Towards Code Smell Detection: A Multiphase Methodology Using Fuzzy Logic-based Feature Selection

Bruno Miguel Gonçalves Monteiro

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Nuno Gonçalo Coelho Costa Pombo

Novembro de 2024

Towards Code Smell Detection: A Multiphase Methodology Using Fuzzy Logic-based Feature Selection

Declaração de Integridade

Eu, Bruno Miguel Gonçalves Monteiro, que abaixo assino, estudante com o número de inscrição M12439 de Engenharia Informática da Faculdade de Engenharia, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o **Código de Integridades da Universidade da Beira Interior**.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 29/11/2024

Assinado por: **BRUNO MIGUEL GONÇALVES MONTEIRO**
Num. de Identificação: 30696167
Data: 2024.11.29 19:13:18+00'00'

Dedication

To my dearest parents, for their unwavering support and numerous sacrifices, which made my university journey possible.

Acknowledgements

At the end of this dissertation, it is with great gratitude and sincere appreciation that I extend my acknowledgements to those whose support and encouragement have been essential for its realization.

Firstly, I would like to express my deepest gratitude to my supervisor, Prof. Doutor Nuno Gonçalo Coelho Costa Pombo, for his unwavering support and guidance throughout this dissertation. I would, also, like to express my gratitude for the opportunity to work with him. This collaboration has not only enriched my academic journey, but has also been a pivotal source of growth, both personally and professionally. Working with my supervisor has been a privilege, and I am sincerely thankful for the opportunity.

Secondly, I would like to express my heartfelt gratitude to Prof. Doutor Kouamana Bousson of the Department of Aerospace Sciences for his invaluable help and support throughout this work. His guidance greatly contributed to the success of this dissertation. This collaboration enriched my academic journey and provided me with new insights into areas of study that were previously unfamiliar to me. Working with Prof. Doutor Kouamana Bousson has been a privilege, and I am truly thankful for both the opportunity and the time he dedicated to me for this work.

I extend my gratitude to all the members of the Secure and Intelligent Networked Software Systems laboratory (**sins**-lab) who, in various forms, have provided me with constant support in the realization of this dissertation. The enriching environment and collective expertise of the lab have played a pivotal role in shaping the outcomes of this research.

I would like to express my deepest appreciation to my parents and brother for their encouragement and belief in my abilities. Their support has been a constant source of strength and motivation throughout this academic journey.

Additionally, I would like to express my gratitude to my girlfriend for her support and daily motivation. Her encouragement undoubtedly played a significant role in the successful completion of this dissertation.

Finally, I want to thank all my friends, whom I consider as my second family, for the companionship and support they have provided throughout my time at this university. Thank you for all the discussions that we had during this research, which helped me unconditionally in my work.

Towards Code Smell Detection: A Multiphase Methodology Using Fuzzy Logic-based Feature Selection

Resumo

Garantir a qualidade de *software* continua a ser um desafio fundamental na engenharia de *software*. *Code smells* são características no código que podem indicar a necessidade de refatorização e indicam potenciais problemas no seu *design* e implementação, tornando o código mais difícil de compreender, manter e evoluir. Caso não sejam resolvidos, esses *smells* podem resultar em *bugs*, o que torna a sua resolução tão crítica como a dos próprios *bugs*.

Atualmente, diversos estudos têm explorado várias técnicas de *Machine Learning (ML)* para a deteção de *code smells*. No entanto, a classificação incorreta de instâncias continua a ser um desafio significativo, conduzindo frequentemente a resultados inconsistentes e comprometendo a fiabilidade das previsões do modelo. A seleção de caraterísticas surgiu como uma técnica fundamental para melhorar a acurácia destes modelos, reduzindo a dimensionalidade e removendo caraterísticas irrelevantes ou redundantes, permitindo assim previsões mais precisas na deteção de *code smells*. Entre estes, o método de seleção de caraterísticas baseado no *Modified Fuzzy C-Means algorithm with supervision (MFCMS)*, que integra *clustering* e aprendizagem supervisionada, revela-se bastante promissor. Ao captar relações complexas entre caraterísticas, o *MFCMS* tem o potencial de melhorar a fiabilidade da deteção de *code smells*, conduzindo a previsões mais consistentes.

Esta dissertação apresenta uma abordagem para melhorar a deteção do *code smell long method* utilizando técnicas de *ML*. O método integra o pré-processamento de dados, a seleção de caraterísticas e um modelo de *ML* para melhorar a acurácia e a eficiência. Embora os resultados experimentais tenham revelado que a abordagem proposta proporciona algumas melhorias relativamente ao modelo base, a sua principal contribuição reside na aplicação do *MFCMS* para redução da dimensionalidade, especialmente quando combinada com *Principal Component Analysis (PCA)*. Esta combinação reduziu a correlação entre caraterísticas e melhorou o processo de seleção. Apesar destes avanços, perduram alguns erros de classificação, evidenciando a necessidade de melhorias no futuro.

Este trabalho estabelece as bases para estudos futuros, particularmente na investigação de métodos de seleção de caraterísticas adequados para a deteção de *code smells*. Isso permitirá o desenvolvimento de ferramentas mais robustas e automatizadas, capazes de melhorar a qualidade e a manutenção do software.

Palavras-chave

Code smells; *Modified Fuzzy C-Means algorithm with supervision*; Seleção de caraterísticas; Qualidade de *Software*; Engenharia de *Software*.

Towards Code Smell Detection: A Multiphase Methodology Using Fuzzy Logic-based Feature Selection

Resumo alargado

Introdução

O primeiro capítulo tem como objetivo contextualizar o trabalho desenvolvido nesta dissertação, apresentando o tema principal e explicando a sua relevância. Neste capítulo, é abordada a motivação e o enquadramento do estudo, destacando a importância da temática no contexto atual. Além disso, é definido o problema que esta dissertação visa resolver, assim como os objetivos principais que a orientaram. Em seguida, o capítulo apresenta uma descrição detalhada das principais contribuições do trabalho, finalizando com a organização do documento.

Motivação e Enquadramento

Na área da engenharia de *software*, os testes e a garantia de qualidade são um passo importante para a produção de um *software* fiável e de boa qualidade. Neste contexto, existem vários métodos que foram introduzidos para melhorar a qualidade do código e garantir a robustez do *software*. Algumas dessas técnicas, como a refatorização e a previsão de defeitos, requerem a utilização de *code smells* como medidas ou características.

Os *bad smells* ou *code smells* são características no código de um programa que podem indicar um problema com o seu *design* e implementação, podendo tornar o código mais difícil de compreender e manter. *Martin Fowler* e *William Brown* definiram um extenso catálogo de *code smells* e, desde então, foram propostas várias ferramentas que visam detetar este tipo de *smells*. Atualmente, na literatura, existem cada vês mais propostas que utilizam *Machine Learning (ML)* para lidar com esta deteção, porém estes modelos apresentam alguns desafios, nomeadamente o elevado número de instâncias mal classificadas e problemas relacionados com o conjunto de dados utilizado.

Por este motivo, é essencial desenvolver ferramentas que possam detetar corretamente estes *smells*, melhorando não só a acurácia, mas também reduzir a dependência na intuição humana. Além disso, a não resolução destes *code smells* pode levar ao aumento do custo de manutenção de diferentes projetos ou pode também levar ao aparecimento de *bugs*. Integrando a deteção de *code smells* esta dissertação pretende contribuir para a área de investigação de *code smells*, fornecendo ideias e metodologias que possam também vir a ser aplicadas em estudos futuros.

Esta dissertação desenvolve-se em duas áreas interligadas – desenvolvimento de *software* e qualidade de *software*, com foco na deteção de *code smells*.

Definição do Problema e Objetivos

O principal problema abordado neste trabalho é a ineficiência e subjetividade na deteção de *code smells* em sistemas de *software*. Os métodos atuais dependem frequentemente da experiência do ser humano, o que pode levar a resultados inconsistentes e dificultar a manutenção

da qualidade do código. Além disso, muitos desses métodos apresentam desempenho limitado, especialmente no que diz respeito à precisão. Esta dissertação propõe uma abordagem automatizada para a deteção de *code smells*, com o objetivo de melhorar a eficiência e precisão desses mecanismos, fornecendo observações relevantes para a área de pesquisa.

O objetivo principal deste estudo é desenvolver uma abordagem de deteção de *code smells* que utilize técnicas de seleção de caraterísticas para reduzir a dimensionalidade dos dados e melhorar o desempenho do algoritmo. Ao identificar e selecionar as caraterísticas mais relevantes, pretende-se aumentar a precisão do algoritmo e, ao mesmo tempo, reduzir a complexidade computacional. Adicionalmente, a solução proposta poderá ser integrada noutras investigações desta área, contribuindo para a validação e consolidação da eficácia do método.

Os objetivos específicos desta dissertação incluem:

1. Investigar os principais desafios associados à deteção de *code smells*;
2. Identificar e selecionar uma técnica de seleção de caraterísticas adequada para esta aplicação;
3. Desenvolver uma solução que utilize esta técnica para aumentar a precisão da deteção;
4. Apresentar de forma detalhada a implementação efetuada para facilitar a sua integração em pesquisas e aplicações futuras.

Principais Contribuições

As principais contribuições, resultantes da investigação e implementação da abordagem proposta, são as seguintes:

1. Uma solução de deteção de *code smells*, com um foco particular na deteção do *code smell long method*;
2. Estabelecimento de uma base para investigação futura na área dos *code smells*, através da introdução de uma metodologia que integra eficazmente o mecanismo de seleção de caraterísticas implementado.

Base Teórica

O segundo capítulo providencia os principais conceitos sobre *code smells*, explorando as suas taxonomias e o impacto que estes têm no processo de desenvolvimento de *software* e na qualidade de *software*. Além disso, analisa alguns conceitos de *ML* devido à sua ampla aplicabilidade na deteção de *code smells*. Termina com a análise de um método denominado Fuzzy C-Means *clustering*, avaliando o seu potencial para a seleção das características do conjunto de dados, tentando, desta forma, reduzir a dimensionalidade do mesmo e, focando nas características mais importantes, poderá contribuir para que o modelo obtenha melhores resultados.

Desta forma, este capítulo apresenta as linhas gerais que definem a criação de um método de deteção de *code smells*, fornecendo a base teórica e metodológica necessária para o desenvolvimento da abordagem proposta nesta dissertação.

Trabalho Relacionado

O terceiro capítulo destaca trabalhos relacionados com a área de investigação de *code smells*, apresentando diferentes abordagens e técnicas utilizadas para a deteção destes problemas. Entre os estudos abordados, destacam-se aqueles que utilizam métodos de *ML* e técnicas de seleção de características, evidenciando o potencial destas metodologias na melhoria da precisão e eficiência dos modelos de deteção.

Neste contexto, realça-se a utilização do método *Modified Fuzzy C-Means algorithm with supervision (MFCMS)* para seleção de características, que integra técnicas de clustering com supervisão, mostrando-se promissor na redução da dimensionalidade dos dados e na melhoria da precisão dos modelos. Além disso, o conjunto de dados *Madeyski Lewowski Code Quest (MLCQ)* é analisado como uma fonte relevante e viável para estudos nesta área, devido à sua composição rica em projetos da indústria, o que reforça a sua aplicabilidade no desenvolvimento de soluções para a deteção de *code smells*.

Detalhes de Implementação

O capítulo 4 apresenta o principal desenvolvimento realizado ao longo deste estudo, detalhando a especificação da metodologia adotada, os processos realizados e as justificações para as escolhas feitas ao longo do trabalho. Adicionalmente, são discutidos o ambiente de computação utilizado e algumas das dificuldades encontradas durante a implementação da metodologia.

Em termos de experiências, foram realizadas duas, diferenciando-se pelo tipo de conjunto de dados utilizado. A primeira experiência utilizou apenas o conjunto de dados pré-processado com o método de normalização `MinMax`. No entanto, devido à possível correlação entre características, que poderia impactar negativamente o processo de seleção, decidiu-se, na segunda experiência, implementar *Principal Component Analysis (PCA)* para reduzir essa correlação. O objetivo era avaliar o impacto dessa abordagem no processo de seleção de características e verificar como a descorrelação das variáveis influenciava o desempenho do modelo de detecção de *code smells*. Essas duas experiências foram cruciais para a avaliação do método de detecção, uma vez que a redução da dimensionalidade e a remoção de correlações entre características podem otimizar a precisão e eficiência do modelo, garantindo melhores resultados na detecção dos *code smells*.

Resultados e Discussão

O capítulo 6 apresenta os resultados obtidos a partir das experiências realizadas, seguidos de uma análise e discussão detalhada dos mesmos.

Para avaliar as duas experiências realizadas, foi utilizado um conjunto de teste com dados nunca antes vistos pelo modelo, garantindo uma avaliação imparcial e precisa. Durante os testes, foi feita uma comparação com o modelo base de cada experiência, ou seja, com o modelo treinado sem a aplicação do método de seleção de características. Nos modelos que utilizaram a seleção de características, observou-se uma ligeira melhoria em termos de acurácia, precisão, *AUC*, e *F1-Score*. Contudo, constatou-se que ainda existia um número considerável de instâncias mal classificadas, o que pode ser atribuído ao facto do conjunto de dados não ser balanceado. Isso destaca a importância, para estudos futuros, de criar um conjunto de dados mais balanceado, a fim de avaliar melhor este cenário.

