at our results, 28 types of code smells were predicted in the PSs. God Class, Long Method, Feature Envy, and Data Class were the most predicted smells, they also got more attention in the recent studies, as shown in Table 3.12. The possible reasons for selecting these code smells are:

- They cover different design problems.
- Their critical influence on software quality.
- They are the most frequently occurring code smells.
- These code smells are well known and easy to understand.
- The availability of these code smells in the automated tools.

In terms of using machine learning to predict code smells, we observe that most of the PSs used a small number of code smells. Furthermore, not all code smells proposed by Fowler [31] are covered by the previous studies so some of these code smells that might have an effect on the software quality have not been studied yet (for example in complete library class and Inappropriate Intimacy). Hence, more studies can be conducted to study these code smells.

3.3.3 The accuracy of machine learning techniques (RQ3)

We observed that the recall metric was the most frequently used metric followed by the accuracy, precision metrics. However, Some researchers (for instance the authors of [4]) reported that the metrics accuracy, F-measure and ROC Area show several points of views of the performances of predictive models and showed that using only one of these measures can show the whole performance measure, therefore, these metrics should also be employed to evaluate the performance of applied machine learning approaches for code smells prediction, however, recall and precision should be used together to adequately assess the performance of a model. In addition, they reported that it is not recommended to use accuracy metrics alone especially in the case of having unbalanced datasets (negative and positive classes are unbalanced). We observed that most of the used performance metrics are important measures for the classification models: recall, accuracy, precision, F1-score, AUC, and Specificity [94], [95], [96]. Only two metrics used for a regression problem which are RMSE and MAE measures. These two metrics were proposed to use for regression models [97], [98]. Furthermore, only two studies applied the Grid-search algorithm. This algorithm used to search and find the optimal parameters of machine learning algorithms that yield the highest performance. Therefore, we believe that more studies are needed to address this issue to generate more efficient code smell prediction models.

3.3.4 The used datasets (RQ4)

Based on our findings, the Java language got more attention from the previous studies. Most of the PSs tested their approaches using systems written in java. For this reason, more studies are needed here to investigate the capabilities of machine learning techniques to predict code smells in other programming languages Such as C#, C++. Moreover, we found that, 5 studies (**S1, S2, S4, S7, S6**) build their datasets from different systems, while, the other studies build their datasets based on one system. Building datasets from different systems is the best way to avoid the issue of bias. In addition, most of PSs used software metrics as the independent variables, While, recent studies [99], [100] have reported that, software metrics are insufficient to optimize software quality. Therefore, this issue can be considered as an important limitation of the PSs. To address this issue, the implementation of other resources to use it for identifying the characteristic of smelly code such as dynamic, historical, and textual aspects. In addition, the manual validation process of the dataset is an important step to have an effective dataset.

Feature selection is an essential preprocessing step in machine learning that aims to select a small subset of the relevant features from the original dataset by eliminating redundant, irrelevant or noisy features in order to achieve better learning performance, i.e., enhances accuracy, lower computational cost and time [101]. Our findings show that only one study S1 used Gain Ratio Feature Evaluation technique as feature selection techniques. Thus, more studies are needed to investigate the use of feature selection in code smells prediction. On the other hand, only two studies applied

multiclassification. Therefore, future studies should consider multiclassification.

Datasets Analysis

In this subsection, we analyze the available datasets that were used in **S4** [4], and **S8** [28]. The first dataset was built by Arcelli Fontana et al. [4] who collected systems from one of the massive curated benchmark datasets, called Qualitas Corpus [102], release 20120401r. 74 systems out of 111 Java systems of the corpus were selected. 37 systems were ignored since they could not be compiled, consequently, bad smell prediction could not be achieved. In these 74 software systems, there are 6,785,568 lines of code, 3420 packages, 51,826 classes, and 404,316 methods.

The software metrics presented in Table 3.26 were used as independent variables in their machine learning approaches for predicting class and method level smells. At the class level, there were two smells: Data Class and God Class and at the method level, there were also two smells: Feature Envy and Long Method. At the method level, 82 software metrics were computed for each system, while, at the class level, only 61 software metrics were computed. The computed operation was done using a tool developed by the authors called “DFMC4J”.

Four datasets were built by the authors of **S5**, one for every smell, i.e., every dataset included code that has smelly and non-smelly parts. Table 3.29 shows the set of automatic predictors that were employed by the authors to predict each code smell to establish the dependent variable for the bad smell prediction methods.

Generally, achieving 100% recall is unusual in this field, meaning that an automatic prediction method might not recognize real bad smell instances even when combin-

Table (3.29) Bad Smell prediction Tools

predictors	Code Smell
iPlasma	Long Method, Data Class, Feature Envy, God Class
Marinescu	Long Method
Fluid Tool	Feature Envy, Data Class
Antipattern Scanner	Data Class
PMD	Long Method, God Class

ing multiple predictors. A stratified random sampling of the methods/classes of the studied systems was utilized to enhance their confidence in validity of the dependent variable and address the false positives: this sampling performed 1,986 instances as shown in Table 3.30, the balance of the training set being $\frac{2}{3}$ negative and $\frac{1}{3}$ positive instances, where positive means the system is affected by a code smell, while negative means it is not affected. Later, a manual validation of these instances was conducted by three master students to check the prediction results.

Table (3.30) Sampling Procedure

Code Case	Instances
Smelly	826
Non-smelly	1,160
Total	1,986

The authors made a normalization of the sampled dataset: by removing the non-smelly and smelly elements and building four separate datasets using every kind of software smell, containing 280 non-smelly and 140 smelly instances, hence there were 420 instances in each dataset. The training set for the machine learning methods was represented by these four datasets.

The second dataset was built by Maiga et al. [103], [28] who analyzed three open-source systems that are freely available and have been utilized in several studies: Azureus v2.3.0.6, Xerces v2.7.0, and ArgoUML v0.19.8.

The software metrics presented in Table 3.26 were used as independent variables in their machine learning technique for predicting four code smells: God Class, Swiss Army Knife, Functional Decomposition, and Spaghetti Code. These metrics were used as training inputs. They calculated the values of the software metrics to use them as independent variables and the computed operation was done using a tool called POM. This tool is an extensible framework that depends on the PADL meta-model [104] and contains more than sixty software metrics [105].

Maiga et al. built three datasets for every system and every code smell. These datasets comprise two parts: smelly or non-smelly classes in equal numbers. They used 30 functional decomposition classes in Azureus v2.3.0.6 to build each dataset and added 30 non-functional decomposition classes by selecting them randomly in the remaining classes of Azureus v2.3.0.6. After this, they divided the 60 classes randomly into the three datasets: Dataset1, Dataset2, and Dataset3, making sure that each dataset contains 10 functional decomposition instances and 10 non-functional decomposition instances. Therefore, every dataset has 20 instances. Finally, they used practitioners' feedback on the results as input for the training dataset.

3.3.5 The tools used to implement the machine learning algorithms (RQ5)

We observe that most of the studies used Weka because this tool is the most common and widely used for implementing machine learning algorithms [4], [26]. In addition, this tool contains 37 algorithms for classification problems [26] and is considered a

reliable tool [24]. Furthermore, it is freely available. However, based on our findings we can claim that using only two tools in this field not enough since the used tools do not provide all types of machine learning techniques. This is a clear opportunity for many researchers to improve the prediction of code smells by using other application environments which are considered as in the top of ranking for implementing machine learning techniques, and also have numerous libraries which are the implementations of recent machine learning techniques, for example, Python, R, and Julia.

3.4 Threats to validity

In this section, we identify some potential threats to validity which might have biased our systematic literature review. The potential threats to the validity of our research are searching and choosing the relevant studies and the data extraction processes.

The authors in [64], [106] reported that a common threat to SLR is the process of finding all the relevant research. To find the relevant research, one threat is the process of selecting online databases. To mitigate this threat, five online research databases were selected. We selected the well-known online databases that cover all journal papers and conferences: ACM, IEEE, Scopus, Springer, Science Direct.

In order to mitigate the threat of excluding a relevant study or including an irrelevant study in the study selection phase, three researchers undertook the selection process and resolved all conflicts in the case of disagreement.

To mitigate bias in the data extraction phase, data extraction cards were constructed. Three researchers performed this process, one of them extracting the data

then inserting this into the cards, while the others examined the collected data. In the case of disagreement, a discussion was conducted to resolve these conflicts.

3.5 Chapter Summary

In this Chapter, we conducted an SLR for empirical studies on the machine learning techniques used to predict code smells. This chapter has started by presenting a general background of code smells, and machine learning techniques. To achieve the study objectives, we carried out an SLR following the general guidelines defined by Kitchenham and Charters [64]. First, a wide literature search was conducted in five online databases to identify relevant studies. Then, a set of inclusion and exclusion criteria and quality assessments were devised to obtain the primary studies. The selected studies were then analyzed, classified and compared using our defined criteria including types of machine learning algorithms, prediction accuracy, predicted code smells and datasets, resulting in the selection of 17 primary studies.

The results of this chapter provide knowledge for both practitioners and researchers about the most frequently code smells predicted and the machine learning technique used to predict them. The results also provide information about the accuracy measures used in the experimental studies that practitioners and researchers can use for comparison with existing studies. The study also provides analysis about the tools used for code smell prediction and correction. Furthermore, the details of the datasets used in bad smell prediction and correction studies are also reported.

The main results of the SLR show:

- God Class and Long Method, Feature Envy, and Data Class are the most frequently predicted code smells.
- SVM, J48, NB, RF, SMO, JRIP are the most machine learning algorithms used.
- Machine-learning techniques that consistently outperform the other machine learning techniques J48 algorithm, RF, JRip.
- Weka is the most commonly used machine learning tool while one study employed Tensor Flow in Python.
- Some code smells have not been investigated.
- The application of machine learning techniques to predict code smells is still a new area and needs further investigation.
- The application of ensemble learning approaches for code smells prediction has gained less attention.
- Only one study used feature selection techniques.

CHAPTER 4

EXPERIMENTS

In this chapter, we present our experiment. We start by describing the main hypothesis. Then, presenting the proposed model. Next, we present the experiment setup. After that, we reported the threats to validity. The results are then discussed and analyzed.

4.1 Research hypothesis

This section briefly describes the hypothesis of our empirical studies. As mentioned in Chapter 3, the main observations are that the most used machine learning approaches to predict software smells are single classifiers, and ensemble learning methods have not been used, this is from one hand. On the other hand, it is observed that pre-processing has not gained attention in case of code smells prediction. Therefore, one of the objectives, as mentioned in Chapter 1 is to address these issues.