Apesar disso, o método de seleção de características utilizado revelou-se eficaz, especialmente na segunda experiência, que incluiu características descorrelacionadas. Assim, observa-se que o descorrelamento das características teve um impacto positivo no desempenho do método de seleção adotado.

Conclusão e Trabalho Futuro

O sétimo capítulo apresenta as principais conclusões do estudo desenvolvido nesta dissertação, destacando a importância da deteção de *code smells* no desenvolvimento de *software*. Além disso, propõe orientações para trabalhos futuros, com o objetivo de aprimorar e validar de forma mais robusta a metodologia apresentada.

Identificar e resolver *code smell* é crucial para garantir um processo de desenvolvimento de software eficaz. Isso assegura que o software permanece robusto, fácil de compreender, manter e evoluir. Por conseguinte, a deteção de *code smells* torna-se essencial para identificar e corrigir de forma proativa os potenciais problemas, conduzindo, em última análise, à eliminação destes defeitos do software. A aplicação de uma metodologia que envolva *ML* aliada de um bom método de seleção de características torna-se essencial para a deteção de forma automática deste tipo de *smells*.

Apesar das dificuldades encontradas durante a implementação, o protótipo da metodologia foi desenvolvido e avaliado com sucesso, permitindo atingir os principais objetivos propostos para este estudo. No entanto, ao analisar os resultados, constatamos que ainda há espaço para melhorias na nossa proposta.

Relativamente a trabalho futuro, foram identificadas várias melhorias que podem ser incorporadas na metodologia para otimizar os resultados obtidos. Entre elas, destaca-se a criação de um conjunto de dados mais balanceado, com o objetivo de avaliar o seu impacto nos resultados das duas experiências realizadas. Além disso, propõe-se a utilização de um método alternativo para otimização de hiperparâmetros, de forma a tornar este processo mais eficiente e eficaz em comparação com o método aplicado na nossa implementação.

Towards Code Smell Detection: A Multiphase Methodology Using Fuzzy Logic-based Feature Selection

Abstract

Ensuring the quality of software code remains a fundamental challenge in software engineering. Code smells are characteristics in the source code that could infer a need for refactoring and imply potential problems with its design and implementation, making the code harder to understand, maintain, and evolve. If left unresolved, these smells can lead to the appearance of bugs, making them as critical to address as software defects themselves.

Recent research has explored various Machine Learning (ML) techniques for detecting code smells. However, misclassification continues to be a significant challenge, often leading to inconsistent outcomes and compromising the reliability of the model's predictions. Feature selection has emerged as a key technique to improve the accuracy of these models by reducing dimensionality and removing irrelevant or redundant features, thereby allowing for more precise predictions in code smell detection. Among these, the feature selection method based on Modified Fuzzy C-Means algorithm with supervision (MFCMS), which integrates both clustering and supervised learning, shows significant promise. By capturing complex relationships between features, MFCMS has the potential to improve the reliability of code smell detection, leading to more consistent predictions.

This dissertation introduces a multiphase approach to enhance the detection of the long method smell using ML techniques. The method integrates data preprocessing, feature extraction, and machine learning modeling to improve both accuracy and efficiency. Although experimental results revealed that the proposed approach provides some improvements over the baseline model, its main contribution lies in applying MFCMS for dimensionality reduction, especially when combined with PCA. This combination reduced feature correlation and enhanced the feature selection process. Despite these advancements, some misclassifications remain, indicating that further refinement is needed.

This work lays the foundation for future research, particularly in exploring feature selection methods for code smell detection. It unlocks the potential for developing more robust, automated tools that can improve software quality and maintainability

Keywords

Code smells; Modified Fuzzy C-Means algorithm with supervision; Feature selection; Software quality; Software engineering.

Contents

List of Figures

List of Algorithms

List of Tables

Acronyms

AI Artificial Intelligence

FN False Negative

FP False Positive

ML Machine Learning

TN True Negative

TP True Positive

AUC Area Under the Curve

BBN Bayesian Belief Network

FCM Fuzzy C-Means

FPR False Positive Rate

OOP Object-oriented programming

PCA Principal Component Analysis

ROC Receiver Operating Characteristic

SMO Sequential Minimal Optimization

TPR True Positive Rate

UCI University of California–Irvine

WSL Windows Subsystem for Linux

k-NN k-nearest neighbors

NLOC Number of Lines of Code

MLCQ Madeyski Lewowski Code Quest

SBSE Search-based Software Enginnering

MFCMS Modified Fuzzy C-Means algorithm with supervision

CC-CDS Competitive Coevolutionary Code-Smells Detection

LibSVM Library for Support Vector Machines

sins-lab Secure and Intelligent Networked Software Systems laboratory

Chapter 1

Introduction

This chapter lays the foundation for the research presented in this dissertation, highlighting the significance of the study and establishing the context in which it is situated. It provides a basis for understanding the academic relevance and practical implications of the research, setting the stage for the work that follows. The chapter is organized into the following sections:

- **Motivation and Scope (1.1)** – this section outlines the primary motivation behind this study and defines its scope. It explains the key challenges in detecting code smells, emphasizing the need for effective solutions in this domain;
- **Problem Statement and Objectives (1.2)** – this section outlines the main problem that motivates this study. It also presents the primary objectives to be achieved throughout this dissertation;
- **Main Contributions (1.3)** – this section highlights the key contributions of this dissertation;
- **Document Organization (1.4)** – this section provides an overview of the overall document organization, succinctly describing the content of each chapter.

1.1 Motivation and Scope

In software engineering, testing and quality assurance are crucial steps towards producing high-quality and reliable software [1]. Numerous methods have been suggested to enhance the code quality and ensure the software robustness [2]. A few of those techniques, such as refactoring and defect prediction, require using bad smells as measures or features [2].

Code smells, or bad smells, refer to characteristics in a program's source code that may reveal a problem with its design and implementation. These smells can make code harder to comprehend, maintain and evolve, ultimately affecting the software scalability and adaptability over time [3].

An extensive catalog of code smells has been defined, originating from the pioneering work of Martin Fowler and William Brown [4]. Since then, a number of tools have been proposed to detect these smells [5]. More recently, there has been a trend in research applying ML models for code smell detection [6]. While these models show promise, they also face challenges, including misclassification of code smells and limitations related to the dataset quality [7].

Therefore, it is essential to develop tools that can consistently and automatically detect code smells. These tools should not only enhance detection accuracy, but also reduce reliance

on human expertise, as code smell detection often depends heavily on subjective judgment. Moreover, failing to address this code smells can lead to the accumulation of technical debt, which increases maintenance costs and may ultimately result in software bugs.

Scalable and objective detection tools offer a path towards reducing reliance on human expertise and intuition, making it easier to integrate code smell detection into continuous software development workflows. Early detection of code smells helps maintain a clean codebase, reduces the risk of bugs, and supports easier adaptation and maintenance across software projects. This dissertation aims to contribute to code smell detection research by offering insights and methodologies that improve code smell detection and establish a foundation for further studies in this field.

The scope of this dissertation unfolds within two interconnected areas – software development and software quality – with focus on the detection of code smells.

1.2 Problem Statement and Objectives

The main problem addressed in this work is the inefficiency and subjectivity involved in detecting code smells within large-scale software systems. Current detection methods often depend on human expertise and intuition, which can lead to inconsistent results and increase the challenge of maintaining code quality. Moreover, some methods show limited performance when evaluated solely by accuracy, as the precision of these detection mechanisms often falls short. This dissertation seeks to address these challenges by developing an automated code smell detection approach that aims to improve both the precision and efficiency of detection. An approach like the one proposed here could provide valuable insights within the field of code smells research.

The primary objective of this study is to design a code smell detection approach that leverages feature selection to reduce dataset dimensionality and enhance the overall performance of the detection algorithm. By identifying and selecting the most relevant features, this approach aims to increase precision while minimizing computational complexity. Additionally, this study seeks to develop a solution that can be integrated into broader research initiatives, contributing to further validation and consolidation of the approach's effectiveness. Importantly, it aims to introduce a feature selection method that can be analyzed and applied more widely within the code smells domain.

The specific objectives of this dissertation are as follows:

1. Research the key challenges associated with code smell detection;
2. Identify and select a suitable feature selection approach for effective application in code smell detection;
3. Develop a code smell detection solution that utilizes this feature selection mechanism to enhance detection precision;

4. Provide an in-depth detail of the implementation to support integration into future research and practical applications.

1.3 Main Contributions

The main contributions of this dissertation, derived from the research and implementation of the proposed approach, are outlined as follows:

1. A code smell detection solution, with a particular focus on detecting the long method smell;
2. Establishment of a foundation for future research in the field of code smells, by introducing a methodology that effectively integrates the implemented feature selection mechanism.

1.4 Document Organization

The current document is structured into six chapters and is outlined as follows:

- Chapter 1 – **Introduction** – presents the motivation and scope of this dissertation, along with its objectives and main contributions. This chapter concludes with an overview of the document organization;
- Chapter 2 – **Background** – provides a comprehensive overview of the foundational concepts and theories related to code smells, as well as key ML concepts that are relevant to the research;
- Chapter 3 – **Related Work** – offers a review of existing research relevant to the dissertation topic, with a particular focus on methods for detecting code smells;
- Chapter 4 – **Implementation Details** – provides a detailed explanation of the implementation processes undertaken as part of the proposed methodology;
- Chapter 5 – **Results and Discussion** – presents the findings from the experiments conducted to evaluate the proposed code smell detection approach, followed by a detailed interpretation of the results;
- Chapter 6 – **Conclusion and Future Work** – summarizes the key findings and contributions of the dissertation, reflecting on the overall research process and its impact on the field of code smell detection. Additionally, it identifies the limitations of the current approach and suggests potential solutions for future research.

Chapter 2

Background

2.1 Introduction

This chapter explores the foundational concepts of code smells, examining their definitions, taxonomy, and impact on the overall software development and quality assurance. It also provides an in-depth review of the different approaches used for code smell detection, with a particular focus on those employing ML.

This chapter is divided into three main sections:

- **Code smells (2.2)** – provides an overview of the code smells concept, the taxonomy, the specific code smell applied in the study and different approaches for code smell detection;
- **Machine Learning for Code Smell Detection (2.3)** – presents an overview of ML concepts, including mechanisms to improve performance such as hyperparameter optimization, the main learning model used in code smell detection, and the performance metrics utilized in this field;
- **Fuzzy C-Means Clustering (2.4)** – provides an explanation of the Fuzzy C-Means (FCM) concept and explores its potential application in feature selection.

2.2 Code smells

2.2.1 Concept

Code smells, or bad smells, are characteristics in the source code that could infer a need for refactoring and imply potential problems with its design and implementation [4]. These smells have the potential to negatively impact the code, influencing factors such as maintainability, evolvability and overall comprehensibility of the codebase.

As significant as software bugs, code smells are treated with a high level of importance by software engineers because they have the potential to evolve into bugs if left unaddressed. Moreover, refactoring works as a proactive measure to reduce the negative effects of code smells and thereby alleviate the likelihood of bug occurrences in the future [8].

Originally, Fowler and Beck described a collection of code smells and explained how they can be addressed through refactoring. Additionally, the authors emphasize that the refactoring process is a continually evolving task, given the absence of a precise criterion to determine its completion [4]. Nonetheless, researchers have been challenging this idea by developing techniques capable of automatically detecting code smells in a given project [9].

2.2.2 Taxonomy of Code Smells

Mäntylä et al. conducted a detailed study of the code smells outlined by Fowler and Beck, aiming to establish a taxonomy for these smells. Also, they found it beneficial to categorize similar code smells together, as it improves their overall comprehensibility [10]. Initially, they classified the code smells into seven categories [10]; however, more recently Mäntylä et al., proposed a revised version containing only six categories [11]. Taking this into consideration, they proposed the following categories:

- **Bloaters** – refer to code, methods, or classes that have grown significantly over time, making them difficult to manage. This frequently happens as new features or changes are introduced;
- **Object-orientation abusers** – encompasses all the code smells associated with malpractices of Object-oriented programming (OOP) languages principles;
- **Change preventers** – refers to code structures that significantly impede the modification of the software. To clarify, these are signs that modifying one element in a particular location will require changes in other places as well;
- **Dispensables** – refers to elements in the code whose absence would make the code cleaner, more efficient and increase the overall comprehensibility;
- **Couplers** – refers to smells that involve classes or components tightly interconnected or dependent of each other;
- **Others** – smells that do not fit into any of the previous categories.

Detecting all code smells is crucial for maintaining a high-quality codebase. However, developing a system capable of identifying every code smell simultaneously remains an ongoing challenge. As a result, most researchers prioritize the detection of specific code smells, focusing on a selected group. This dissertation focus on the detection of a specific Bloater code smell, providing a prototype that showcases a promising approach and paves the way for its broader application to other code smells.

2.2.3 Long Method Smell Description

The Long Method code smell was first introduced by Fowler and Beck in their book [4], and was later summarized and refined by the authors of [12] in a more accessible form.

A Long Method is typically defined by an excessive Number of Lines of Code (NLOC), which can negatively impact code readability and maintainability. According to [12], a method with more than ten lines should be inspected for potential signs of this smell.

Code smells associated with size and complexity, such as the Long Method, are among the most prevalent in software development [13]. They often arise unintentionally as new functionalities are added to methods without removing or restructuring existing code. Over time,

this incremental expansion can result in bloated methods that become overly complex and difficult to manage, often going unnoticed during the development process.

Addressing the Long Method smell typically involves refactoring large methods into smaller, more modular components. This practice not only enhances code maintainability but also improves readability and quality. As highlighted in [12], breaking down larger methods into well-defined units is a widely recognized technique for mitigating this common issue.

2.2.4 Techniques for Code Smell Detection

According to Fowler and Beck, detecting code smells in software is a human process based on intuition [4]. This task typically becomes easier as developers enhances their experience in code smell detection.

However, this process is very-time consuming and often subjective, as it depends heavily on the individual performing the detection [4]. To make this more unbiased, recent studies focused on developing approaches for detecting code smells automatically [14], reducing the reliance on human intuition and minimizing potential biases in the detection process.

Despite various efforts to create an automated code smell detection system, as highlighted in [5], the task remains challenging due to the complexity of accurately identifying these smells.

In an effort to better categorize these approaches, Kessentini et al. proposed a classification for code smell detection techniques [15]. This system organizes approaches into categories ranging from fully automated to guided manual inspection. The authors identified seven main categories for existing detection techniques: manual approaches, symptom-based approaches, metric-based approaches, probabilistic-based approaches, visualization-based approaches, search-based approaches, and cooperative-based approaches. Additionally, ML-based approaches have been emerging as a potential solution for enhancing code smell detection accuracy and precision.

2.2.4.1 Manual Approaches

In the literature, Fowler and Beck outlined a series of code smells and how they could be identified manually through human intuition [4]. These techniques rely on the expertise of the human conducting the detection, which makes them arduous, time-consuming and susceptible to errors. Another concern with these approaches is their tendency to rely more on human intuition than on exact scientific principles [15].

2.2.4.2 Symptom-based Approaches

Symptom-based approaches, as highlighted in [15], encompass the characterization of code-smell symptoms involving various notions such as class roles and structures. Symptom descriptions are then analyzed and correlated to detection algorithms. Some works that applied this approach include the example of DECOR [16]. The authors of the latter defined the steps necessary for code smell specification and detection of smells [16].

However, this approach faces two primary limitations [15]. Firstly, symptoms are not always objectively measurable and can be subjective and difficult to define. Secondly, the number of code smells that requires manual specification, classification based on rules, and association with detection methods can be exceptionally vast for a comprehensive list of code smells [15]. Consequently, the inherent ambiguity and context-dependence of symptoms contribute to the time-consuming and error-ridden nature of symptom-based approaches [15].

2.2.4.3 Metric-based Approaches

These approaches analyze various software metrics to quantify the structural properties of the code in order to detect code smells [15]. The metrics used in these analyses serve as quantitative indicators, helping in the identification of potential areas of concern that may require closer inspection or refactoring [15]. This metric-based approach enables a comprehensive analysis of the codebase, assisting in the improvement of the overall code quality and the detection of potential issues.

Radu et al. introduced a mechanism known as detection strategy, which encompasses a set of metric-based rules designed to identify discrepancies from well-established design principles. This framework allowed the authors to define detection strategies for ten design problems [17].

According to [15], the efficacy of employing metric-threshold combinations to identify code smells remains a subject of debate. Specifically, for each code smell, the rules that categorize its characteristics need to be calibrated to determine the appropriate threshold values for the associated metric. It is important to emphasize that different thresholds should be tested to identify the most effective configuration, as there is no rule of thumb for this task.

2.2.4.4 Probabilistic-based Approaches

Probabilistic-based approaches involve determining the probability of occurrence of a specific smell. The majority of the implementations that use this approach employ fuzzy logic or Bayesian Belief Network (BBN) [15]. The output of these systems is the probability of occurrence of a given smell.

2.2.4.5 Visualization-based Approaches

Visualization-based code smell detection methodologies typically involve semi-automated processes to guide developers in code smell identification [5]. The main objective is to leverage human expertise in incorporating complex contextual information within the detection process [15]. An example of this approach can be found in a framework developed by Guillaume et al., which supports quality analysis by seamlessly integrating human expertise into the evaluation process [18].

2.2.4.6 Search-based Approaches

Search-based approaches for code smell detection are derived by contributions made in the Search-based Software Enginnering (SBSE) domain [15]. SBSE applies search-based techniques to optimize problems within software engineering [15]. Most implementations of this approach incorporate ML techniques, which have become one of the primary methods used by researchers in the detection of code smells [5]. While ML has shown promise in this area, its effectiveness is often contingent upon the quality of the data used. Poor-quality or insufficient data can significantly impact the performance and accuracy of ML algorithms, making data quality a critical factor in achieving reliable results [15].

2.2.4.7 Cooperative-based Approaches

Cooperative-based approaches aim to enhance both accuracy and performance in the detection of code smells. This is achieved through the collaborative execution of multiple activities [5]. In their work, Boussaa et al. introduce an approach called Competitive Coevolutionary Code-Smells Detection (CC-CDS). This method employs two distinct populations that work cooperatively [19]. The first population generates a comprehensive set of detection rules based on metric-based techniques. Simultaneously, the second population actively maximizes the number of artificial code smells that are not covered by the current set of rules. This dynamic interplay between the two populations shows the superiority of competitive co-evolution over single population evolution, achieving enhanced efficiency and effectiveness in uncovering code smells [19].

2.2.4.8 Machine learning-based Approaches

The approaches presented in the sections 2.2.4.1 to 2.2.4.7 have specific issues that may limit their applicability. Firstly, many approaches for code smell detection can be subjective, leading to lack of agreement among the detectors. Secondly, most detectors rely on metric-threshold specifications. This means that selecting the correct threshold is a critical aspect, as it significantly influences the performance of the detectors [20]. Considering this, ML techniques can be implemented to detect code smells, while also addressing these current limitations [21].

ML has been applied in various scenarios for code smell detection, primarily using binary classification to determine whether a code block contains a specific smell based on the extracted features [22]. Recent research suggests that the choice of ML approaches has enhanced the effectiveness of detecting these smells [20].

Furthermore, the application of ensemble methods, such as random forest, has been proposed and shown to achieve a promising result in code smell detection [23]. However, there are factors that affect the replicability of these studies. The primary challenge is the lack of availability and detailed explanation of how some datasets are handled by the authors [24].

Moreover, some features in the dataset might not significantly contribute to detecting a code smell, but their inclusion can still affect the classification model's outcome. This underscores

the need for feature selection to eliminate irrelevant variables, which, in theory, can improve the model's precision [25].

It is important to note that Section 2.3 will provide a detailed exploration and explanation of ML approaches for code smell detection, which is essential to the work carried out in this dissertation.

2.3 Machine Learning for Code Smell Detection

Machine Learning (ML) stands out as one of today's most prominent technologies, with applications spanning a wide range of research fields and industries. Within software engineering, ML plays a crucial role in tasks such as software bug prediction [26] and is increasingly recognized as an effective approach for code smell detection. This section explores the importance of ML in code smell detection, highlighting its effectiveness in detecting these smells [22]. Additionally, it discusses how feature selection and hyperparameter optimization can be leveraged to improve the performance of these models.

2.3.1 Concept of Machine Learning

ML is a subset of Artificial Intelligence (AI), which involves creating algorithms that perform tasks requiring human-like intelligence. These algorithms can automatically produce useful results from given inputs [27]. In simpler terms, ML enables computers to learn from data or past experiences and make predictions or descriptions without being explicitly programmed to do so [28].

This process is based on a model with predetermined parameters, which represent the internal structure and logic of the model. During the learning phase, the model fine-tunes these parameters using training data or past experiences. Once trained, the model can make predictions if it is a predictive model or analyze data if it is a descriptive model [28, 27].

In ML there are three main types of learning:

- **Supervised Learning** – As mentioned earlier, this type of learning is widely used in ML and involves training models on labeled data. Labeled data consists of input features (i.e., predictors) and corresponding outputs. By learning the relationship between these features and outputs during training, the model can effectively predict outcomes on new and unseen data. In essence, supervised learning enables the model to generalize its predictions based on the patterns it has learned from the provided examples [27];
- **Unsupervised Learning** – Unlike supervised learning, which deals with labeled data and predefined outcomes, unsupervised learning focuses on unlabeled data. In this case, the model's objective is to uncover hidden patterns, structures, or groupings within the data itself [28]. This can involve tasks such as clustering data points into groups based on their similarities or identifying anomalies and outliers within a dataset [27]. The model learns to identify these patterns and relationships independently, without relying on prior knowledge or human intervention. [29];

- **Reinforcement Learning** – This approach entails an agent interacting with an environment, learning through trial and error. The agent receives rewards for desired actions and penalties for undesired ones based on the actions it takes to achieve a certain goal [28]. Over time, the agent adjusts its actions based on the received rewards and penalties to improve its decision-making and optimize its behavior within the given environment [30].

Supervised learning is commonly used in code smell detection, where classifiers are trained on software metrics to identify the presence or absence of code smells. This approach is particularly beneficial in the field, as ML can learn complex patterns from data, potentially leading to more accurate and effective code smell detection [28].

However, the performance of supervised learning models is highly dependent on the quality and size of the training data [27]. Considering this, it is essential to carefully choose the dataset for the detection system and ensure the data is relevant for the problem being solved.

Moreover, supervised learning models can be further refined through various techniques, including data preprocessing, feature selection, and hyperparameter tuning and optimization [27].

Data preprocessing is crucial because the quality of the data directly affects the model performance. To improve data for training, it is important to clean the data, handle missing values, and perform normalization. Normalization ensures that feature values are scaled appropriately, which can improve performance, speed up convergence and reduce the risk of overfitting [31]. Additionally, by scaling features to a similar range, normalization prevents features with larger scales from dominating the learning process, ensuring equal contribution from each feature [32].

Feature Selection is a crucial technique in ML that can significantly enhance the model accuracy and efficiency. By selecting the most critical features and removing redundant and irrelevant ones, feature selection increases the predictive power of algorithms. This process is particularly valuable as it directly impacts the model performance, reduces overfitting and lowers computational costs [33].

The process of feature selection has three main benefits:

- **Enhanced model performance** – the noise in the data is reduced with the removal of irrelevant and redundant features. This allows the learning algorithms to focus on the most important features, leading to more accurate and generalizable models [34];

- **Reduced computational cost** – datasets with fewer features require less processing power to train models compared to larger ones. This makes the training time faster and enables testing of more complex algorithms that might have been too computationally expensive to train otherwise [34];

- **Improved interpretability** – by focusing only on the most important features and removing noise, it becomes easier to understand how the model makes its predictions. This approach can help identify which features contribute more to those predictions, enhancing the interpretability of the model [35].

In the context of feature selection, various methods can be applied to perform this task. However, this dissertation introduces an alternative method for feature selection that has not been utilized in code smell detection, but shows promise for this research. Therefore, section 2.4 will discuss and explain how fuzzy c-means clustering can be employed for feature selection, highlighting its benefits for use in this field.

Hyperparameter optimization is a critical aspect of ML model development. The performance of any ML model is significantly influenced by its hyperparameters, as these parameters control the learning process and define the model's architecture. However, in some cases, optimization techniques are not applied to select the optimal hyperparameters. The absence of a systematic approach often leads practitioners to rely on trial-and-error strategies, which can result in suboptimal configurations and negatively impact the model's performance [36].

Hyperparameter optimization addresses this challenge by systematically searching for the best combination of hyperparameters. This process is treated as an optimization problem, where different hyperparameter configurations are evaluated through training and validation at each iteration. By exploring a range of possible values, hyperparameter optimization aims to find the set of parameters that maximizes the model's performance. Section 2.3.2 provides a more detailed description of the hyperparameter optimization technique applied in this study.

2.3.2 Hyperparameter Optimization

As mentioned earlier, hyperparameter optimization is a crucial step in developing ML models. Hyperparameters differ from model parameters because they are set before the training process begins and cannot be learned during training. The challenge in optimizing hyperparameters lies in the vast number of possible combinations, making it difficult to find the optimal configuration [37].

To simplify this process, various optimization techniques are used depending on the specific learning model. Two of the most commonly used methods are Random Search and Grid Search [37]. Among these, Grid Search is typically preferred for its systematic approach in exploring the hyperparameter space.

2.3.2.1 Grid Search

Grid Search is a widely used technique for hyperparameter optimization in ML. It involves exhaustively searching through a predefined subset of hyperparameters to find the optimal combination for a given model. The process entails defining a grid, which represents all possible combinations of hyperparameter values within a specified range. Each combination

is then tested by evaluating the model's performance using an objective function, and the best-performing combination is selected based on a chosen performance metric [38].

This approach is valued for its simplicity and effectiveness, especially for smaller and less complex models, as it guarantees that all possible hyperparameter combinations within the search space are explored [37]. However, Grid Search can become computationally expensive and time-consuming, particularly when dealing with models that have a large number of hyperparameters or when the range of values for each hyperparameter is large [38].

Despite its computational cost, the simplicity of Grid Search is one of its key advantages. It is straightforward to implement and does not require advanced algorithms or specialized techniques. This ease of implementation makes it a popular choice. The Grid Search procedure is outlined in Algorithm 1, highlighting the step-by-step process of testing each possible combination of hyperparameters.

Algorithm 1 Grid Search

1: **Define the search space:**
2: **for** each hyperparameter H_i **do**
3: Specify the range of possible values V_i for H_i.
4: **Create a grid of all possible combinations of parameter values:**
5: Generate a matrix of all possible combinations of the specified hyperparameter values.
6: **for** each combination $C_k = \{v_1, v_2, \ldots, v_n\}$ where $v_i \in V_i$ **do**
7: **Train and evaluate:** Train the model M using configuration C_k and evaluate its performance P_k on a validation set using a specified metric (e.g., accuracy or mean squared error).
8: **Record results:** Store the configuration C_k and its corresponding performance P_k.
9: **Select the best combination:** Identify the configuration C_{best} that resulted in the best performance P_{best}.
10: **Output:** C_{best}.