To answer the research question related to the empirical studies defined in Chapter 1 and here, we formulate the following hypothesis.

RQ1: To which extent ensemble learning techniques outperform single classifiers on code smells prediction?

A null hypothesis of this research question can be formulated as follows:

H_0 : Applying ensemble learning has no affects on prediction bad smell comparing with single-learning techniques? In other words, there is no difference between ensemble and single leaning techniques. Accordingly the alternative hypothesis is:

H_a : Ensemble learning techniques are more accurate and efficient to predict code smells in systems.

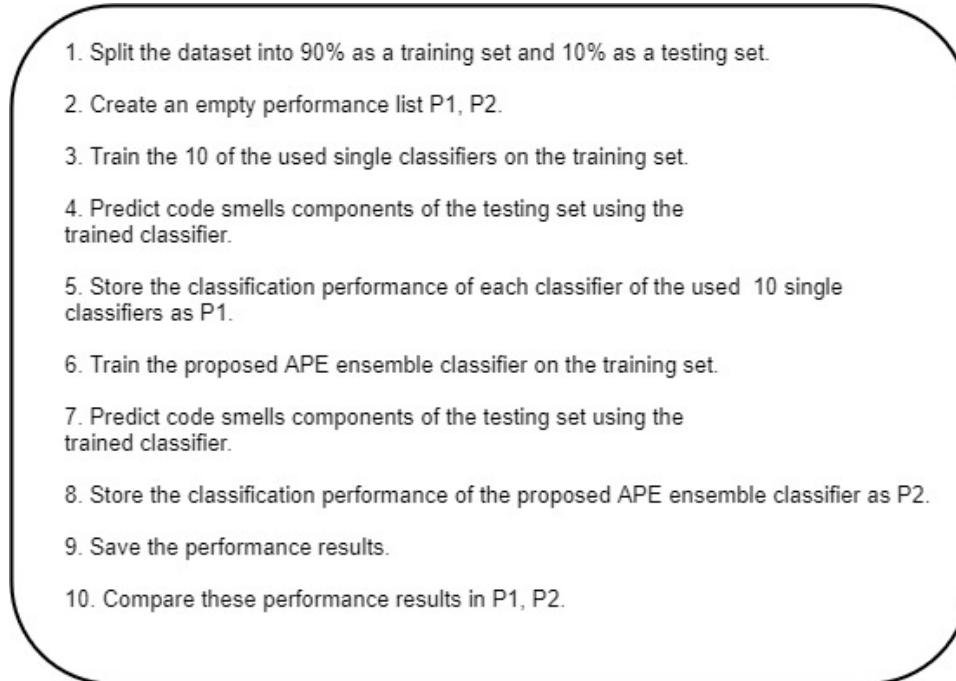


Figure (4.1) Hypothesis H_a evaluation algorithm

Figure 4.1 states the steps to evaluate hypothesis H_a . Based on the collected performance results P1 and P2, we would identify to which extent the performance results of the proposed APE model have outperformed the performance results of the single classifiers on code smells prediction. Consequently, the objective of this article

is achieved.

4.1.1 The proposed ensemble learning models

In this study, we proposed an ensemble classification model to address the issue of code smells prediction. This model composes a set of individual machine learning techniques whose classification performance are averaged to provide the final classification. The averaging process yields to reduce uncorrelated errors and, hence, improving overall classification performance [107]. In this model, we use the APE method. Ensemble learning is proposed to enhance the performance of weak classifiers through combining several single classifiers using different methods [41]. Therefore, ensemble methods are able to offer robustness against most single models' problems such as, irrelevant features, feature redundancy and data imbalance that usually hinder code smells datasets. Unlike voting methods, APE model is in conformity with the Area Under the Curve (AUC) metrics, which calculates the degree of certainty correlated with the chosen class. As well as, using voting methods need a threshold to specify decisions to chosen classes. This threshold may cause errors in the decisions made [53].

In the proposed APE, we selected the most commonly used classifiers in code smells prediction problems: RF, W-SVMs, GB, DT, SGD, MLP, MNB, LR, GPC, and BNB. All classifiers individually predict the probability that the source code belongs to the smelly class. Afterword, the averaging process on the output probabilities are performed to find the final probability estimation. Figure 4.2 shows the APE framework.

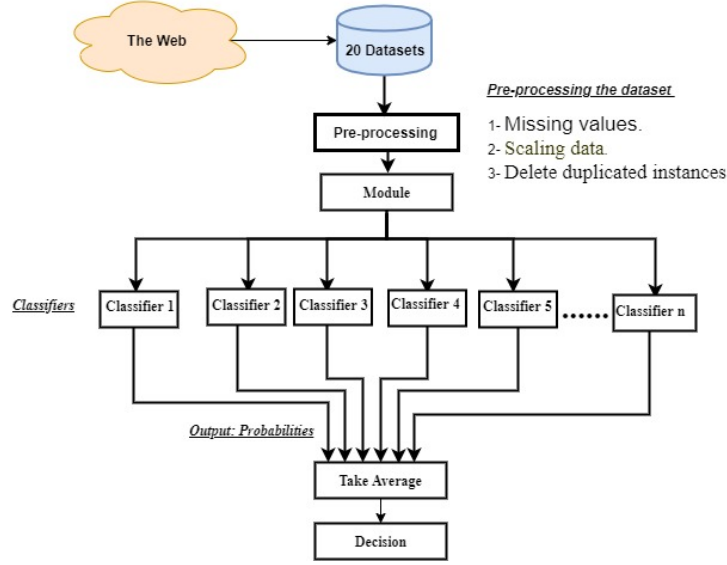


Figure (4.2) APE structure

4.2 Experiment setting

This section shows the experimental design and setup details of the conducted experiments.

4.2.1 Experimental setup and Tool

In this thesis, we performed experiments using a Windows 10(64-bit)-based computer with processor Intel(R) Core (TM) i7-4500U CPU 180GHz 2.40 GHz, and 4 GB of RAM. All experimented algorithms were performed using Scikit-Learn package on Python. We believe that, using Scikit-Learn package improves the ability to generate the proposed ensemble models. These packages tend to yield the flexibility of building such ensemble models. Consequently, we prefer using the Scikit-Learn package instead of WEKA.

4.2.2 Evaluation Measures

In this study, we applied the Area Under the Curve (AUC) measure to evaluate the classification performance. Receiver Operating Characteristics (ROC) is a common quantitative measure to evaluate classification models against imbalanced datasets [108], [109]. It is plotted true positive rate (TPR) against false positive rate (FPR) and it gets close to value 1 with the best classification, but with a bad classification yields value between 0 to 0.5 [4].

4.2.3 Evaluation method

A 10-fold cross-validation evaluation was employed to estimate the prediction performance of the proposed models and evaluate the validity of this study. The classification performance of each technique is estimated against each code smell dataset. We consider two performance factors: the performance of eight single classifiers, and the proposed ensemble models. The 10-fold cross-validation mainly employed to address the bias issue [29] specifically in the case of datasets that are time-consuming and costly to build. As well as, 10-fold cross-validation guarantees that, each fold includes the equal balances of the smelly and non-smelly classes. In this situation, the class imbalance ratio is suitably maintained.

4.2.4 Datasets Description and Pre-processing

The proposed approach was applied on 20 benchmarking datasets collected from two studies. Table 4.1 summarizes these datasets based on different defined criteria,

namely:

- Code smell: the type of predicted smells
- Number of used systems
- Smelly instances: number of smelly instances in each dataset.
- Non-smelly instances: number of non-smelly instances in each dataset
- Inputs: number of features in each dataset

Table (4.1) Used datasets

Dataset Number	Ref	Code smell	# of system	#smelly instances	#Non-smelly instances	#inputs
DS1	[4]	God Class	74	140	280	65
DS2	[4]	Data Class	74	140	280	65
DS3	[4]	Long Method	74	140	280	87
DS4	[4]	Feature Envy	74	140	280	87
DS5	[28]	Functional Decomposition	1	20	22	50
DS6	[28]	God Class	1	25	26	50
DS7	[28]	Spaghetti Code	1	36	38	50
DS8	[28]	Swiss Army Knife	1	16	17	50
DS9	[28]	Functional Decomposition	1	16	17	50
DS10	[28]	God Class	1	38	40	50
DS11	[28]	Spaghetti Code	1	154	162	50
DS12	[28]	Swiss Army Knife	1	31	33	50
DS13	[28]	Functional Decomposition	1	27	29	50
DS14	[28]	God Class	1	41	44	50
DS15	[28]	Spaghetti Code	1	71	75	50
DS16	[28]	Swiss Army Knife	1	45	56	50
DS17	[28]	Functional Decomposition	3	63	68	50
DS18	[28]	God Class	3	104	110	50
DS19	[28]	Spaghetti Code	3	261	275	50
DS20	[28]	Swiss Army Knife	3	92	106	50

Datasets from 1 to 4 were collected by [4]. These datasets were built by using 74 systems written in java language. As shown in Table 4.1 DS1 and DS2 contain class-level smells type, including God Class and Data Class. The DS3 and DS4 datasets contain method-level smells, including Long Method and Feature Envy. These datasets include the most commonly utilized software metrics including as inputs (independent variables). These metrics involve metrics proposed by Chidamber and Kemerer

in 1991 [110], [111]. The dependent variable is a binary variable designating whether a class is smelly or no.

On the other hand, datasets from 5 to 20 were constructed by [28]. These datasets were built by using three open-source systems that are freely available. They have been utilized in several studies: Azureus v2.3.0.6, Xerces v2.7.0, and ArgoUML v0.19.8. As shown also, all datasets contain class-level smells type (i.e., God class, Spaghetti Code (SC), Functional Decomposition (FD), and Swiss Army Knife (SAK)). The instances are evenly distributed between classes. These datasets include the most commonly utilized software metrics including as inputs (independent variables). These metrics involve metrics proposed by Chidamber and Kemerer, and QMOOD metrics proposed by proposed by Bansiya and Davis 2002 [112], [113]. The dependent variable is a binary variable indicating whether a class is smelly or not smelly. We built the datasets from 17 to 20 by merging datasets from 5 to 16 that have the same type of smells to investigate the ability of proposed models to predict code smells from different systems.

Datasets Pre-processing

Dataset pre-processing is an essential preparation phase in the machine learning process, it has an important effect on the performance of supervised learning techniques. The pre-processing phase includes transforming the dataset into an understandable format, remove noisy, missing data, and inconsistent data. In this study, we performed some techniques to prepare collected datasets to enhance the classification performance of the proposed models.