2.3.3 Machine Learning Models

Many ML models have been used for detecting code smells. However, research in this field shows that Random Forest stands out as a particularly effective option. Random forest can effectively detect code smells, making it a popular choice among researchers and practitioners in this field [39, 40].

Studies suggest that Random Forest outperforms other ML models in the detection of smells, achieving superior precision, recall and F-Measure on these tasks [39].

2.3.3.1 Random Forest

Random Forest is a supervised learning algorithm known for its accuracy and robustness. It excels at tackling various tasks due to its ability to handle complex data patterns [41]. This algorithm is an ensemble method that builds a forest of numerous uncorrelated decision trees – referred to as estimators – each producing its own prediction [42]. The final prediction

is then obtained by combining the individual predictions from these trees, which helps to improve the overall accuracy, as illustrated in Figure 2.1.

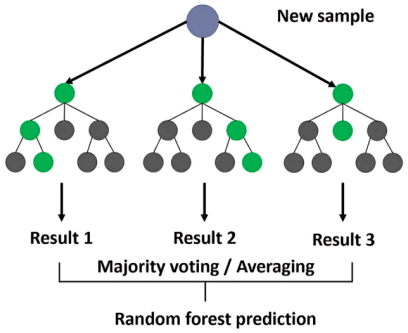

Figure 2.1: Random Forest Scheme [43].

Random Forest algorithm is as an efficient and accurate algorithm for predictive modeling. It is well-suited for large datasets with high-dimensional feature spaces and performs well even when some data points have missing values [41]. Additionally, Random Forest has relatively shorter training times compared to other complex ensemble models, making it a practical choice for real-world applications.

One of the key strengths of Random Forest lies in its ability to reduce the risk of overfitting – a common issue in decision trees. Individual decision trees are prone to overfitting because they can become overly complex, capturing noise and specific patterns in the training data that do not generalize well to unseen data. By constructing multiple decision trees and averaging their predictions, Random Forest effectively reduces the overall variance and prediction error, which lowers the tendency to overfit [42].

Another essential feature of Random Forest is the incorporation of randomness, which is introduced by training each tree on a unique subset of features and data samples. Specifically, each decision node within a tree is trained on a randomly selected subset of features rather than the entire feature set, ensuring that each tree captures slightly different aspects of the data. This approach helps to decorrelate the trees, making them more independent from one another and further enhancing the accuracy of the final prediction by reducing bias and variance [41].

In summary, Random Forest's ensemble method, combined with its robust handling of high-dimensional data and its mechanisms to prevent overfitting, makes it an ideal choice for code smell detection and other ML tasks where both precision and generalization are essential.

2.3.4 Performance Metrics

Evaluating a ML model is crucial to determine its effectiveness and suitability for a specific research domain. By analyzing various performance metrics, it is possible to gain insights into the model's accuracy, precision, recall, and overall behavior, thereby identifying its strengths and limitations.

A fundamental step in evaluating performance metrics involves understanding the concept and structure of the Confusion Matrix, which is essential for accurately interpreting results in classification tasks.

The Confusion Matrix is a structured table used to evaluate a classification model by comparing predicted and actual class labels, typically using data that the model has not seen during training. The term *confusion* refers to the matrix's role in identifying instances where the model misclassifies, revealing how it may confuse one class with another [44].

In binary classification, as applied in models like Random Forests, the confusion matrix contains four elements that represent different prediction outcomes, as shown in Figure 2.2.

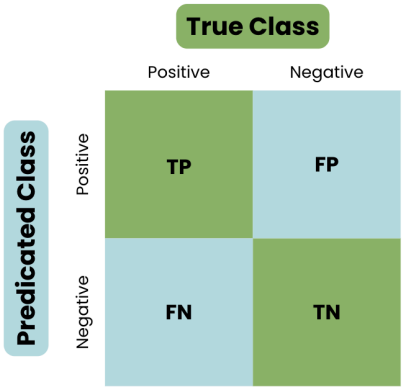

Figure 2.2: Confusion Matrix Structure [45].

Additionally, understanding the fundamental elements of a confusion matrix, as shown in Figure 2.2, is essential, as these components directly influence the calculation of performance metrics that evaluate model accuracy, precision, and recall. This confusion matrix consists of four core elements, each representing a distinct type of prediction outcome based on the actual and predicted classes:

- **True Positive (TP)** – occur when the model correctly classifies an instance as positive, meaning the predicted outcome is positive and aligns with the actual positive class. This indicates the model's ability to accurately identify instances that belong to the positive class [44];

- **True Negative (TN)** – occur when the model correctly classifies an instance as negative, meaning the predicted outcome is negative and corresponds with the actual negative class. This reflects the model's ability to accurately identify instances that belong to the negative class [44];

- **False Positive (FP)** – occur when the model incorrectly classifies an instance as positive, meaning the predicted outcome is positive while the actual class is negative. This indicates the model's failure to distinguish instances that belong to the negative class, misclassifying them as positive [44];

- **False Negative (FN)** – occur when the model incorrectly classifies an instance as negative, meaning the predicted outcome is negative while the actual class is positive. This highlights the model's failure to identify instances belonging to the positive class, misclassifying them as negative [44]. [44].

As stated previously, several key metrics are commonly used to evaluate the performance of classification models, some of which can be derived from the confusion matrix. These metrics are explained as follows:

- **Accuracy** – accuracy measures the overall correctness of a classifier by quantifying the ratio of correct predictions to the total number of predictions, as demonstrated by equation 2.1. However, it is reliable only when there is an equal number of samples belonging to each class [46].

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN} \tag{2.1}$$

  where TP represents the True Positives, TN the True Negatives, FP the False Positives and TN the True Negatives;

- **Precision** – precision measures the proportion of true positives predictions among all positive predictions made by the model, as demonstrated by equation 2.2. It is a crucial metric because it addresses a key limitation of accuracy described above[47].

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.2}$$

  where TP represents the True Positives and FP the False Positives;

- **Recall** – recall, also known as sensitivity, calculates the fraction of true positive predictions among all actual positive instances, as shown by equation 2.3 [47].

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.3}$$

  where TP represents the True Positives and FN the False Negatives;

- **F1-Score** – F1-Score is a useful metric for evaluating model performance in classification tasks, particularly when dealing with imbalanced data or binary classification [48]. It combines precision and recall into a single metric using the harmonic mean, as shown in equation 2.4. Unlike the arithmetic mean, the harmonic mean gives more weight to smaller values, ensuring a balanced result. This property prevents models from achieving a high F1-Score by excelling in only one of precision or recall. The F1-Score ranges from 0 to 1. A score closer to 1 indicates that the model achieves

a good balance between precision and recall, accurately identifying positive instances while minimizing FP and FN. A score closer to 0 suggests that the model struggles to make correct classifications in terms of both precision and recall.

$$\text{F1-Score} = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{2.4}$$

- **Area Under the Curve (AUC)** – the AUC is a metric used to evaluate the effectiveness of a classifier. It quantifies the model's ability to distinguish between the two classes it aims to predict. The AUC is derived from the Receiver Operating Characteristic (ROC) curve, which illustrates the trade-off between the True Positive Rate (TPR) and the False Positive Rate (FPR) at various classification thresholds. The TPR represents the proportion of actual positive instances correctly identified, while the FPR represents the proportion of negative instances incorrectly classified as positive. An AUC score of 1 indicates perfect classification, meaning the model can distinguish between the two classes with complete accuracy, whereas an AUC of 0.5 suggests that the model's performance is no better than random guessing. In general, a higher AUC indicates a more effective classifier, while a lower AUC value suggests poorer performance [49].

2.4 Fuzzy C-Means Clustering

2.4.1 Concept

Fuzzy C-Means (FCM) is a soft clustering technique that allows one sample of data to belong to two or more clusters [50]. Contrary to traditional unsupervised clustering algorithms, which assigns a data point to a single clustering, FCM assigns a membership degree (probability of belonging to a specific cluster) between 0 and 1 for each data point for each cluster [51]. This is determined by the distance of all data items to the centroids (center of the clusters) [52].

In clustering, data samples are organized into groups, typically classified into two primary types:

- **Hard Clustering** – in hard clustering, each data sample is assigned exclusively to a single cluster. This approach implies that each sample can belong to only one specific cluster;
- **Soft Clustering** – in soft clustering, a data sample may belong to multiple clusters, each with an associated probability. This approach allows samples to have membership in more than one cluster, reflecting degrees of membership.

The FCM algorithm aims to minimize an objective function, presented in Equation 2.5, that represents the distance from any given data point to a cluster center weighted by the membership of the data point in the cluster.

$$J_m(U, v) = \sum_{k=1}^{N} \sum_{i=1}^{c} u_{ik}^m \|y_k - v_i\|^2 \tag{2.5}$$

where N is the number of data points, c is the number of clusters, y_k is the i-th data point, v is the vectors of centers, v_i is the center of cluster i, u_{ik} is the degree of membership of y_k in cluster i and m is the fuzzification parameter that controls the degree of fuzziness (in clustering refers to the extent to which a data point can have partial membership in multiple clusters simultaneously) [53, 54].

The FCM minimizes the objective function through a series of iterative refinements [50]. This process is performed as follows:

1. **Initialization** – initialize the cluster centers randomly or based on some prior knowledge or heuristic. Define the number of clusters c and the fuzzification parameter m;

2. **Calculation of Membership Values** – calculates the membership values u_{ik} for each data point y_k in each cluster i using the equation 2.6.

$$u_{ik} = \frac{1}{\sum_{j=1}^{c} (\frac{\|y_k - v_i\|}{\|y_k - v_j\|})^{\frac{2}{m-1}}} \tag{2.6}$$

   where u_{ik} represents the degree to which an observation k belongs to a cluster i, y_k refers to the k-th data point, v_i represents the cluster center i, c denotes the total number of clusters, m is the fuzzification parameter, $\|y_k - v_i\|$ represents the Euclidean distance between the observation y_k and the cluster center v_i and $\|y_k - v_j\|$ represents the Euclidean distance between the observation y_k and the center of the cluster v_j;

3. **Update Cluster Centers** – update the cluster centers v_i using the membership values and data points, as expressed by equation 2.7.

$$v_i = \frac{\sum_{k=1}^{N} u_{ik}^m y_k}{\sum_{k=1}^{N} u_{ik}^m} \tag{2.7}$$

   where v_i represents the cluster center i, u_{ik} denotes the membership value of the observation k in the cluster i, m is the fuzzification parameter, y_k refers to the k-th data point being considered and N represents the total number of data points. In essence, this formula updates the cluster centers based on the weighted average of data points considering their membership values and the fuzziness parameter m;

4. **Iteration** – repeat the calculation of membership values (step 2) and update of cluster centers (step 3) iteratively until convergence or until reaching a predefined number of iterations.

Unlike k-nearest neighbors (k-NN), FCM assigns a membership value to each data point for every cluster, allowing data points to belong to multiple clusters simultaneously. This flexibility is particularly useful when dealing with complex data structures or datasets with ambiguous or overlapping class boundaries [51].

The FCM algorithm is particularly useful in environments with noise and outliers, as it is less sensitive to these conditions compared to traditional clustering methods. This robustness not only enables smoother transitions between clusters but also enhances interpretability by offering a comprehensive view of how data points relate to each cluster. As a result, fuzzy clustering provides a more detailed representation of the dataset's structure [51].

Research on FCM has highlighted both its strengths and the challenges associated with achieving stable convergence. Although the algorithm is generally easy to implement and can be adapted to various applications, convergence stability is significantly influenced by the initialization of cluster centers. Proper initialization is essential, as it substantially impacts the algorithm's convergence behavior and helps avoid local optima. Converging toward a region with the global optimum improves the algorithm's accuracy and overall performance [55].

Moreover, the number of iterations required for the FCM to converge is affected not only by the initialization of the cluster centers, but also by the size of the dataset. As the dataset size increases, both computational complexity and the number of necessary iterations generally rise [55]. For larger datasets, the FCM algorithm must perform additional calculations to update membership degrees and cluster centroids, often requiring more iterations to reach convergence.

2.4.2 Applicability in Feature Selection

In the context of feature selection, the FCM algorithm stands out as a versatile tool due to its ability to handle overlapping clusters and assign membership values to data points. Unlike traditional hard clustering methods, FCM allows data points to belong to multiple clusters simultaneously, with varying degrees of membership. This flexibility makes it particularly well-suited for feature selection, where complex interactions and overlapping relationships between features are common, challenging conventional approaches [56].

By using the membership values from FCM, features that exhibit strong relationships with multiple clusters can be identified, enabling a more nuanced approach to selecting relevant features. Studies have demonstrated that FCM-based feature selection methods can outperform traditional techniques, particularly when dealing with datasets that contain complex or overlapping data distributions, allowing for more accurate models [57].

Additionally, the FCM, along with its modified variants, has shown significant potential for enhancing feature selection in ML tasks. For example, the MFCMS method, which integrates supervised information into the traditional FCM framework, has been shown to enhance feature selection by identifying more relevant features, ultimately improving the performance of predictive models [58]. As such, FCM-based feature selection methods, including the MFCMS, are particularly effective in domains where feature selection is crucial for managing high-dimensional data, noisy features, and complex relationships among features.

2.5 Conclusion

This chapter explored the fundamental aspects of code smell detection, providing a comprehensive overview of its definition, types, and significance in software development. It discussed various techniques and algorithms used for detecting code smells, highlighting both traditional and modern approaches, with a particular focus on ML approaches. Additionally, the chapter examined the relevance of FCM and its variants for feature selection, emphasizing their ability to handle complex data relationships and iterations. Overall, this chapter provided a solid foundation for understanding the principles and methodologies that underpin the work presented in this dissertation.

Chapter 3

Related Work

3.1 Introduction

In the realm of software engineering and code quality, numerous studies have tackled the detection and mitigation of code smells – a crucial aspect in ensuring the maintainability and robustness of software systems. To effectively pave the way for the research presented in this dissertation, it is crucial to analyze and evaluate the current work in this field, identifying open research gaps and opportunities within the existing literature.

This chapter is divided into four main sections:

- **Comparing and experimenting machine learning techniques for code smell detection (3.2)** – this study provides a overall comparison of different ML techniques, specifying which ones yielded better results in detecting code smells;
- **MLCQ: Industry-Relevant Code Smell Data Set (3.3)** – this section introduces a dataset that can be used by various researchers to detect code smells, highlighting the advantages of this dataset over those already published;
- **Automatic detection of Long Method and God Class code smells through neural source code embeddings (3.4)** – this section provides a summary of a study conducted by Kovačević et al., in which they used the MLCQ dataset and considered it a good option for comparing different implementations in the domain of code smell detection;
- **Feature selection based on a modified fuzzy C-means algorithm with supervision (3.5)** – this section provides an overall introduction to the MFCMS algorithm, revealing the potential of this approach for feature selection.

3.2 Comparing and experimenting machine learning techniques for code smell detection

Fontana et al. conducted a comprehensive study that compared and evaluated a range of ML techniques for detecting code smells, offering valuable insights into their effectiveness. Their approach involved training various ML models using a manually validated dataset compiled by the authors, which included 74 software projects sourced from the Qualitas Corpus, a diverse collection of Java software systems of different sizes and domains. This approach ensures that the ML process does not depend solely on a specific project, thereby enhancing the generalizability of the results.

The researchers investigated four code smells – Feature Envy, Data Class, Long Method, and Large Class – using six ML techniques. These techniques included both regular models and those that leverage boosting, a method that combines weak learners to create a stronger one with improved predictive performance. The selected classification algorithms, such as J48, JRIP, Random Forest, Naive Bayes, Sequential Minimal Optimization (SMO), and Library for Support Vector Machines (LibSVM), are commonly used in software engineering and are capable of producing interpretable models. To ensure the generability of the models and identify the best performer for each code smell, 10-fold cross-validation was employed during training. Additionally, the authors used Grid Search optimization to explore different parameter settings and identify the optimal configuration for each model. Finally, the performance of these models was evaluated using standard performance metrics like accuracy, F-measure, and AUC [21].

The experiments revealed that all methods were effective at detecting the different code smells, achieving an accuracy above 90% for each of them. Random Forest, in particular, consistently produced the best results across all four training datasets, making it a valuable option for this type of detection task. Interestingly, the study found that the boosting technique offered no significant advantage and sometimes even led to performance degradation compared to non-boosted models [21].

However, the authors identified two potential limitations of their study. Firstly, the amount of data used to train the models might have been limited. Secondly, they did not incorporate a feature selection algorithm, which helps identify the most relevant features and could potentially improve model performance. The authors noted that feature selection is an active area of research and highlighted the potential benefits of including such an algorithm in future studies [21].

3.3 MLCQ: Industry-Relevant Code Smell Data Set

Madeyski et al. collaborated with professional software developers to create a dataset called Madeyski Lewowski Code Quest (MLCQ). They recognized a potential limitation in existing datasets, which were primarily obtained from a small number of outdated projects by a single person whose experience is unknown. To address this limitation, they partnered with professional software developers to review code samples and identify possible bad smells [59].

The dataset contains 14739 code reviews made by 26 developers with professional experience across four different code smells – Blob Class (also known as God Class), Data Class, Feature Envy and Long Method – totaling 4770 code samples, with 4129 from industry-relevant projects. These code samples were generated and extracted from Java projects selected from GitHub, and then reviewed by the corresponding developers. The order of the reviews was determined by an acquisition tool used by the authors, and they were asked to classify the samples into four severity samples: critical, major, minor, and none. However, developers could skip the review if they were uncertain of the result.

Madeyski et al. highlighted several key ways in which their dataset stands out. Firstly, all reviewers are actively employed in the software development industry. Secondly, most samples were collected by developers who are neither students nor researchers. Additionally, the dataset provides unique and detailed insights into the professional and academic backgrounds of the reviewers. This last feature is particularly noteworthy, as it enables researchers to trace each code review back to the developer's background. According to the authors, this distinctive aspect could open up new research opportunities [59].

Regarding the dataset, the samples include the type of code smell, its severity, and the exact location in the source code. However, software metrics are not included in the samples, as they can be calculated using different tools. The dataset is publicly available and comes with a reproducer package to facilitate its use.

The authors concluded that this collection of data can be used primarily for detecting code smells. However, it could also serve as an auxiliary dataset for defect prediction and other research fields [59].

3.4 Automatic detection of Long Method and God Class code smells through neural source code embeddings

Kovačević et al. evaluated the effectiveness of traditional code metrics versus code embeddings in detecting God Class and Long Method smells. They compared the performance of ML models trained on these two types of feature representations, using traditional code metrics and pre-trained neural source code embeddings like CuBERT. According to the authors, this study was the first to use such embeddings for detecting code smells and the first to perform classification on the MLCQ dataset.

The authors compared different detection techniques on the same large-scale dataset, providing a consistent comparison baseline. To ensure their study is replicable and to facilitate further comparisons with other research, they provided a replication package. This package includes all variants of the MLCQ dataset used, such as the dataset with all extracted software metrics. The authors concluded that the MLCQ dataset is a suitable option for comparing different implementations due to its large size and public availability, making it ideal for code smell detection. They addressed the problem using binary classification, labeling samples as `smelly` if their severity level was minor, major, or critical, and as `non-smelly` if there was no severity. In this setup, the `smelly` class was considered positive, while the `non-smelly` class was considered negative [60].

Kovačević et al. used three main detection methods to compare their advantages and disadvantages. The first method employed heuristic-based approaches extracted from a literature review to classify code smells. However, the authors found that these approaches heavily depend on the defined thresholds and rules. The second method involved training a random forest model using software metrics extracted from CKTool and Repository Miner. The third

method converted each code sample into an embedding using pre-trained neural source code embeddings and fed this embedding into a leaning model along with their labels [60].

The authors concluded that ML-based detectors have a significant advantage over heuristic ones. Unlike heuristics, which rely on the hard-coded thresholds that might need adjustment depending on the tool, ML-based methods use source code metrics or embeddings as indicators. They found that ML models using embeddings performed slightly better than those trained with software metrics. Additionally, they observed and concluded that the process of extracting software metrics can be complex and time-consuming [60].

3.5 Feature selection based on a modified fuzzy C-means algorithm with supervision

Francesco Marcelloni introduced a feature selection method based on a Modified Fuzzy C-Means algorithm with supervision (MFCMS). This method enhances traditional unsupervised FCM clustering by incorporating labeled patterns. The labeled data helps improve the accuracy of cluster modeling and the selection of features that are more effective at distinguishing between clusters. The main idea is to identify features with low variance within clusters and high discriminative power between clusters. A new index is proposed to quantify the discrimination capability of each feature, where higher values imply a greater ability to differentiate clusters [58].

Similarly to the traditional FCM implementation, the MFCMS algorithm operates by minimizing an objective function. However, MFCMS differs by incorporating both supervised and unsupervised components. The algorithm updates iteratively the membership values, cluster prototypes, and features weights, where higher weights are assigned to features with lower variance in a cluster. Furthermore, this approach evaluates the minimum distance between each cluster's centroid and the centroids of other clusters. This helps identify features that are particularly effective at separating clusters. The features are selected if their corresponding discrimination index exceeds a specified threshold [58].

The method was tested on various real-world datasets from the University of California–Irvine (UCI) repository using the k-NN algorithm for classification. MFCMS was compared with other feature selection methods, and consistently showed superior generalization accuracy across all datasets. Furthermore, it identified a significantly smaller subset of features without compromising classification performance. Experimental results confirmed that MFCMS outperformed other methods in both accuracy and feature reduction.

The authors concluded that MFCMS offers an effective solution for feature selection. Moreover, the algorithm successfully selects relevant features, improving classification performance while reducing the dimensionality of the data.

3.6 Conclusion

This chapter explored the current landscape of research related to code smell detection and feature selection techniques using FCM clustering. Various methodologies, datasets, and algorithms that have influenced the field of code smell detection were reviewed, highlighting their advantages and contributions to the area of study.

Furthermore, the potential of MFCMS as a feature selection mechanism was identified. The MLCQ dataset and the commonly used Random Forest model in code smell detection were also examined, providing insights into the types of data and learning algorithms that effectively support this research. In summary, understanding these approaches, the dataset, and the Random Forest model establishes a strong foundation for the methodology applied in this dissertation and will guide the direction of the study moving forward.

Chapter 4

Implementation Details

4.1 Introduction

In this chapter, we delve into the specific methodologies and techniques employed in the implementation of our proposed approach for code smell detection. This section outlines the practical aspects of our research, including the choices made regarding our methodology, the algorithms selected, and the challenges encountered during the implementation phase. These details are crucial for understanding how our approach was constructed and optimized to achieve effective results.

This chapter is divided into seven main sections:

- **Methodology (4.2)** – this section outlines the main methodology applied in this dissertation, outlining each one of the phases that comprises it;
- **Dataset (4.3)** – this section describes the selected dataset and explains how it was preprocessed for the experiments conducted;
- **Feature Selection Mechanism (4.4)** – this section provides a detailed explanation of the implementation process for the feature selection mechanism. It outlines each phase step by step, explaining the necessary procedures to achieve the desired outcome;
- **ML Model Implementation (4.5)** – this section explains the rationale for selecting a particular ML model for our approach and details the process behind its implementation and optimization;
- **Evaluation of the Experiments (4.6)** – this section discusses the performance metrics used to evaluate our approach and explains how they were implemented;
- **Computation Environment (4.7)** – this section describes the computational environment in which all experiments were conducted, including the virtual environment used;
- **Implementation Challenges (4.8)** – This section covers the challenges faced during the implementation process and explains the solutions applied to overcome them.

4.2 Methodology

The primary objective of this study was to implement a feature selection mechanism to enhance the effectiveness of code smells detection. To accomplish this, we adopted a multiphase approach, which was divided in four key stages:

1. **Dataset** – in this phase, we gathered a dataset consisting of code samples from various software projects, both containing instances of the long method smell and free of them. The dataset was carefully selected for its diverse composition of software projects, all drawn from real-world industrial scenarios. This ensures that the dataset reflects practical coding environments and includes a variety of project types, sizes, and complexity levels, making it well-suited for evaluating code smell detection in real-world applications. Additionally, the dataset was preprocessed to guarantee consistency and enhance its suitability for analysis;

2. **Feature Selection Mechanism** – a feature selection mechanism was implemented to identify and select relevant features, reducing the dimensionality of the dataset while preserving crucial information for code smell detection. This was achieved through the application of the MFCMS method, complemented by a series of optimization techniques, to enhance the efficiency and effectiveness of the feature selection process. These optimizations aimed to identify the optimal parameters for the algorithm execution, thereby improving the overall performance of the selection mechanism;

3. **ML Model Implementation** – the third phase involved training a ML model using the selected features to accurately identify the presence of code smells. Additionally, this phase included training the same model on the entire dataset to facilitate a comparison between the two approaches. Furthermore, the hyperparameters of the ML model were optimized to ensure optimal performance in the detection process;

4. **Evaluation of the Experiments** – in the final phase, the effectiveness of the detection method was assessed through a set of performance metrics. This evaluation allowed us to determine whether the selected features contributed to an improved detection process and to evaluate the potential of the selection method for future research.

The flowchart shown in Figure 4.1 provides a comprehensive overview of the methodology we followed throughout the experiment, detailing each step from the initial dataset preprocessing to the final evaluation phase. Each step in the previous list corresponds to a specific part of the flowchart, exemplifying the progression of the experiment. This flowchart offers a clear and structured representation of the entire workflow, emphasizing the iterative nature of both the clustering and feature selection phases. In the following sections, we will delve into the specific implementation details of each phase, as illustrated in Figure 4.1. It is important to note that the preprocessing phase differs depending on the specific experiment being conducted. Therefore, the phase is presented in a generalized form in the flowchart shown in Figure 4.1. Section 4.3 will offer a more detailed explanation of how the dataset was prepared for each experiment.