- **Mean imputation:** It replaces missing values using the feature's mean value computed from the available feature's values to fill in missing values on the remaining cases. This technique usually used with numeric data [114], [115].
- **Scaling data:** This technique used in case of having a difference between the maximum and minimum values, for example, if the minimum value is 0.1 and the maximum value is 1000. When scaling data is implemented the value magnitudes and scaled to appreciably low values. This is necessary for many machine learning algorithms. We applied one of the most common scaling methods: min-max normalization method with intervals of [0,1] [114], [115].
- Delete duplicated instances of each collected dataset.

4.3 Results and analysis

Twelve models are generated using single and ensemble classifiers on the considered 20 datasets. In the followings we show and analyze the obtained results.

4.3.1 Performance evaluation of single classifiers

Table 4.2 presents the obtained results in terms of AUC for each of the generated models using single classifiers on the considered datasets. The highest results are presented in bold while the lowest results are underlined.

For datasets constructed in [4], the main observations from the results of OS1_datasets can be listed and analysed as follows. (i) it was observed that all

obtained results are high and stable performance results. In these datasets, all implemented models got a classification performance results more than 91%. There are some reasons behind gotten these results such as: these datasets were built systematically, and several appropriate tools were applied to build them. In addition, these datasets are larger in terms of number of the independent variables and number instances. (ii) It also can be observed that the highest classification performance result of 99.77% was obtained using LR classifier with OS1 Datasets. Furthermore, all implemented model obtained more than 99% except BNB model. These results indicate that OS1_Long_Method is constructed well, and it is easy for models to learn it. The main observations from the obtained results for OS2_datasets are as follows. It can be observed that obtained results of OS2_datasets that contain Functional Decomposition code smell type weren't competing with other datasets. These results indicate that either this type is difficult for models to learn. It can be observed that, the worst classification result of 55% were obtained using DT model with OS2_Azureus_Function. We observed that the obtained results of OS1_datasets and OS2_datasets that contain God class code smell type are high and stable. In some cases, an AUC of 100% is obtained while the worst obtained result of this type was 86.67%.

Table 4.3 and Figure 4.3 presents the best classifier for each dataset. MNB achieves the highest performance in 6 datasets. While LR achieves the highest performance in 5 datasets three of them the datasets built in [4]. It also achieves the second highest performance for OS1_Data_Class dataset. In addition, it achieves the highest performance in other two datasets. GPC reported the highest performance in five

Table (4.2) The obtained results of the single classifiers

Dataset	SVM	DT	LR	MLP	SGD	MNB	GPC	BNB
OS1_Data_Class	98.98	97.14	99.21	<u>95.89</u>	99.52	99.13	98.6	98.9
OS1_God_Class	99.41	<u>94.64</u>	99.46	98.6	99.23	99.26	99.08	97.67
OS1_Feature_Envy	96.22	93.75	97.86	94.03	97.63	97.83	96.58	<u>91.02</u>
OS1_Long_Method	99.64	99.11	99.77	97.12	99.34	99.74	99.46	<u>93.34</u>
OS2_ArgoUML_Functiona	77.5	<u>61.67</u>	80	80	80.83	83.33	80	69.17
OS2_ArgoUML_God_Class	100	86.67	100	100	100	100	100	91.39
OS2_ArgoUML_Spaghetti	87.64	<u>80.42</u>	88.13	82.29	87.5	81.39	91.88	85.49
OS2_ArgoUML_Swiss_Arm	95	92.5	90	100	<u>87.5</u>	97.5	95	100
OS2_Azureus_Functiona	82.5	<u>55</u>	90	95	90	90	75	92.5
OS2_Azureus_God_Class	97.71	<u>92.5</u>	95.83	93.33	96.04	97.29	97.5	96.67
OS2_Azureus_Spaghetti	87.58	85.34	94.23	81.01	94.84	95.62	93.97	<u>79.19</u>
OS2_Azureus_Swiss_Arm	<u>87.5</u>	93.33	97.01	90.35	95.97	97.64	94.79	96.04
OS2_Xerces_Functional	<u>68.89</u>	74.17	76.67	71.67	75.56	76.67	77.22	86.67
OS2_Xerces_God_Class	100	<u>95.25</u>	100	98.87	99.5	100	100	98.8
OS2_Xerces_Spaghetti	91.84	90.98	95.92	91.1	97.88	96.26	94.82	<u>89.5</u>
OS2_Xerces_Swiss_Army	81.83	82	84.5	<u>74.17</u>	86.17	84.83	88	80.17
OS2_Functional-Decomp	76.31	<u>71.79</u>	80.19	73.59	81.08	81.55	76.7	77.56
OS2_God_Class	98.75	97.68	98.84	99	98.76	98.75	98.17	<u>96.59</u>
OS2_Spaghetti_Code	86.33	89.74	91.36	83.13	93.86	91.85	91.03	<u>82.57</u>
OS2_Swiss_Army_Knife	83.55	86.27	88.58	<u>81.58</u>	89.84	88.21	90.54	83.81

datasets. While SGD and MLP achieve the highest performance in four datasets. SVM got the highest performance in three datasets. BNB got the highest performance only in two datasets. Despite the DT achieves acceptable performance but it reports the lowest performance comparing with other single classifiers.

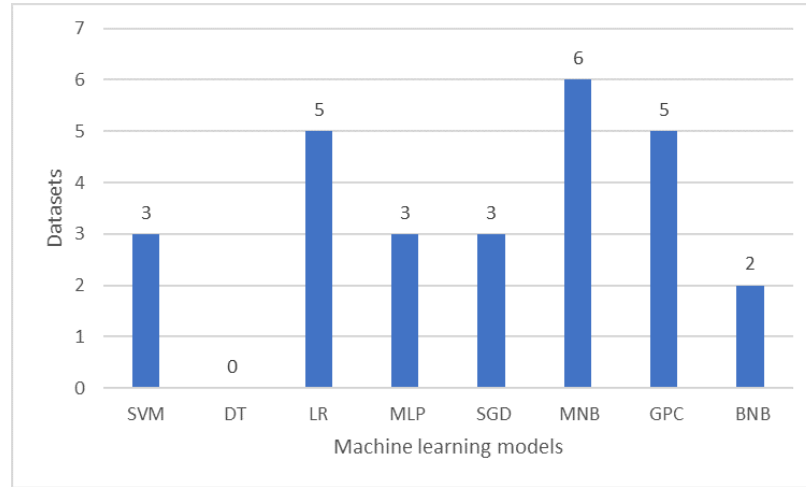


Figure (4.3) Best Classifier per dataset

Table (4.3) Best Classifier for each dataset

Dataset	Best Classifier
OS1_Data_Class	SGD
OS1_God_Class	LR
OS1_Feature_Envy	LR
OS1_Long_Method	LR
OS2_ArgoUML_Functiona	MNB
OS2_ArgoUML_God_Class	SVM, LR, MLP, SGD, MNB, GPC
OS2_ArgoUML_Spaghetti	GPC
OS2_ArgoUML_Swiss_Arm	BNB
OS2_Azureus_Functiona	MLP
OS2_Azureus_God_Class	SVM
OS2_Azureus_Spaghetti	MNB
OS2_Azureus_Swiss_Arm	MNB
OS2_Xerces_Functional	BNB
OS2_Xerces_God_Class	SVM, LR, MNB, GPC
OS2_Xerces_Spaghetti	SGD
OS2_Xerces_Swiss_Army	GPC
OS2_Functional_Decomp	MNB
OS2_God_Class	MLP
OS2_Spaghetti_Code	SGD
OS2_Swiss_Army_Knife	GPC

4.3.2 Performance evaluation of the proposed models

As we mentioned earlier the proposed models composes ten of individual machine learning techniques whose classification performance are averaged to provide the final classification. We estimate the obtained classification performance of the heterogeneous APE model against the obtained by three homogeneous ensemble learning techniques namely: GB, RF, and the proposed APE W-SVMs. APE W-SVMs model is composed of 10 classifiers of W-SVMs model (homogeneous ensemble learning). They evaluated on the twenty code smells datasets.

Table 4.4 shows the AUC results obtained by the four ensemble classifiers, the highest results are presented in bold while the lowest results are underlined. The

highest performance is obtained using APE ensemble models on most datasets with achieving an AUC measure of 100%. The other techniques achieve high results. On the other hand, the heterogeneous APE ensemble learning model outperformed APE (W-SVMs) homogeneous ensemble learning model, which definitely provided to deal with imbalanced data issues. This obviously shows that APE has been enabled to adapt its decision boundaries to fit better the minority class samples by using the averaging of decision probabilities. So, its efficient to deal with imbalanced datasets. Every classifier in the APE model attempting to recompense for the errors produced by the other classifiers. Furthermore, APE model overcomes against the irrelevant and redundant features issues. Consequently, ensembles are robust for code smell prediction.

An impressive observation is presented from the obtained performance classification results of the introduced heterogeneous APE model, which involves RF, W-SVMs, and GB as three of its base classifiers. In particularly, Using the OS2_ArgoUML_Functiona dataset, the proposed heterogeneous APE model achieved AUC of 97.5% while RF and GB achieved a low AUC of 78.33% and 72.5%, respectively. Also, using OS2_Azureus_Functiona dataset RF achieved AUC of 75%, while the proposed heterogeneous APE model achieved AUC of 90%. Moreover, using OS2_Xerces_Swiss_Army dataset APE (W-SVMs) reached AUC of 81.8%, while the proposed heterogeneous APE model achieved AUC of 98.8%. It is obvious that the averaging power of the proposed heterogeneous APE model is completely not influenced by the individual, although the poor, performance of one of its base classifiers.

The obtained classification results of our experiment are shown in Tables 4.4.

The results also show the ensemble models outperform single classifiers. Hence, we prove the Hypothesis H_a which state that heterogeneous ensemble learning classifiers outperform the single classifiers. The obtained results of homogeneous APE and heterogeneous APE are detailed in Figures in the Appendix B.2. Table B.1 summarizes a comprehensive comparison between applied models. It can be noticed that the proposed heterogeneous APE model outperforms all other used models. Consequently, the Hypothesis H_a of this study is proved.

The experimental results show that applying ensemble learning can improve the generated models to predict code smells in systems. So, the null hypothesis H_0 will be rejected and we accept the alternative hypothesis H_a .

Table (4.4) Classification results of APE, APE W-SVMs, GB and RF models.