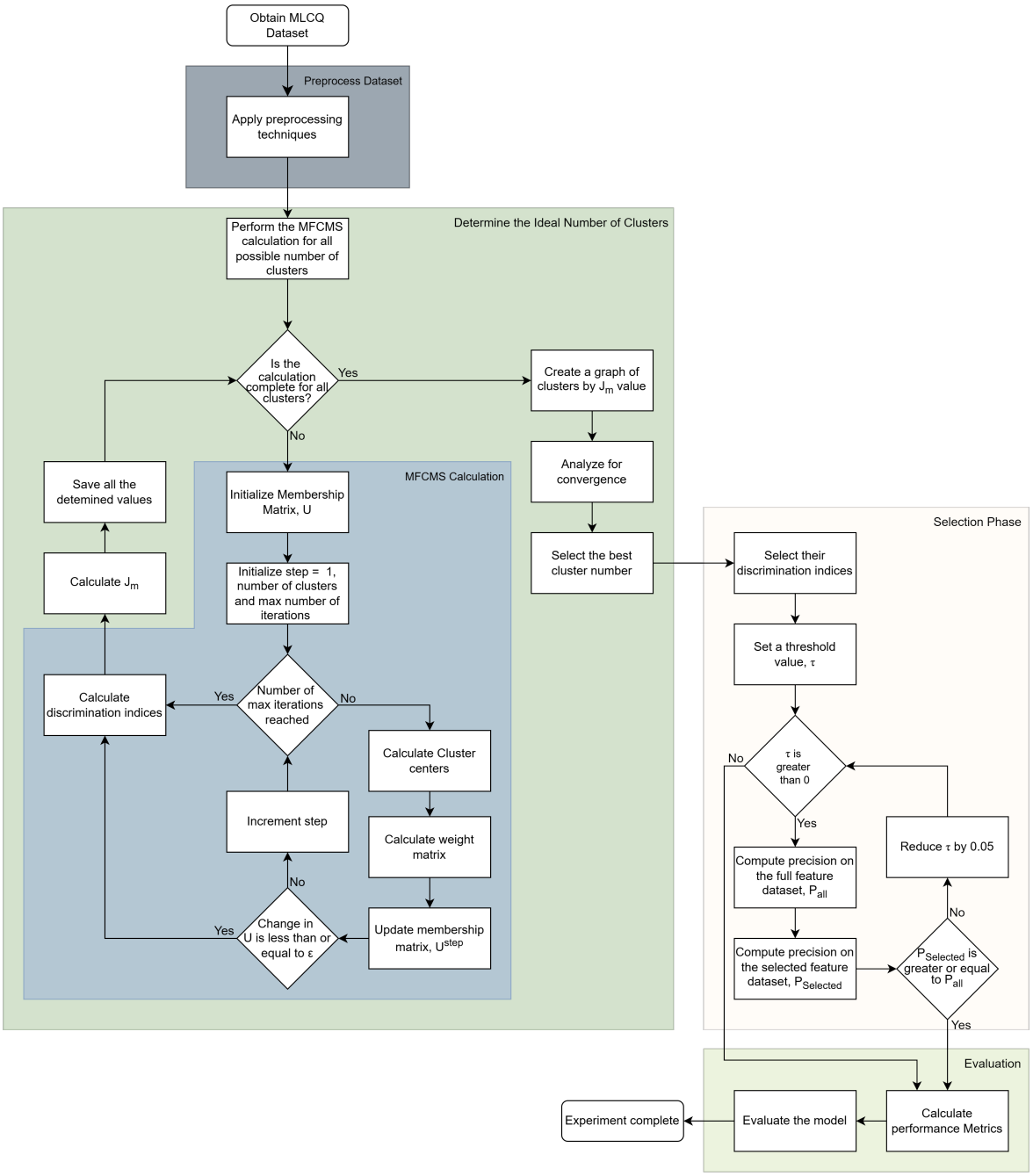

Figure 4.1: Flowchart of the Methodology Execution

4.3 Dataset

The dataset chosen for the implementation of this detection approach was the MLCQ dataset introduce in section 3.3.

According to various literature reviews and at the time of developing this dissertation, several datasets were no longer publicly available. This lack of access hindered the selection of an appropriate dataset for the research task. In many recent studies, authors have opted to create their own datasets, making the situation more challenging, as most do not share these datasets or provide information on how they were developed. Therefore, the MLCQ dataset

stands out as one of the few publicly available datasets where the authors clearly explain its creation process and the rationale behind its development, as previously mentioned in Section 3.3.

As noted by the authors of [59], this dataset was developed by professional software developers, highlighting the reviewers' expertise in identifying code smells. This aligns with the notion articulated by Fowler and Beck that detecting code smells is a process based on human intuition. It emphasizes that samples created by experts in the field could enhance the detection process using ML, as these samples are highly meaningful, thereby highlighting the potential of this dataset for effective detection.

However, this original dataset does not include any software metrics for the detection mechanism proposed in this dissertation. Since extracting and identifying the best software metrics is an extremely difficult and time-consuming task, as discussed in the section 3.4, we will be using the dataset with 26 software metrics extracted by the authors of [60] for method-level detection, which is already serialized for binary classification. Furthermore, the authors have pre-identified the instances to be used for both the training and test sets, ensuring consistent and reliable evaluation throughout the use of this dataset. The table 4.1 presents the distribution of positive and negative instances in the training and test sets, totaling to a total of 2408 instances used across all data sets.

	Training Set	**Test set**
Negative Instances	1703	428
True Instances	223	54
Total	1926	482

Table 4.1: Distribution of Instances in Training and Test Sets

For the first experiment, the dataset was normalized using the Min-max normalization technique. Normalizing the input features enhances the performance of many machine learning algorithms by ensuring that all features are on a comparable scale, as many algorithms struggle with unscaled data. This step was crucial to maximize the overall performance of our approach, ensuring optimal conditions for both model training and evaluation.

Min-max normalization is one of the most common approaches to normalize data. This method scales each feature to a specified range, typically between zero and one, by adjusting the values according to the feature's minimum and maximum values [61]. This process is illustrated by the Equation 4.1.

$$x_{\text{scaled}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \tag{4.1}$$

where x is the original feature value, and $x_{\min}$ and $x_{\max}$ are the minimum and maximum values of that feature.

This normalization technique was implemented using the preprocessing module of the `scikit-learn` library, which offers a convenient way to apply various data preprocessing methods. We

efficiently scaled the features of the entire dataset to the range $[0, 1]$ using the built-in `MinMaxScaler` function.

Additionally, due to the potential for high correlations among the data, which could negatively impact model performance and interpretability, we conducted two experiments. In the first, we used the data formatted as described earlier. In the second, we applied a different data formatting approach using Principal Component Analysis (PCA).

PCA is a powerful statistical technique commonly used for dimensionality reduction in data science and ML. The PCA transforms a dataset with potentially correlated features into a new set of uncorrelated variables known as principal components. These components are linear combinations of the original features and are ordered by the amount of variance they capture, with the first component accounting for the most variance and each subsequent one capturing progressively less [62]. The main goal of this technique is to simplify complex datasets, reduce dimensionality and preserve the most critical information, making it a valuable tool for data compression and noise reduction [63].

PCA is often associated with dimensionality reduction, but it can also be used without reducing the number of components. In this case, the algorithm still transforms the original correlated components into a new set of uncorrelated principal components, addressing the known issue of multicollinearity. The removal of the correlation between features will improve the stability of a ML algorithm, leading to more reliable and interpretable models.

As the MLCQ dataset contains potentially high correlations among features, we applied PCA to reduce these correlations without reducing the number of features, keeping them for use with our feature selection approach. By mitigating correlations and minimizing potential noise, this step aims to improve the performance and reliability of our approach.

To apply this approach, we used the MLCQ dataset in its original state, as provided by the authors of [60]. We noted that the dataset contains two categorical features, `constructor` and `hasJavaDoc`. Consequently, we separated these features before transforming the dataset using the PCA algorithm. After separating the categorical variables, we transformed the remaining 24 features using the `PCA` function available in the `decomposition` module of the `scikit-learn` library. Finally, we normalized the transformed dataset with the `MinMaxScaler` function from the same library and reintroduced the categorical variables to the dataset. This process allowed us to obtain a dataset with reduced correlation compared to the one used in the first experiment.

4.4 Feature Selection Mechanism

As mentioned in 3.5, feature selection using MFCMS shows promise as a mechanism for identifying the most important features in the dataset. This approach reduces the overall dimensionality of the dataset, thereby enhancing the performance of the ML model. The implementation of this approach encompasses three main phases:

1. **MFCMS calculation** – this phase involves implementing all the mathematical functions necessary for executing the MFCMS algorithm. These functions enable the algorithm to achieve convergence during execution, which is crucial for determining the discrimination indices used in the selection phase. Further details about this process is provided in section 4.4.1;

2. **Determine the Ideal Number of Clusters** – this phase is responsible for identifying the optimal number of clusters for the clustering algorithm, which is essential for ensuring accurate execution and to provide meaningful insights. Achieving the correct clustering structure is critical for the overall effectiveness of the feature selection process. Further details about the implementation of this process is present in section 4.4.2;

3. **Selection Phase** – this phase is responsible for selecting the most important features. The goal is to achieve better results by reducing the dataset's dimensionality, focusing on the features that have the greatest impact on the performance of a particular ML model. This process is explained in detail in section 4.4.3, where the algorithm involved in this task is described.

4.4.1 MFCMS Calculation

The algorithm of MFCMS is built upon the FCM algorithm, detailed in section 2.4, with modification to incorporate the supervised aspect of the MFCMS. This involves leveraging instances with known outcomes, specifically the presence or absence of long method smells. This factor is one of the key points that distinguish both algorithms, as it allows MFCMS to refine its clustering process based on labeled data.

In contrast to traditional clustering algorithms where membership values are either zero or one, fuzzy clustering methods, including both FCM and MFCMS, assign membership values on a continuous scale within the interval $[0, 1]$. This enables greater flexibility and improved handling of overlapping clusters, as discussed in 2.4.

Similarly to the FCM algorithm, the MFCMS clustering optimizes an objective function to achieve an effective partitioning of the dataset. The addition of labeled patterns serves as reference points that shape cluster boundaries, thereby enhancing accuracy. By incorporating these labeled instances, MFCMS refines its objective function to better distinguish clusters based on known outcomes, resulting in a more informed data partitioning [58]. Equation 4.2 details the transformation applied to the objective function in this algorithm.

$$J_{m,\alpha}(U, V, W) = \sum_{i=1}^{C}\sum_{k=1}^{N} u_{i,k}^{m}\tilde{d}_i^2(x_k, V_i) + \alpha \sum_{i=1}^{C}\sum_{k=1}^{N}(u_{i,k} - l_{i,k}b_k)^m \tilde{d}_i^2(x_k, V_i) \tag{4.2}$$

where α balances the unsupervised and supervised components, C is the number of clusters, N the number of instances in the dataset, $\tilde{d}_i^2$ represents the weighted distance, m the fuzzification

parameter, u the membership of a cluster center to a specific instance, V the cluster center, b the binary vector and l the membership of the labeled patterns.

The optimization of the objective function in equation 4.2 is carried out through an iterative process. At each step, the membership values $u_{i,k}$, the cluster centers V_i and the weighted matrix $\tilde{z}_{i,t}$ are updated to minimize the objective function. This functions can be explained as the following:

- **Update Membership values** – The membership values represent the degree to which an instance belongs to a cluster. These values are updated at each iteration based on the weighted distance between the instance and all cluster centers, using the equation 4.3.

$$u_{r,k} = \frac{1}{1+\alpha}\left[\frac{1+\alpha(1-b_k\sum_{i=1}^{C} l_{i,k})}{\sum_{i=1}^{C}\frac{\tilde{d}_r^2(x_k,V_r)+\delta}{\tilde{d}_i^2(x_k,V_i)+\delta}} + \alpha l_{r,k} b_k\right], \forall r \in [1..C] \text{ and } \forall k \in [1..N], \quad (4.3)$$

  where α balances the unsupervised and supervised components, C is the number of clusters, $\tilde{d}_i^2$ represents the weighted distance, u the membership of a cluster center to a specific instance, V the cluster center, x_k the k-th data point, b the binary vector, l the membership of the labeled patterns, N the number of data points and δ a small positive real number;

- **Update Cluster centers** – To ensure that the cluster centers accurately represent the data points within each cluster, they are recalculated at every iteration using equation 4.4. This iterative process allows the algorithm to refine the cluster boundaries and improve the overall clustering quality.

$$V_{s,t} = \frac{\sum_{k=1}^{N} x_{k,t}\left(u_{s,k}^m + \alpha\left(u_{s,k} - l_{s,k}b_k\right)^m\right)}{\sum_{k=1}^{N}\left(u_{s,k}^m + \alpha\left(u_{s,k} - l_{s,k}b_k\right)^m\right)}, \forall s \in [1..C] \text{ and } \forall t \in [1..F], \quad (4.4)$$

  where α balances the unsupervised and supervised components, C is the number of clusters, u the membership of a cluster center to a specific instance, b the binary vector, l the membership of the labeled patterns, m the fuzzification parameter, $x_{k,t}$ the t-th feature of the k-th data point, N the total number of data points and F the total number of features;

- **Update weight matrix** – The weight matrix is updated at each iteration to adjust the contribution of each feature to the clustering process using the equation 4.5. This allows to assign more importance to the features that contributes significantly to distinguish

clusters and reduce the influence of less relevant features.

$$z_{i,t} = \frac{F}{\sum_{f=1}^{F} \left[\frac{\sum_{k=1}^{N} u_{i,k}^{m}(x_{k,t}-V_{i,t})^2 + \alpha \sum_{k=1}^{N}(u_{i,k}-l_{i,k}b_k)^m (x_{k,t}-V_{i,t})^2}{\sum_{k=1}^{N} u_{i,k}^{m}(x_{k,f}-V_{i,f})^2 + \alpha \sum_{k=1}^{N}(u_{i,k}-l_{i,k}b_k)^m (x_{k,f}-V_{i,f})^2} \right]^{\frac{1}{p-1}}}, \tag{4.5}$$

$$\forall i \in [1..C] \text{ and } \forall t \in [1..F],$$

where F is the total number of features, C the total number of clusters, N the total number of data points, α balances the unsupervised and supervised components, x is a data point, u the membership of a cluster center to a specific instance, b the binary vector, l the membership of the labeled patterns, m the fuzzification parameter, V the cluster center and p the factor value.

Additionally, it is important to note that, unlike the traditional FCM implementation, the distance used in these operations is the weighted distance, calculated as shown in equation 4.6. This represents a key distinction from the conventional FCM approach, which typically relies on Euclidean distance.

$$\tilde{d}_i^2(x_k, V_i) = \sum_{f=1}^{F} (x_{k,f} - V_{i,f})^2 z_{i,f}^p, \tag{4.6}$$

where each $z_{i,f}$ represents the weight associated with the component f with respect to the cluster i, x is the data point and V the cluster center.

Taking this into account, the pseudocode outline in Algorithm 2 served as the basis for the implementation of the MFCMS algorithm. Each step of the algorithm was implemented using Python programming language, leveraging libraries such as `numpy` to efficiently handle mathematical operations and matrix computations. The use of `numpy` facilitated the vectorized implementations of core tasks like updating membership values, calculating cluster centers and evaluating the convergence criterion, resulting in improved performance of the code. Furthermore, `numpy` was used to efficiently initialize the matrix U^0 randomly with the `rand` function from the module `random` of the library, with the seed set to a consistent value across experiments to maintain reproducibility.

Algorithm 2 MFCMS algorithm

Fix the number of clusters C, the factor parameter p, the fuzzification parameter $m = 2$ and the maximum number of iterations $max\_iterations$
Initialize, $step = 1$
Initialize, membership matrix U^0 randomly
while $step \leq max\_iterations$ **do**
  Calculate the cluster centers V^{step} with equation 4.4, using U^{step}
  Calculate weight matrix Z^{step} with equation 4.5, using V^{step} and U^{step}
  Update membership matrix U^{step} with equation 4.3, using V^{step} and Z^{step}
  if Compare using matrix norm: $||U^{step} - U^{step-1}|| \leq \varepsilon$: Typically $\varepsilon = 1 \times 10^{-5}$ **then**
    Stop
  else
    Update $step = step + 1$

After the execution of the MFCMS algorithm, it was necessary to determine the discrimination indices, which are the values that distinctly classify each data point into one of the fuzzy clusters. This means that a feature with higher discrimination index is more important in distinguishing between clusters because it contributes more to the separation. Therefore, discrimination indices can also serve as a guide for feature selection, helping identify which features are most critical to retain in the model. The equation 4.7 shows how the discrimination indices were calculated.

$$D_{i,f} = \frac{\left(\frac{d_{f,i,\bar{j}}(V_i, V_{\bar{j}_f})}{\sum_{f=1}^{F} d_{f,i,\bar{j}}(V_i, V_{\bar{j}_f})} \cdot F + z_{i,f} \right)}{2}, \tag{4.7}$$

where $d_{f,i\bar{j}}(V_i, V_{\bar{j}_f}) = \min_{j \neq i} d_{f,ij}(V_i, V_j)$ and $z_{i,f}$ is the weight associated with feature f in cluster i.

To execute the experiment, it was essential to define several key variables that significantly influence the outcome of the MFCMS algorithm. The primary variables included:

- C – the number of clusters to be used in the algorithm. In this study, multiple values were tested to determine the optimal number of clusters for achieving the best performance, which will be discussed in further detail in Section 4.4.2;
- α – the value α was set to 1 to incorporate the supervised aspect of the algorithm. According to the authors, to execute the algorithm in an unsupervised manner, α should simply be set to 0;
- m – the value m is the fuzzification parameter that controls the degree of fuzziness. It was set to 2, a typical value used in FCM implementations, and was also the value used by the authors of [58]. This fuzzification parameter determines the extent to which a data point can have partial membership in multiple clusters simultaneously;
- p – the value p can be described as the factor that captures the distribution and geometric size of the points within a cluster, considering the shape and volume of the

area covering the cluster data. This factor can be adjusted to influence the clustering technique's behavior, allowing more flexibility in modeling clusters of different shapes and volumes, rather than constraining them [58]. In [58], the author concluded through experimentation that a value in the interval $[2, 3]$ provided a good classification performance with fast convergence times. Based on these findings, we set p to 2.3, as suggested by the author, since this value yielded promising results in his work.

- ε – this value represents the admissible error for the overall execution of the MFCMS algorithm, determining when the algorithm stops. For this purpose, we set ε to 1×10^{-5} following the author's recommendation [58];

- δ – this value was added to prevent division by zero during the calculation of the membership degree of a cluster center for a specific element. This scenario is possible if a cluster centroid coincides with a specific data point [58]. Consequently, we set the value of δ to 1×10^{-7};

- *seed* – this value was set to 42 to ensure that the initialization of the membership matrix remained consistent across multiple experiments. This consistency is crucial for enabling fair comparisons of the results and assessing the performance of the algorithm across different runs;

- *max_iterations* – this value represents the maximum number of iterations for the MFCMS algorithm. It was set to 10000 iterations.

- b – this value represents a binary vector, where its length corresponds to the number of data points used. Each element in the vector indicates whether a data point is labeled or not, as described in Equation 4.8. A value of 1 signifies that the data point is labeled, while 0 indicates that it is unlabeled. This vector is critical for distinguishing between supervised and unsupervised data points in the algorithm.

$$b_k = \begin{cases} 1 & \text{if } x_k \text{ is labeled} \\ 0 & \text{if } x_k \text{ is not labeled} \end{cases}, \tag{4.8}$$

  where $k = 1...N$, N is the number of data points and x_k the k-th data point;

- L – this value represents the membership of the labeled patterns, where $L = [l_{i,k}]$ with $i = 1...C, k = 1...N$. Here C is the number of clusters and N the total number of data points. The membership of the labeled patterns guides the clustering process by enforcing partial supervision, ensuring labeled data points adhere more strictly to their corresponding clusters, thus enhancing the formation of clusters that align with known labels. The membership of the labeled patterns was determined using equation 4.9.

$$l_{i,k} = \begin{cases} 1 & \text{if } c(x_k) = i \\ 0 & \text{otherwise} \end{cases}, \tag{4.9}$$

  where $c(x_k)$ is the class of x_k.

4.4.2 Determine the Ideal Number of Clusters

Determining the ideal number of clusters for the MFCMS algorithm is crucial for effective feature selection, as it ensures that the clusters accurately represent meaningful patterns in the data. Selecting the appropriate number of clusters helps uncover the underlying structure, supports the selection of relevant features with minimal redundancy, and enhances both classification performance and convergence efficiency. This balance allows the MFCMS algorithm to identify the features that most effectively differentiate the clusters, ensuring that the selected features are the most informative for the detection approach. This task was carried out using Cauchy's criterion for convergence.

The Cauchy convergence criterion states that a sequence x_n converges to a limit X if, for every positive number $\varepsilon > 0$, there exists an integer K such that for all indices n and m greater than K, the following condition holds:

$$|x_n - x_m| < \varepsilon.$$

This means that the terms of the sequence become arbitrarily close to each other as n and m increase. In other words, as the indices of the sequence grow larger, the values x_n and x_m approach one another, indicating that they are converging towards the same limit X [64].

In the context of the MFCMS algorithm, applying Cauchy's criterion enabled us to determine the ideal number of clusters to use, thereby enhancing the overall feature selection process. This process was performed taking into account the algorithm 3.

Algorithm 3 Applicability of the Cauchy's criterion for convergence

1: **for** each cluster c_k in $C = \{2, ..., c_n\}$ **do**
2: Perform the MFCMS algorithm with algorithm 2
3: Determine the value of the objective function, J_m with equation 2.5
4: Save the J_m and number of cluster values
5: Determine a graph that represents the relationship between the number of clusters and corresponding J_m values to see the convergence pattern
6: By analyzing the graph choose the correct number of clusters

The J_m function outlined in the algorithm 3 corresponds to the objective function used in the MFCMS implementation. Although the algorithm is initially executed with the regularization parameter $\alpha = 1$ to incorporate supervised information, the final J_m value is calculated using only the unsupervised part of the function (i.e., by setting $\alpha = 0$), as shown in equation 4.10. This approach ensures that the final clustering evaluation reflects the natural structure of the data, free from bias introduced by the supervised component, and enables a clearer and more consistent comparison across different cluster configurations. Consequently, the J_m function, in this context, exclusively involves the unsupervised clustering process, without any influence from the supervision or label-based constraints.

$$J_{m,\alpha}(U,V,W) = \sum_{i=1}^{C}\sum_{k=1}^{N} u_{i,k}^{m}\tilde{d}_i^2(x_k,V_i) \tag{4.10}$$

where C is the number of clusters, N the number of instances in the dataset, $\tilde{d}_i^2$ represents the weighted distance, m the fuzzification parameter, u the membership of a cluster center to a specific instance and V the cluster center.

After obtaining all the J_m values, we were able to generate a graph that illustrates the relationship between the number of clusters and the corresponding J_m values. This visualization was accomplished using `Matplotlib` library, which enabled us to create a clear and informative plot. The graph provides valuable insights into how the objective function changes with different cluster numbers, helping us identify the optimal number of clusters for our analysis. By analyzing the plot and having in consideration the Cauchy's convergence criterion, we can easily observe trends and make informed decisions about the number of clusters that best captures the underlying structure of the data. The optimal number of clusters is selected based on the point just before the convergence trend, where the objective function stabilizes, indicating that adding more clusters does not significantly improve the data fit. This method helps identify the elbow point – the point at which further increases in the number of clusters result in diminishing returns. By selecting the optimal cluster count, we balance capturing the data's inherent structure while avoiding overfitting, leading to a more interpretable and effective clustering solution.

In the end, once the best number of clusters is found, the corresponding cluster configuration is selected to perform the MFCMS algorithm and applied within the feature selection mechanism as well. This final configuration is expected to contribute to improved results in the feature selection process, as it theoretically captures the underlying structure of the data more effectively.

4.4.3 Selection Phase

This phase focuses on the selection of the most important features of the dataset that will contribute to a better outcome of the overall learning model.

The key idea for determining which features are important is based on the values of the discrimination indices. These values reflect how well each feature in the dataset distinguishes between clusters. A higher value of a discrimination index indicates that a feature plays a more significant role in differentiating clusters. These values were previously calculated using equation 4.7 during the execution of the MFCMS algorithm.

The features are selected based on a predefined threshold τ, where only those with discrimination indices for each cluster satisfying $D_{i,f} > \tau$ are considered important. Since different clusters may select different features, the final set of selected features is determined by taking the union of the features selected across all clusters. This approach ensures that all relevant

features are included. This process is crucial, as it forms a key step in the feature selection algorithm, which is demonstrated in Algorithm 4.

Algorithm 4 Selection algorithm

1: Set the τ to an appropriate value
2: **while** $\tau > 0$ **do**
3: Select the features based on the predefined threshold τ
4: Compute the precision P_{all} for the model by applying the employed learning model introduced in section 4.5 on data characterized by all the features
5: Compute the precision $P_{selected}$ for the model by applying the employed learning model introduced in section 4.5 on data characterized only by the selected features
6: **if** $P_{selected} \geq P_{all}$ **then**
7: Stop
8: **else**
9: Update $\tau = \tau - 0.05$

As seen in algorithm 4, the third phase described in the methodology in section 4.2 was implemented alongside the selection phase. This was essential for computing the precision of the learning model and for evaluating which features contributed the most to its overall performance. This process helped in identifying and selecting the most important features.

4.5 ML Model Implementation

According to recent literature reviews and the works mentioned in section 3.2 and 3.4, Random Forest is one of the most widely used and effective ML models for the detection task, consistently outperforming most individual models in terms of accuracy [65]. Therefore, it was the chosen one for the experiments conducted in this dissertation due to its proven track record and potential for high performance.

The construction of the model for this experiment includes not only the model itself, but also incorporates techniques such as cross-validation. This technique involves dividing the dataset into multiple subsets, or folds, and training the model on most of these subsets while performing validation on the remaining one. This process is repeated multiple times, based on the number of folds defined, with a different subset used as the validation set each time. The primary goal of this approach is to prevent overfitting by providing a more reliable estimate of the model's performance on unseen data [66].

The ML model was implemented using the `scikit-learn` library, specifically the `ensemble` package, which includes the `RandomForestClassifier` module. Following this, the Grid Search optimization was performed, as detailed in section 4.5.1. This optimization process was crucial to identify the optimal model configuration, resulting in significantly improved performance. This ensures that the Random Forest model generalizes well to new data and performs optimally on the available data.

To ensure reliable and fair model training and evaluation, the `KFold` module from `scikit-learn` was employed. This tool enabled the splitting of the dataset into 10 equal parts, or folds.

Firstly, `shuffle` was set to `True`. This shuffling of the data prevents any patterns or biases from influencing the results. For example, if the data were organized based on a categorical feature such as the presence or absence of a particular smell, shuffling could ensure that each fold receives a balanced mix of examples. Additionally, the `random_state` parameter was set to `42`. This guarantees consistent randomness in fold selection across different experiment runs, ensuring reproducibility [67]. Repeating the experiment should yield the same results each time, enhancing trust in the findings.