Dataset	APE W-SVMs	APE	RF	GB
OS1_Data_Class	<u>99.6</u>	100	99.8	99.9
OS1_God_Class	99.7	99.9	<u>99.06</u>	99.7
OS1_Feature_Envy	99.1	100	<u>98.34</u>	99.2
OS1_Long_Method	100	100	100	<u>99.91</u>
OS2_ArgoUML_Functiona	95	97.5	78.33	<u>72.5</u>
OS2_ArgoUML_God_Class	100	100	100	<u>97.08</u>
OS2_ArgoUML_Spaghetti	95.6	100	92.74	<u>89.65</u>
OS2_ArgoUML_Swiss_Arm	100	100	100	<u>92.5</u>
OS2_Azureus_Functiona	90	90	75	81.25
OS2_Azureus_God_Class	100	100	<u>96.46</u>	98.75
OS2_Azureus_Spaghetti	<u>94.9</u>	97.9	95.63	96.25
OS2_Azureus_Swiss_Arm	97.6	100	<u>89.69</u>	97.22
OS2_Xerces_Functional	92.2	100	81.39	<u>79.17</u>
OS2_Xerces_God_Class.	100	100	100	100
OS2_Xerces_Spaghetti	<u>95.5</u>	100	97.35	98.93
OS2_Xerces_Swiss_Army	<u>81.8</u>	98.8	92.83	98
OS2_Functional	83.5	96.7	<u>81.7</u>	83.5
OS2_God_Class	99.8	100	<u>98.79</u>	99.3
OS2_Spaghetti	<u>91.4</u>	99.3	95.96	96.64
OS2_Swiss_Army	<u>87.6</u>	99.7	95.61	97.39

In order to demonstrate the capability of the proposed models, we have to compare

them with previous studies that were done and evaluated on the same datasets. In this regard, the only two studies made their datasets available [4], [28], so we conducted our experiments based on these datasets. To evaluate their experiments, one of them used AUC measure [4], while the other used precision and recall measures [28]. We compared the results of the proposed method with those of the previous study [4] because they used the same evaluation measure. Table 4.5 shows this comparison. When comparing our results to those of the previous study, we observe that the proposed models outperform the results obtained from the previous study.

Table (4.5) Comparison between the proposed models and the related work

Dataset	APE	APE (W-SVMs)	Related Work
OS1_Data_Class	100	99.6	99.9
OS1_God_Class	99.9	99.7	99.2
OS1_Feature_Envy	100	99.1	99
OS1_Long_Method	100	100	99.9

It can be observed that, our approach is more accurate than the literature. This can be attributed for several reasons. First of all, we applied several pre-processing phase which has not applied in the related works. Pre-processing step is a significant and important step to prepare a high data quality with significant features/attributes [116]. This defiantly will generate more accurate models. In addition, our method is based on ensemble learning. Ensemble learning is an efficient and powerful technique to improve the performance of the single classifiers [117], [118].

4.3.3 Impacts of the number of classifiers on the APE classification performance

As stated above, the proposed model is composed of 10 base classifiers. In the case of the homogeneous APE model, the model composed of a single base classifier with different parameters settings. While in the heterogeneous APE model, the model composed of different base classifiers applied on the considered 20 datasets. The impacts of the number of base classifiers in the homogeneous, heterogeneous APE models are analyzed and the results shown in Figures in the Appendix B.2. As we observed in some cases, for example in the case of OS2_ArgoUML_Spaghetti B.2c, OS2_Xerces_Swiss_Army B.4d, OS2_Swiss_Army B.5d datasets adding more base classifiers does not enhance the classification performance of the APE model. It did not enhance beyond ten classifiers i.e., a small drop in performance when using more than ten classifiers. This drop in performance obviously shows the appearance of overfitting issue which is due to base learners biased towards training data instances.

4.4 Threats to validity

we identify some potential threats to validity of our experiment. The first threat to internal validity is the selection of metrics used as independent variables. Various software metrics are represented in the literature. Most of them might be a more suitable indicator for software code smells. But we utilized metrics that were available from the used datasets. One of the most significant threats to external validity

is the size of datasets. Some used datasets have a small size (i.e., small instances) this may have an impact on the validity of classification performance. For instance, OS2_ArgoUML_Spaghetti and OS2_ArgoUML_Swiss_Arm, they have only 33 instances. To mitigate this threat, we build 4 datasets by combining datasets that have the same type of smells.

Moreover, the machine-learning models were generated using Scikit-Learn package implemented on Python which has been applied for several similar problems.

4.5 Chapter Summary

In this chapter, we carried out several experiments to address the issue of code smells prediction using advance machine learning techniques. This chapter has started with a description of the main hypothesis. Then, identifying the proposed ensemble APE models. Next, presenting the experiment setup, the experimental setup and tool, evaluation measures, evaluation method, and datasets description and pre-processing. The results are then analyzed. In section results and analysis, using 20 datasets we carried out our experiments as follows:

- Several machine learning techniques are applied to generate different models that are able to predict the code smells. Particularly, we applied eight single learning approaches and evaluated them.
- Investigated the efficiency of ensemble learning techniques to overcome the issues of code smells prediction by applying the proposed heterogeneous and homogeneous ensemble learning APE models.

- Comparing between obtained results with related work.

The experimental results show:

- These results present the robustness of our proposed models to overcome against code smells prediction issues using single classifiers.
- The heterogeneous APE models outperform other models.
- Increasing the number of base classifiers on the proposed ensemble learning APE model has a positive effect on its classification performance. The best classification performance is obtained when using APE with 10 base classifiers.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this chapter, the Conclusion, methodologies and contributions of the thesis are summarized, it also presents some suggestions for future research work are presented.

5.1 Conclusion

We have performed an SLR for empirical studies on the machine learning techniques used to predict code smells. The main objective was to systematically review and analyze the machine learning techniques applied to predict code smells from different perspectives: the code smells that were employed in the experiments, the types of machine learning techniques applied to predict code smells, a comparison between these models in terms of prediction accuracy and performance, the datasets used, and the tool utilized to implement the machine learning models.

A literature search was conducted using five online databases to retrieve the rele-

vant studies. As a result, seventeen primary studies relating to the five research questions in this study were selected. The results show that God Class and Long Method, Feature Envy, and Data Class are the most frequently predicted code smells. We also observed that machine learning algorithms SVMs, J48, NB, RF, BBNs, SMO, JRIP are used the most. We also observed that the standalone machine learning algorithms that consistently outperform the other standalone machine learning algorithms are the J48 algorithm, RF, JRip, NB, Instance-Bases Learning with Parameter k (IBK), Instance-based learning (IBL), SVM, SMO, and BLR. The algorithm with the worst performance is VFI.

Our SLR also indicated that the family of J48 techniques and B-Random Forest are the top techniques for predicting all types of code smells while the SVM techniques are the worst. As well, we noticed that, in some cases, the use of boosting techniques on the models does not always enhance their performance. With respect to the dataset, it was found that all the studies used Java-based case studies and most of them used software metrics as independent variables. Only two of the 13 studies made the dataset available. Finally, we found that Weka is the most commonly used machine learning tool while one study employed Tensor Flow in Python.

As a result of this study, we make the following observations to be considered when conducting new research that use machine learning for bad smell prediction: 1) B-J48 unpruned achieved the highest performance results in all PSs, it got 99.63% of F-Measure value when it was applied for Long method prediction, followed by B-J48 pruned which achieved 99.26% of F-Measure value when it was applied for Data Class prediction, 2) some code smells have not been investigate, 3) the use of ensemble

learning techniques for code smells prediction has gained less attention as it was used in 2 studies only, 4) only 2 studies used the Grid-search algorithm to configure the machine learning algorithms parameters to find optimal parameters, 5) four studies built their datasets from different systems, 6) only two study used multiclassification, and 7) only one study used feature selection technique.

It can be seen that the application of machine learning techniques to predict code smells is still a new area and needs further investigation. Therefore, more research effort should be focused to facilitate the employment of machine learning techniques to address the issue of predicting code smells. The outcomes of this SLR are submitted in Arabian Journal for Science and Engineering journal.

Based on the observations obtained from the conducted SLR which motioned that, the most used machine learning techniques to predict code smells are single classifiers, and ensemble learning methods have not been widely utilized. Therefore, we performed an empirical study to address this lack.

Our empirical study investigated the extent to which ensemble learning techniques outperforms single classifiers on code smells prediction. We introduce an ensemble learning APE model to overcome the problem of code smells prediction. This algorithm implemented in two-way (i.e., homogeneous ensemble and heterogeneous ensemble).

To analyze the efficiency of the proposed models, we evaluated it on 20 benchmarking datasets. starting by applying Eight single learning approaches to generate different models that are able to predict the code smells. Next, Investigating the efficiency of the proposed ensemble learning APE models to address the issues of code

smells prediction. The applied models are evaluated using AUC metrics. A comparison between all applied models with related work is then reported. Finally, we investigated the effect of number of base classifiers on the ensemble learning APE classification performance.

Our experiments findings report that, in most cases, the proposed technique APE outperformed other single classifiers. It achieved AUC values of 100% on most used datasets. Finally, the application of proposed ensemble model APE has effects to enhance the classification performance of code smells prediction.

5.2 Summary of the contributions

In our thesis we performed the following contributions,

- We have conducted an SLR to find the gaps in the field of code smells prediction using machine learning techniques.
- We have performed a comprehensive study for single classifiers in code smells prediction.
- We have Investigated the efficiency of ensemble learning techniques to overcome the issues of code smells prediction by applying the proposed heterogeneous and homogeneous ensemble learning APE models.

5.3 Future Works

There are several issues that researchers of code smell prediction need to address. The promising studies that can be conducted are as following:

- Future studies should consider multi classification, and to which extent machine learning techniques can address it. Hence, we need to build a dataset that contains more than two classes.
- As we mentioned the available datasets were built based on a collection of projects written in Java. So, we intend to build datasets from different program languages such as Python and C#, and then investigate the ability of machine learning methods to address code smells issues these program languages.

REFERENCES

- [1] P. Danphitsanuphan and T. Suwantada, “Code smell detecting tool and code smell-structure bug relationship,” in *Engineering and Technology (S-CET), 2012 Spring Congress on.* IEEE, 2012, pp. 1–5.
- [2] A. Güzel and Ö. AKTAS, “A survey on bad smells in codes and usage of algorithm analysis,” *International Journal of Computer Science and Software Engineering (IJCSSE)*, vol. 5, no. 6, 2016.
- [3] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, “A bayesian approach for the detection of code and design smells,” in *Quality Software, 2009. QSIC’09. 9th International Conference on.* IEEE, 2009, pp. 305–314.
- [4] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, “Comparing and experimenting machine learning techniques for code smell detection,” *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.
- [5] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, “Software complexity and maintenance costs,” *Communications of the ACM*, vol. 36, no. 11, pp. 81–95, 1993.

- [6] A. Yamashita and L. Moonen, “Do code smells reflect important maintainability aspects?” in *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE, 2012, pp. 306–315.
- [7] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, “Quantifying the effect of code smells on maintenance effort,” *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2012.
- [8] R. Roy, R. Stark, K. Tracht, S. Takata, and M. Mori, “Continuous maintenance and the future—foundations and technological challenges,” *CIRP Annals*, vol. 65, no. 2, pp. 667–688, 2016.
- [9] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. do Nascimento, M. F. Freitas, and M. G. de Mendonça, “A systematic review on the code smell effect,” *Journal of Systems and Software*, vol. 144, pp. 450–477, 2018.
- [10] P. Kruchten, R. L. Nord, and I. Ozkaya, “Technical debt: From metaphor to theory and practice,” *Ieee software*, vol. 29, no. 6, pp. 18–21, 2012.
- [11] C. Seaman and Y. Guo, “Measuring and monitoring technical debt,” in *Advances in Computers*. Elsevier, 2011, vol. 82, pp. 25–46.
- [12] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, “Method-book: Recommending move method refactorings via relational topic models,” *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, 2013.

- [13] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 350–359.
- [14] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, “Decor: A method for the specification and detection of code and design smells,” *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.
- [15] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, “Mining version histories for detecting code smells,” *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2014.
- [16] F. Palomba, D. A. A. Tamburri, F. A. Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, “Beyond technical aspects: How do community smells influence the intensity of code smells?” *IEEE transactions on software engineering*, 2018.
- [17] F. Palomba, A. Zaidman, and A. De Lucia, “Automatic test smell detection using information retrieval techniques,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 311–322.
- [18] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.
- [19] M. J. Munro, “Product metrics for automatic identification of” bad smell” design problems in java source-code,” in *11th IEEE International Software Metrics Symposium (METRICS’05)*. IEEE, 2005, pp. 15–15.

- [20] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, “On the use of developers’ context for automatic refactoring of software anti-patterns,” *Journal of Systems and Software*, vol. 128, pp. 236–251, 2017.
- [21] S. Tarwani and A. Chug, “Predicting maintainability of open source software using gene expression programming and bad smells,” in *Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO), 2016 5th International Conference on.* IEEE, 2016, pp. 452–459.
- [22] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro, “Experience report: Evaluating the effectiveness of decision trees for detecting code smells,” in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on.* IEEE, 2015, pp. 261–269.
- [23] D. K. Kim, “Finding bad code smells with neural network models,” *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 7, no. 6, pp. 3613–3621, 2017.
- [24] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, “Detecting code smells using machine learning techniques: are we there yet?” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2018, pp. 612–621.
- [25] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla, “Code smell detection: Towards a machine learning-based approach,” in *Software Maintenance*

- (*ICSM*), *2013 29th IEEE International Conference on*. IEEE, 2013, pp. 396–399.
- [26] N. Maneerat and P. Muenchaisri, “Bad-smell prediction from software design model using machine learning techniques,” in *Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on*. IEEE, 2011, pp. 331–336.
- [27] A. Kaur, S. Jain, and S. Goel, “A support vector machine based approach for code smell detection,” in *Machine Learning and Data Science (MLDS), 2017 International Conference on*. IEEE, 2017, pp. 9–14.
- [28] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, and E. Aimeur, “Smurf: A svm-based incremental anti-pattern detection approach,” in *Reverse engineering (WCRE), 2012 19th working conference on*. IEEE, 2012, pp. 466–475.
- [29] I. H. Laradji, M. Alshayeb, and L. Ghouti, “Software defect prediction using ensemble learning on selected features,” *Information and Software Technology*, vol. 58, pp. 388–402, 2015.
- [30] F. A. Fontana, P. Braione, and M. Zanoni, “Automatic detection of bad smells in code: An experimental assessment.” *Journal of Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.
- [31] K. Beck, M. Fowler, and G. Beck, “Bad smells in code,” *Refactoring: Improving the design of existing code*, pp. 75–88, 1999.

- [32] G. Rasool and Z. Arshad, “A review of code smell mining techniques,” *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, 2015.
- [33] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, “When and why your code starts to smell bad,” in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 403–414.
- [34] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, “A co-operative parallel search-based software engineering approach for code-smells detection,” *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.
- [35] R. Marticorena, C. López, and Y. Crespo, “Extending a taxonomy of bad code smells with metrics,” in *7th ECCOP International Workshop on Object-Oriented Reengineering (WOOR)*. Citeseer, 2006, p. 6.
- [36] “Design patterns and refactoring.” [Online]. Available: <https://sourcemaking.com/refactoring>
- [37] A. L. Samuel, “Some studies in machine learning using the game of checkers,” *IBM Journal of research and development*, vol. 3, no. 3, pp. 210–229, 1959.
- [38] F. Rosenblatt, “Principles of neurodynamics. perceptrons and the theory of brain mechanisms,” CORNELL AERONAUTICAL LAB INC BUFFALO NY, Tech. Rep., 1961.

- [39] D. W. Ruck, S. K. Rogers, M. Kabrisky, M. E. Oxley, and B. W. Suter, "The multilayer perceptron as an approximation to a bayes optimal discriminant function," *IEEE Transactions on Neural Networks*, vol. 1, no. 4, pp. 296–298, 1990.
- [40] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.
- [41] H. I. Aljamaan and M. O. Elish, "An empirical study of bagging and boosting ensembles for identifying faulty classes in object-oriented software," in *Computational Intelligence and Data Mining, 2009. CIDM'09. IEEE Symposium on*. IEEE, 2009, pp. 187–194.
- [42] N. Vapnik Vladimir, "The nature of statistical learning theory," 1995.
- [43] D. S. Broomhead and D. Lowe, "Radial basis functions, multi-variable functional interpolation and adaptive networks," Royal Signals and Radar Establishment Malvern (United Kingdom), Tech. Rep., 1988.
- [44] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers," *Machine learning*, vol. 29, no. 2-3, pp. 131–163, 1997.
- [45] E. Castillo, J. M. Gutierrez, and A. S. Hadi, "Expert systems and probabilistic network models," 1996.
- [46] I. Rish *et al.*, "An empirical study of the naive bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, no. 22. IBM New York, 2001, pp. 41–46.

- [47] Y. Yuan and M. J. Shaw, “Induction of fuzzy decision trees,” *Fuzzy Sets and systems*, vol. 69, no. 2, pp. 125–139, 1995.
- [48] J. R. Quinlan, “C4. 5: Programming for machine learning,” *Morgan Kauffmann*, vol. 38, p. 48, 1993.
- [49] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
- [50] Y. Zhang and A. Haghani, “A gradient boosting method to improve travel time prediction,” *Transportation Research Part C: Emerging Technologies*, vol. 58, pp. 308–324, 2015.
- [51] J. H. Friedman, “Stochastic gradient boosting,” *Computational statistics & data analysis*, vol. 38, no. 4, pp. 367–378, 2002.
- [52] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” in *Proceedings of COMPSTAT’2010*. Springer, 2010, pp. 177–186.
- [53] E. Stork, R. Duda, P. Hart, and D. Stork, “Pattern classification,” *New York [ua]: Academic Internet Publishers*, 2006.
- [54] M. Kuss and C. E. Rasmussen, “Assessing approximate inference for binary gaussian process classification,” *Journal of machine learning research*, vol. 6, no. Oct, pp. 1679–1704, 2005.
- [55] M. Zhang, T. Hall, and N. Baddoo, “Code bad smells: a review of current knowledge,” *Journal of Software: Evolution and Process*, vol. 23, no. 3, pp. 179–202, 2011.

- [56] S. Singh and S. Kaur, “A systematic literature review: Refactoring for disclosing code smells in object oriented software,” *Ain Shams Engineering Journal*, 2017.
- [57] T. Sharma and D. Spinellis, “A survey on software smells,” *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.
- [58] T. Mariani and S. R. Vergilio, “A systematic review on search-based refactoring,” *Information and Software Technology*, vol. 83, pp. 14–34, 2017.
- [59] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, “A review-based comparative study of bad smell detection tools,” in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2016, p. 18.
- [60] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, “Identifying architectural bad smells,” in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 255–258.
- [61] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis,” *Information and Software Technology*, 2019.
- [62] F. L. Caram, B. R. D. O. Rodrigues, A. S. Campanelli, and F. S. Parreiras, “Machine learning techniques for code smells detection: A systematic mapping study,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 02, pp. 285–316, 2019.

- [63] J. Wen, S. Li, Z. Lin, Y. Hu, and C. Huang, “Systematic literature review of machine learning based software development effort estimation models,” *Information and Software Technology*, vol. 54, no. 1, pp. 41–59, 2012.
- [64] B. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering,” 2007.
- [65] 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, no. 3, pp. 845–855, 2015.
- [66] O. Dieste and A. G. Padua, “Developing search strategies for detecting relevant experiments for systematic reviews,” in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 2007, pp. 215–224.
- [67] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, “Bdtex: A gqm-based bayesian approach for the detection of antipatterns,” *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.
- [68] S. Bryton, F. B. e Abreu, and M. Monteiro, “Reducing subjectivity in code smells detection: Experimenting with the long method,” in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. IEEE, 2010, pp. 337–342.
- [69] J. Kreimer, “Adaptive detection of design flaws,” *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 117–136, 2005.

- [70] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, “Classification model for code clones based on machine learning,” *Empirical Software Engineering*, vol. 20, no. 4, pp. 1095–1125, 2015.
- [71] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, “Can i clone this piece of code here?” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 170–179.
- [72] S. Hassaine, F. Khomh, Y.-G. Guéhéneuc, and S. Hamel, “Ids: An immune-inspired approach for the detection of software design smells,” in *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE, 2010, pp. 343–348.
- [73] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, “Tracking design smells: Lessons from a study of god classes,” in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 145–154.
- [74] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
- [75] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, “When and why your code starts to smell bad (and whether the smells go away),” *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.
- [76] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, “An empirical investigation into the nature of test smells,”

- in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 4–15.
- [77] R. Peters and A. Zaidman, “Evaluating the lifespan of code smells using software repository mining,” in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012, pp. 411–416.
- [78] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, “The evolution and impact of code smells: A case study of two open source systems,” in *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*. IEEE Computer Society, 2009, pp. 390–400.
- [79] D. Taibi, A. Janes, and V. Lenarduzzi, “How developers perceive smells in source code: A replicated study,” *Information and Software Technology*, vol. 92, pp. 223–235, 2017.
- [80] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, “An exploratory study on the relationship between changes and refactoring,” in *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE, 2017, pp. 176–185.
- [81] F. Palomba and A. Zaidman, “Does refactoring of test smells induce fixing flaky tests?” in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1–12.

- [82] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, “On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation,” *Empirical Software Engineering*, pp. 1–34, 2017.
- [83] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, “Do they really smell bad? a study on developers’ perception of bad code smells,” in *Software maintenance and evolution (ICSME), 2014 IEEE international conference on*. IEEE, 2014, pp. 101–110.
- [84] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, “Mining version histories for detecting code smells,” *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.
- [85] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, “Decor: A method for the specification and detection of code and design smells,” *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.
- [86] D. Rapu, S. Ducasse, T. Gîrba, and R. Marinescu, “Using history information to improve design flaws detection,” in *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*. IEEE, 2004, pp. 223–232.
- [87] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, “Code-smell detection as a bilevel problem,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 1, p. 6, 2014.

- [88] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, “A textual-based technique for smell detection,” in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.
- [89] A. Ozgur, “Supervised and unsupervised machine learning techniques for text document categorization,” *Unpublished Master’s Thesis, Istanbul: Boğaziçi University*, 2004.
- [90] P. Chaovalit and L. Zhou, “Movie review mining: A comparison between supervised and unsupervised classification approaches,” in *Proceedings of the 38th annual Hawaii international conference on system sciences*. IEEE, 2005, pp. 112c–112c.
- [91] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, “Supervised machine learning: A review of classification techniques,” *Emerging artificial intelligence applications in computer engineering*, vol. 160, pp. 3–24, 2007.
- [92] S. B. Kotsiantis, I. D. Zaharakis, and P. E. Pintelas, “Machine learning: a review of classification and combining techniques,” *Artificial Intelligence Review*, vol. 26, no. 3, pp. 159–190, 2006.
- [93] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, “Code quality analysis in open source software development,” *Information Systems Journal*, vol. 12, no. 1, pp. 43–60, 2002.

- [94] N. Seliya, T. M. Khoshgoftaar, and J. Van Hulse, “A study on the relationships of classifier performance metrics,” in *2009 21st IEEE international conference on tools with artificial intelligence*. IEEE, 2009, pp. 59–66.
- [95] L. Cuadros-Rodríguez, E. Pérez-Castaño, and C. Ruiz-Samblás, “Quality performance metrics in multivariate classification methods for qualitative analysis,” *TrAC Trends in Analytical Chemistry*, vol. 80, pp. 612–624, 2016.
- [96] H. Wang, T. M. Khoshgoftaar, and N. Seliya, “How many software metrics should be selected for defect prediction?” in *Twenty-Fourth International FLAIRS Conference*, 2011.
- [97] S. Minaee, “20 popular machine learning metrics. part 1: Classification & regression evaluation metrics,” Oct 2019. [Online]. Available: <https://towardsdatascience.com/20-popular-machine-learning-metrics-part-1-classification-regression-evaluation-metrics-1ca3e282a2>
- [98] T. Hastie, R. Tibshirani, and J. Friedman, “The elements of statistical learning. springer series in statistics,” in $\therefore$ Springer, 2001.
- [99] I. Candela, G. Bavota, B. Russo, and R. Oliveto, “Using cohesion and coupling for software remodularization: Is it enough?” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 24, 2016.
- [100] C. Simons, J. Singer, and D. R. White, “Search-based refactoring: Metrics are not enough,” in *International Symposium on Search Based Software Engineering*. Springer, 2015, pp. 47–61.

- [101] G. Chandrashekar and F. Sahin, “A survey on feature selection methods,” *Computers & Electrical Engineering*, vol. 40, no. 1, pp. 16–28, 2014.
- [102] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, “The qualitas corpus: A curated collection of java code for empirical studies,” in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. IEEE, 2010, pp. 336–345.
- [103] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, “Support vector machines for anti-pattern detection,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 278–281.
- [104] Y.-G. Guéhéneuc and G. Antoniol, “Demima: A multilayered approach for design pattern identification,” *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 667–684, 2008.
- [105] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi, “Fingerprinting design patterns,” in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, 2004, pp. 172–181.
- [106] A. Nguyen-Duc, D. S. Cruzes, and R. Conradi, “The impact of global dispersion on coordination, team performance and software quality—a systematic literature review,” *Information and Software Technology*, vol. 57, pp. 277–294, 2015.
- [107] S. Amari *et al.*, *The handbook of brain theory and neural networks*. MIT press, 2003.

- [108] Z. Sun, Q. Song, and X. Zhu, “Using coding-based ensemble learning to improve software defect prediction,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 6, pp. 1806–1817, 2012.
- [109] J.-H. Xue and P. Hall, “Why does rebalancing class-unbalanced data improve auc for linear discriminant analysis?” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 5, pp. 1109–1112, 2015.
- [110] S. R. Chidamber and C. F. Kemerer, *Towards a metrics suite for object oriented design*. ACM, 1991, vol. 26, no. 11.
- [111] —, “A metrics suite for object oriented design,” *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.
- [112] P. K. Goyal and G. Joshi, “Qmood metric sets to assess quality of java program,” in *Issues and Challenges in Intelligent Computing Techniques (ICICT), 2014 International Conference on*. IEEE, 2014, pp. 520–533.
- [113] J. Bansiya and C. G. Davis, “A hierarchical model for object-oriented design quality assessment,” *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.
- [114] S. Kotsiantis, D. Kanellopoulos, and P. Pintelas, “Data preprocessing for supervised leaning,” *International Journal of Computer Science*, vol. 1, no. 2, pp. 111–117, 2006.

- [115] J. Huang, Y.-F. Li, and M. Xie, “An empirical analysis of data preprocessing for machine learning-based software cost estimation,” *Information and software Technology*, vol. 67, pp. 108–127, 2015.
- [116] H. A. Al-Jamimi, S. Al-Azani, and T. A. Saleh, “Supervised machine learning techniques in the desulfurization of oil products for environmental protection: a review,” *Process Safety and Environmental Protection*, vol. 120, pp. 57–71, 2018.
- [117] T. G. Dietterich, “Ensemble methods in machine learning,” in *International workshop on multiple classifier systems*. Springer, 2000, pp. 1–15.
- [118] S. Al-Azani and E.-S. M. El-Alfy, “Using word embedding and ensemble learning for highly imbalanced data sentiment analysis in short arabic text,” *Procedia Computer Science*, vol. 109, pp. 359–366, 2017.

APPENDIX A

SLR APPENDIX

Table A.1

Table (A.1) Used systems

System	Studies	Frequency
Xercesv	S1, S3, S4, S7, S8, S9, S12, S16, S17	9
ArgoUML	S1, S3, S4, S6, S7, S8, S12, S16	7
WEKA	S1, S4, S7, S13	4
junit-4.10	S1,S2, S4, S7	4
aoi-2.8.1	S1, S4, S7	3
axion-1.0-M2	S1, S4, S7	3
castor-1.3.1	S1, S4, S7	3
cobertura-1.9.4.1	S1, S4, S7	3
colt-1.2.0	S1, S4, S7	3
columba-1.0	S1, S4, S7	3
displaytag-1.2	S1, S4, S7	3
drawswf-1.2.9	S1, S4, S7	3

Table A.1 continued

System	Studies	Frequency
drjava-s	S1, S4, S7	3
emma-2.0.5312	S1, S4, S7	3
	S1, S4, S7	3
findbugs-1.3.9	S1, S4, S7	3
	S1, S4, S7	3
fitlibraryforfitness-20100806	S1, S4, S7	3
freecol-0.10.3	S1, S4, S7	3
freecs-1.3.20100406	S1, S4, S7	3
freemind-0.9.0	S1, S4, S7	3
galleon-2.3.0	S1, S4, S7	3
ganttproject-2.0.9	S1, S4, S7	3
heritrix-1.14.4	S1, S4, S7	3
hsqldb-2.0.0	S1, S4, S7	3
itext-5.0.3	S1, S4, S7	3
jag-6.1	S1, S4, S7	3
jasml-0.10	S1, S4, S7	3
jasperreports-3.7.3	S1, S4, S7	3
javacc-5.0	S1, S4, S7	3
jedit-4.3.2	S1, S4, S7	3
jena-2.6.3	S1, S4, S7	3
jext-5.0	S1, S4, S7	3

Table A.1 continued

System	Studies	Frequency
jFin_DateMath-R1.0.1	S1, S4, S7	3
jfreechart-1.0.13	S1, S4, S7	3
jgraph-5.13.0.0	S1, S4, S7	3
jgraphpad-5.10.0.2	S1, S4, S7	3
jgrapht-0.8.1	S1, S4, S7	3
jgroups-2.10.0	S1, S4, S7	3
jhotdraw-7.5.1	S1, S4, S7	3
jmeter-2.5.1	S1, S4, S7	3
jmoney-0.4.4	S1, S4, S7	3
jparse-0.96	S1, S4, S7	3
jpf-1.0.2	S1, S4, S7	3
jruby-1.5.2	S1, S4, S7	3
jspwiki-2.8.4	S1, S4, S7	3
jsXe-04_beta	S1, S4, S7	3
jung-2.0.1	S1, S4, S7	3
log4j-1.2.16	S1, S4, S7	3
lucene-3.5.0	S1, S4, S7	3
marauroa-3.8.1	S1, S4, S7	3
megamek-0.35.18	S1, S4, S7	3
mvnforum-1.2.2-ga	S1, S4, S7	3
nekohtml-1.9.14	S1, S4, S7	3

Table A.1 continued

System	Studies	Frequency
openjms-0.7.7-beta-1	S1, S4, S7	3
oscache-2.4.1	S1, S4, S7	3
picocontainer-2.10.2	S1, S4, S7	3
pmd-4.2.5	S1, S4, S7	3
poi-3.6	S1, S4, S7	3
pooka-3.0-080505	S1, S4, S7	3
	S1, S4, S7	3
quartz-1.8.3	S1, S4, S7	3
quickserver-1.4.7	S1, S4, S7	3
quilt-0.6-a-5	S1, S4, S7	3
roller-4.0.1	S1, S4, S7	3
squirrel_sql-3.1.2	S1, S4, S7	3
sunflow-0.07.2	S1, S4, S7	3
tomcat-7.0.2	S1, S4, S7	3
trove-2.1.0	S1, S4, S7	3
velocity-1.6.4	S1, S4, S7	3
	S1, S4, S7	3
webmail-0.7.10	S1, S4, S7	3
xalan-2.7.1	S1, S4, S7	3
xmojo-5.0.0	S1, S4, S7	3
GanttProject v1.10.2	S9, S12	2

Table A.1 continued

System	Studies	Frequency
Azureus	S8, S16	2
Eclipse 3.3.1	S6, S17	2
Apache Commons	S11	1
Mylyn 3.1.1	S6	1
Rhino 1.6	S6	1
Jtds-1.2.8	S5	1
Jchess-1.3.1	S5	1
ArtOfIllusion - 302	S5	1
OrDrumbox - 0.9.81	S5	1
Android-Universal-Image-Loader	S2	1
bigbluebutton	S2	1
	S2	1
clojure	S2	1
dropwizard	S2	1
elasticsearch	S2	1
libgdx	S2	1
metrics	S2	1
netty	S2	1
nokogiri	S2	1
okhttp	S2	1
platform_frameworks_base	S2	1

Table A.1 continued

System	Studies	Frequency
presto	S2	1
retrofit	S2	1
rxjava	S2	1
spring-boot	S2	1
spring-framework	S2	1
storm	S2	1
zxing	S2	1
IYC	S13	1
XProj	S15	1
YProj	S2	1

APPENDIX B

EXPERIMENT APPENDIX

B.1 Comparison between applied models

Table (B.1) Comparison between applied models

Dataset	APE (W-SVMs)	APE	RF	GB	SVM	DT	LR	MLP	SGD	MNB	GPC	BNB
OS1_Data_Class	99.6	100	99.8	99.9	98.98	97.14	99.21	95.89	99.52	99.13	98.6	98.9
OS1_God_Class	99.7	99.9	99.06	99.7	99.41	94.64	99.46	98.6	99.23	99.26	99.08	97.67
OS1_Feature_Envy	99.1	100	98.34	99.2	96.22	93.75	97.86	94.03	97.63	97.83	96.58	91.02
OS1_Long_Method	100	100	100	99.91	99.64	99.11	99.77	97.12	99.34	99.74	99.46	93.34
OS2_ArgoUML_Functiona	95	97.5	78.33	72.5	77.5	61.67	80	80	80.83	83.33	80	69.17
OS2_ArgoUML_God_Class	100	100	100	97.08	100	86.67	100	100	100	100	100	91.39
OS2_ArgoUML_Spaghetti	95.6	100	92.74	89.65	87.64	80.42	88.13	82.29	87.5	81.39	91.88	85.49
OS2_ArgoUML_Swiss_Arm	100	100	100	92.5	95	92.5	90	100	87.5	97.5	95	100
OS2_Azureus_Functiona	90	90	75	81.25	82.5	55	90	95	90	90	75	92.5
OS2_Azureus_God_Class	100	100	96.46	98.75	97.71	92.5	95.83	93.33	96.04	97.29	97.5	96.67
OS2_Azureus_Spaghetti	94.9	97.9	95.63	96.25	87.58	85.34	94.23	81.01	94.84	95.62	93.97	79.19
OS2_Azureus_Swiss_Arm	97.6	100	89.69	97.22	87.5	93.33	97.01	90.35	95.97	97.64	94.79	96.04
OS2_Xerces_Functional	92.2	100	81.39	79.17	68.89	74.17	76.67	71.67	75.56	76.67	77.22	86.67
OS2_Xerces_God_Class.	100	100	100	100	100	95.25	100	98.87	99.5	100	100	98.8
OS2_Xerces_Spaghetti_	95.5	100	97.35	98.93	91.84	90.98	95.92	91.1	97.88	96.26	94.82	89.5
OS2_Xerces_Swiss_Army	81.8	98.8	92.83	98	81.83	82	84.5	74.17	86.17	84.83	88	80.17
OS2_Functional	83.5	96.7	81.7	83.5	76.31	71.79	80.19	73.59	81.08	81.55	76.7	77.56
OS2_God_Class	99.8	100	98.79	99.3	98.75	97.68	98.84	99	98.76	98.75	98.17	96.59
OS2_Spaghetti_	91.4	99.3	95.96	96.64	86.33	89.74	91.36	83.13	93.86	91.85	91.03	82.57
OS2_Swiss_Army	87.6	99.7	95.61	97.39	83.55	86.27	88.58	81.58	89.84	88.21	90.54	83.81

B.2 Normal APE models Results

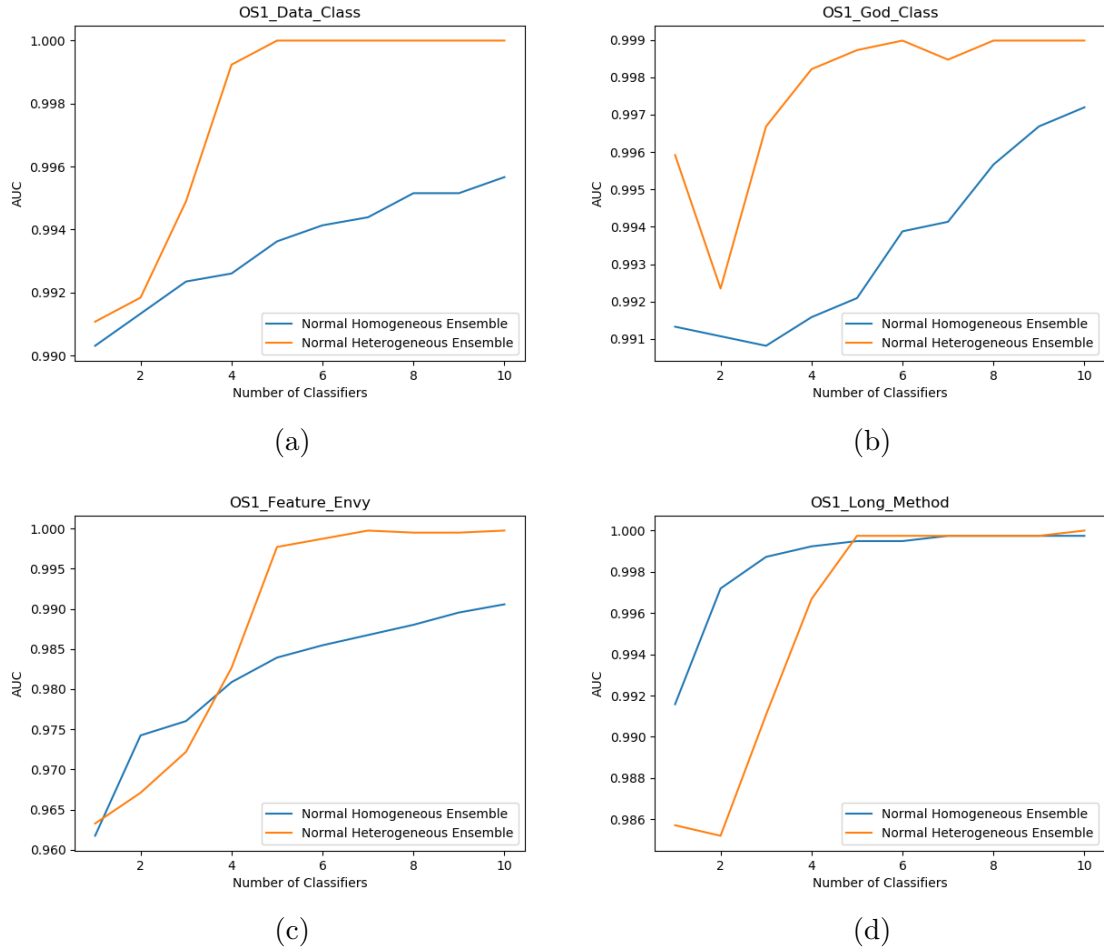
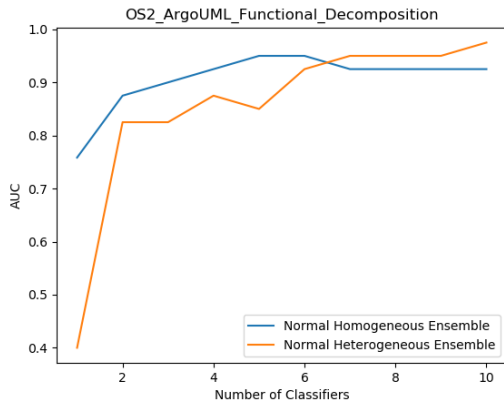
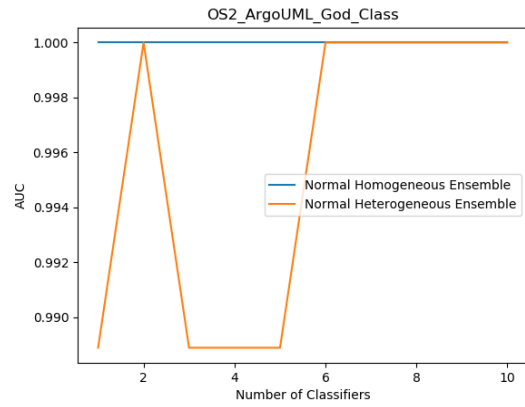


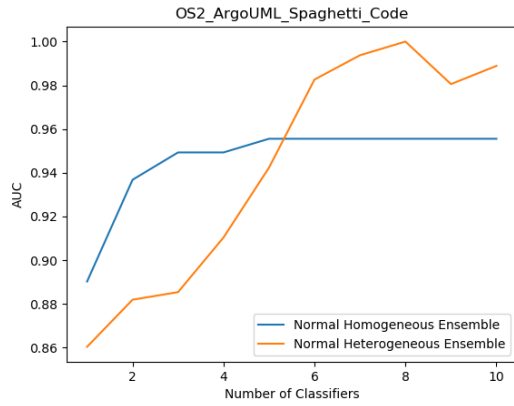
Figure (B.1) Performance evaluation of the normal APE models



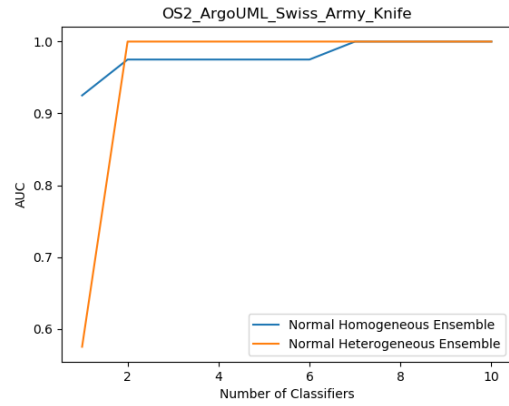
(a)



(b)

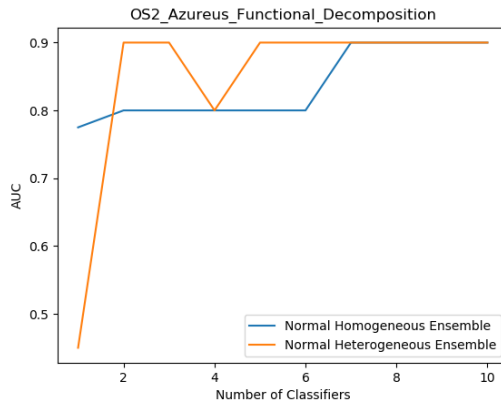


(c)

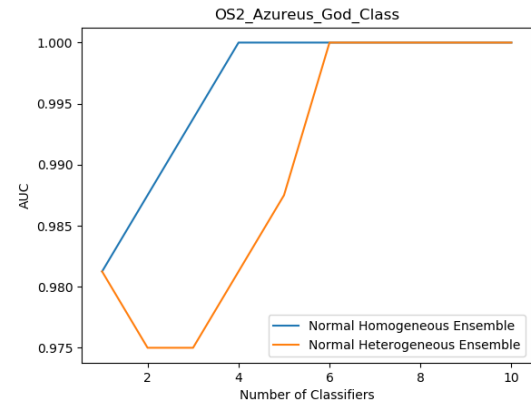


(d)

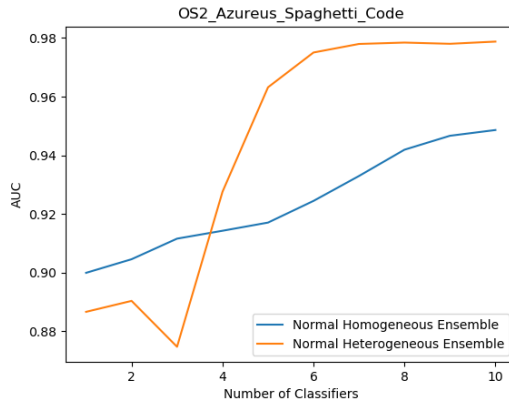
Figure (B.2) Performance evaluation of the normal APE models continued



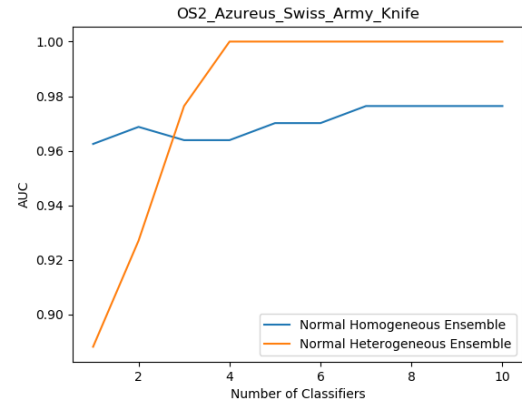
(a)



(b)

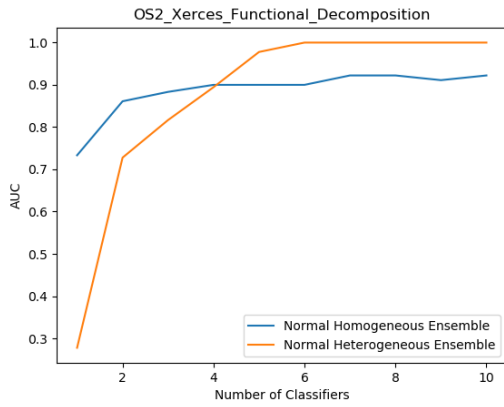


(c)



(d)

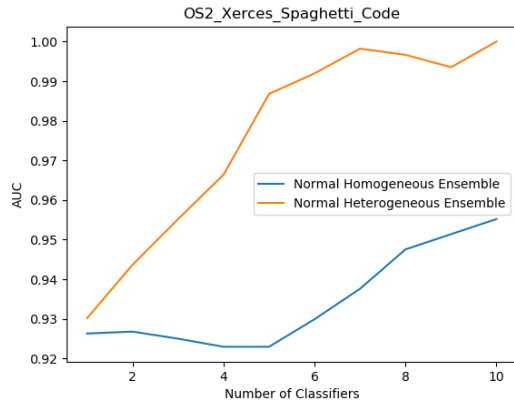
Figure (B.3) Performance evaluation of the normal APE models continued



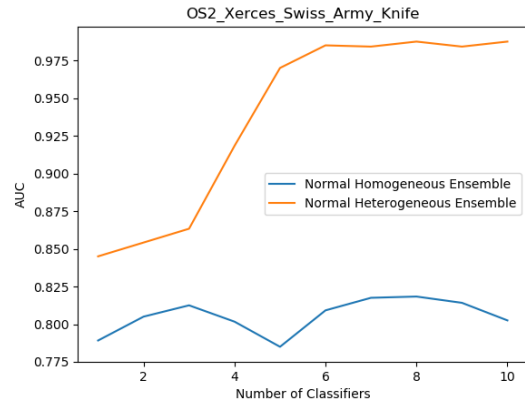
(a)



(b)

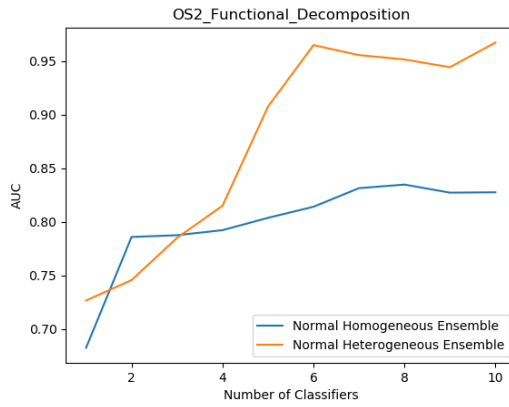


(c)

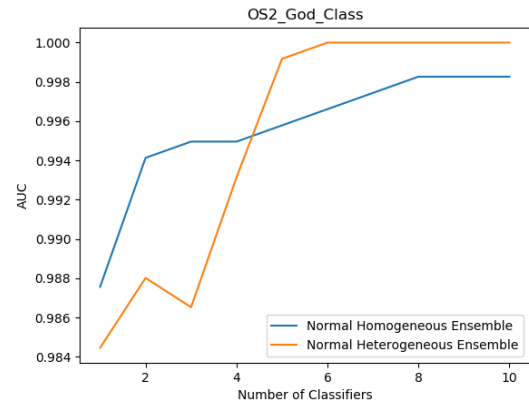


(d)

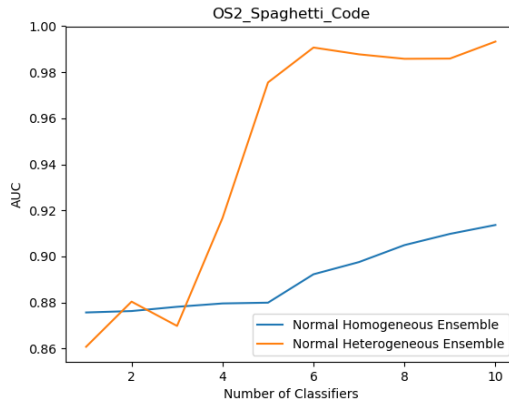
Figure (B.4) Performance evaluation of the normal APE models continued



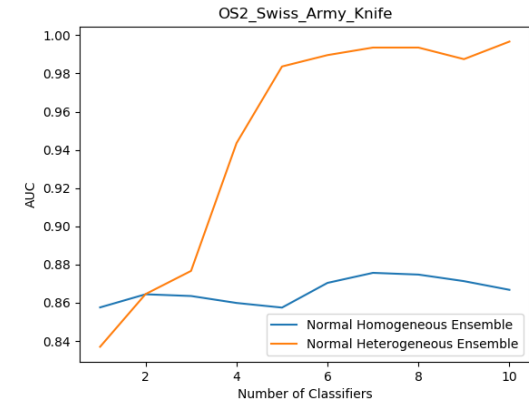
(a)



(b)



(c)



(d)

Figure (B.5) Performance evaluation of the normal APE models continued

VITAE

- **Name:** Ahmed Ali Hamdan AL-Shaaby
- **Nationality:** Yemen
- **Date of Birth:** 12/12/1987
- **Present Address:** Department of Information and Computer Science, King Fahd University of Petroleum and Minerals, P.O. Box 8585, Dhahran 31261, Saudi Arabia.
E-mail : G201408620@kfupm.edu.sa
- **Permanent Address:** Department of Computer Science, Thamar University, Thamar, Yemen.
- **Personal E-mail:** Ahmadali517@gmail.com
- **Pin/Zip:** +967
- **Telephone:** +9677773685246
- **Academic Background:**
 - Received Bachelor of Science (B.S.) in Computer Science from Thamar University, Yemen in 2012 with a GPA of 88.85%.

- Joined the Faculty of Computer Sciences and Information Systems as Demonstrator at Thamar University, Thamar, Yemen, from 2013 to 2015.
- Joined the Information and Computer science department as full time student at King Fahd University of Petroleum & Minerals (KFUPM), Dhahran, Saudi Arabia in September 2015.
- Completed Master of Science (M.S.) in Software Engineering from King Fahd University of Petroleum & Minerals (KFUPM), Dhahran, Saudi Arabia in December 2019, with a GPA of 3.438/4.0.

- **Publications:**

1. Al-Shaaby, A., Aljamaan, H., & Alshayeb, M. (2020). Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review. *Arabian Journal for Science and Engineering*, 1-29.
2. Almuhammadi, S., & Al-Shaaby, A. (2017). A survey on recent approaches combining cryptography and steganography. *Computer Science Information Technology (CS IT)*.
3. AL-Shaaby, A. A., & AlKharobi, T. (2017). Cryptography and Steganography: New Approach. *Transactions on Networks and Communications*, 5(6), 25-25.
4. Al-Shaaby, A., & Ahmed, A. (2018). How Do We Measure Project Success? A Survey. *J Inform Tech Softw Eng*, 8(229), 2.