4.5.1 Hyperparameter Optimization Method

The chosen method for hyperparameter optimization for the Random Forest model was Grid Search. As discussed in 2.3.2.1, this method systematically explores a predefined set of hyperparameters to identify the best combination of values. Its thoroughness and simplicity make Grid Search an ideal choice for this dissertation.

The Random Forest model has many hyperparameters. However, attempting to optimize all of them using Grid Search would significantly increase the computational expense. Therefore, it was necessary to select a subset of hyperparameters that are commonly known to improve predictions, as the main objective is to optimize those that significantly impact the model's performance [68]. With this in mind, the chosen for optimization were:

- `n_estimators` – this hyperparameter controls the number of decision trees in the forest. Increasing this value can improve the accuracy of the model by allowing it to capture more complex patterns in the data;
- `max_depth` – this determines the maximum level of each tree in the Random Forest model. A deeper tree can capture more information about the training data, but may not generalize well to test data;
- `min_samples_split` – this hyperparameter controls the minimum number of samples required to split an internal node of each tree. This helps to prevent overfitting by ensuring that each tree does not become too complex.

The Grid Search optimization technique was implemented using the `GridSearchCV` module from the `scikit-learn` library. To perform this operation, it was necessary to define the `estimator`, which is the learning model being used. Additionally, a dictionary called `param_grid` must be set, where each key correspond to the hyperparameter name and maps it to the corresponding interval of data to be tested. The `cv` parameter was set to perform 10-fold cross-validation across the grid search optimization procedure. Finally, the `scoring` parameter, which controls the strategy for evaluating the performance of the cross-validated model on the validation set, was set to `precision`, as the goal is to focus on improving the model's positive predictions.

Table 4.2 presents the range of values used for each hyperparameter used in the grid search. These ranges were selected based on established practices across various grid search implementations for hyperparameter optimization [69, 70]. To manage computational costs and

time, the range was deliberately limited, as increasing it would considerably raise the computational load.

Hyperparameter	Values Range
n_estimators	[50, 100, 150, 200, 250, 300, 350]
max_depth	[5,10, 15, None]
min_samples_split	[2, 4, 6, 8, 10]

Table 4.2: Hyperparameters values.

4.6 Evaluation of the Experiments

The evaluation of the main two experiments was conducted in compliance with the guidelines outlined in section 2.3.4. We utilized accuracy, precision recall, F1-score and AUC score to gain meaningful insights into the model's performance. These metrics offer a comprehensive evaluation, where accuracy measures overall correctness, precision evaluates the proportion of relevant positive predictions, recall assesses the model's ability to capture true positives, and F1-score balances precision, recall to account for potential class imbalances, and the AUC score reflects the model's ability to distinguish between classes. This multi-metric approach ensures a robust and reliable assessment of our model's effectiveness. Additionally, we employed the confusion matrix, which provides a detailed breakdown of TP, TN, FP and FN, giving deeper insights into the model's classification patterns.

To implement this evaluation procedure, we meticulously prepared the training and test sets for each experiment, ensuring that the evaluation accurately assessed the model's performance on unseen data. After training each model, we generated predictions and used these results to compute the relevant metrics, allowing for a thorough analysis of the outcomes. Each metric was calculated using dedicated functions from the `metrics` module of the `scikit-learn` library, which ensured reliable and standardized calculations for accuracy, precision, recall, F1-score and AUC score.

These metrics not only served as means to evaluate the model's performance but also enabled us to compare the two experimental approaches. As a result, we were able to determine which method was more effective in detecting the long method smell.

4.7 Computation Environment

All these experiments were conducted on a machine equipped with an AMD Ryzen 5 7600X CPU, 32GB of RAM and an NVIDIA GeForce RTX 3060. The operating system used was Windows 10 Pro, and the experiments were implemented in Python 3.10.12. In addition to built-in libraries, the following external libraries were used: `scikit-learn` (version 1.5.0) for ML, `pandas` (version 2.2.2) for data manipulation, `numpy` (version 2.0.0) for numerical computations and `matplotlib` (version 3.9.0) for data visualization. Furthermore, all the project files were executed in a controlled environment with Windows Subsystem for Linux (WSL)2.

4.8 Implementation Challenges

During the implementation of our methodology, several challenges emerged that required attention to ensure the successful implementation of our approach.

First, the availability of an appropriate dataset was a significant challenge. Despite being used in a considerable number of studies, some datasets were no longer available. Additionally, some datasets contained missing values, and their authors did not explain how these were addressed, making it difficult to replicate their methods of using the datasets accurately. Furthermore, some datasets only provided the locations of project files, requiring the extraction of software metrics directly from the corresponding code files. This task, as highlighted by various authors, is quite time-consuming and requires the right tools, which may have affected the overall project timeline. Considering these factors, we opted to use the MLCQ dataset, which includes pre-extracted software metrics and is based on industry-relevant data. This choice offered valuable insights into the application of code smell detection in real-world scenarios.

Second, identifying a suitable FCM clustering algorithm for feature selection that also provides a clear explanation of its methodology was a challenge. The MFCMS stood out as it offered a good explanation of the author's overall methodology and proved to be a viable option for feature selection. However, it lacked guidance on determining the optimal number of clusters and how to select the appropriate one. As a result, we developed our own approach for identifying the ideal number of clusters by visualizing the convergence patterns.

Moreover, the algorithm's complex mathematical operations required careful handling to avoid potential miscalculations that could negatively impact the results. Additionally, these operations can be computationally expensive, especially as the number of clusters increases. To address this, we focused on optimizing the calculations as much as possible. This was achieved using the `numpy` library, which is well-suited for handling complex mathematical operations efficiently.

Finally, another challenge we encountered was the optimization of the hyperparameters of the ML model. In the case of the Random Forest model, there are many different hyperparameters that can be tuned to enhance performance. However, as discussed in 4.5.1, optimizing all of them would be impractical with a reasonable timeframe. Thus, we focused on a subset of hyperparameters that could significantly impact the model's predictions. Furthermore, each hyperparameter has a wide range of possible values, but to ensure the optimization process remained efficient, we restricted this range. To achieve this, we relied on values commonly used in existing studies, which, as confirmed by our experiments, produced the desired results.

4.9 Conclusion

This chapter has meticulously detailed the methodologies and processes applied in the implementation of our approach to achieve the desired result. We began by outlining the dataset

used, emphasizing the preprocessing steps undertaken to ensure its suitability for each experiment. The chapter further elaborated on the feature selection mechanism, showcasing the strategies adopted to enhance the precision and efficiency of the detection process. Furthermore, the implementation of the ML model was discussed, covering the reasoning behind its selection, the optimization of its hyperparameters, and the model evaluation process. Through this overview, we have established a solid foundation for understanding the practical aspects of our study. These details will serve as a guide for future research endeavors and practical applications in the field of code smell detection, particularly for those considering the feature selection mechanism based on the MFCMS algorithm.

Chapter 5

Results and Discussion

5.1 Introduction

This chapter presents and analyzes the results of the experiments conducted to assess the performance of each approach. It examines the main phases of the methodology, highlighting both quantitative results and key observations, offering a comprehensive evaluation of the model's effectiveness. Additionally, it discusses the impact of the critical decisions made throughout the experiments, assessing the approach's strengths, limitations, and potential areas for improvement.

This chapter is divided into the following sections:

- **Ideal Number of Clusters (5.2)** – this section discusses the main findings about the optimal number of clusters for each experiment, presenting the results and identifying the specific cluster counts that are likely to achieve the best results in our methodology;
- **Results for Experiment 1 (5.3)** – this section presents and analyses the results obtained for the Experiment 1, providing insights into the strengths and weaknesses of the approach;
- **Results for Experiment 2 (5.4)** – this section presents and analyzes the results obtained from the Experiment 2, offering insights into the strengths and weaknesses of the approach;
- **Experiment 1 versus Experiment 2 (5.5)** – this section compares the two experiments, highlighting the advantages and weaknesses of each one, and identifies the limitations encountered in both.

5.2 Ideal Number of Clusters

Identifying the ideal number of clusters is a crucial step in our methodology, as it ensures meaningful results. By evaluating various criteria, we aim to determine the cluster count that best represents the underlying structure of the data, thereby enhancing the reliability of our findings. Figures 5.1 and 5.2 illustrate the analysis of the J_m values in relation to the number of clusters, c, for Experiments 1 and 2, respectively.

Upon examining Figure 5.1, we observed that the elbow point in the plot is around $c = 10$. This is where the curve starts to flatten out significantly. Before this point, there is a sharp decrease in the J_m value as the number of clusters increases. However, after the elbow point,

adding more clusters only results in a small reduction of the J_m value, indicating that the optimal number of clusters is likely around 10.

Additionally, considering the Cauchy's convergence criterion, the J_m values should stabilize as the algorithm iterates. Observing the plot, it is possible to analyzed that the process starts to flatten out after $c = 10$. This suggests that the algorithm has likely converged, and adding more clusters would not significantly improve clustering quality.

Therefore, based on the elbow point and the convergence criterion, $c = 10$ appears to be a reasonable choice for the number of clusters in Experiment 1.

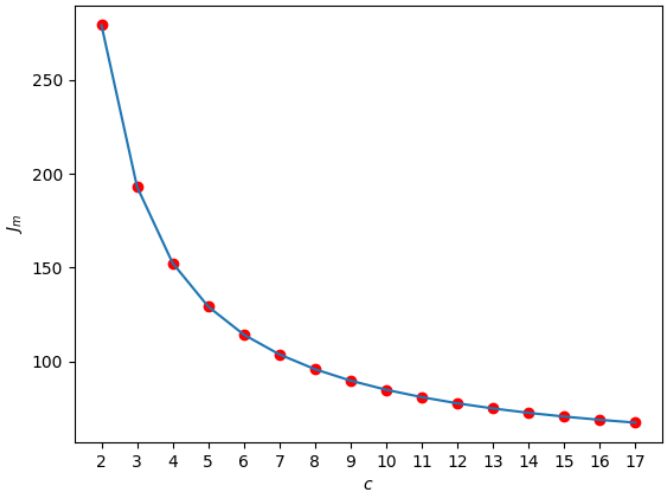

Figure 5.1: Analysis of J_m against cluster number for Experiment 1.

Observing Figure 5.2, we observed that the elbow point in the plot is around $c = 8$. This is where the curve starts to flatten out significantly. Before this point, there is a sharp decrease in the J_m value as the number of clusters increases. However, as already said, after the elbow point, adding more clusters only results in a small reduction of the J_m value, indicating that the optimal number of clusters is likely around 8.

Additionally, considering the Cauchy's convergence criterion and observing the plot, it is possible to analyzed that the J_m values starts to flatten out after $c = 8$, suggesting that the algorithm has likely converged, and adding more clusters would not significantly improve clustering quality.

Therefore, based on the elbow point and the convergence criterion, $c = 8$ seems to be a reasonable choice for the number of clusters in Experiment 2.

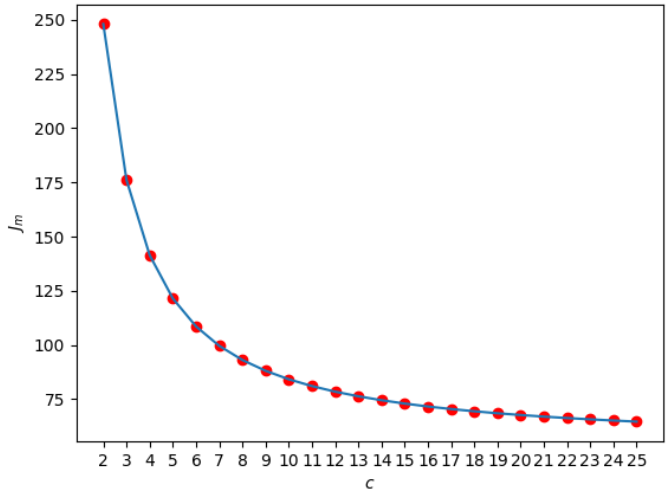

Figure 5.2: Analysis of J_m against cluster number for Experiment 2.

5.3 Results for Experiment 1

Following the analysis conducted in Section 5.2, the experiment was performed with the number of clusters set to 10 for executing the MFCMS algorithm. The threshold value τ was set to 2.0 as an initial criterion for selecting features based on their discrimination values in each cluster.

This value was selected because it was one of the highest discrimination indices observed among the features for each cluster. By choosing an initial threshold of 2.0, based on these maximum values, the aim was to ensure that only features with strong discriminative power – those close to the upper range of observed discrimination values – were selected. This approach ensures that the features included in the clustering process are the most effective at differentiating between clusters, thus enhancing clustering accuracy and interpretability. During the execution, however, this threshold will be further refined to identify the optimal value, with 2.0 serving as a starting point that will be adjusted as needed to achieve the best results.

The table 5.1 presents the dimensionality reduction of the dataset for Experiment 1. As we can observe, with a threshold of $\tau = 0.9999$, a total of 24 features were selected from the original 26, showing that the algorithm's capability for dimensionality reduction. However, in this first experiment, the reduction achieved was not substantial.

τ	Total Number of Features	Number of Features Selected
0.9999	26	24

Table 5.1: Number of features selected for the best threshold value τ in Experiment 1.

The Experiment 1, which utilized the original features without decorrelation, resulted in a relatively small reduction in dimensionality, likely due to the high correlation among fea-

tures, which indicates that many carried overlapping information. As a result, the selection algorithm struggled to distinguish unique contributions effectively and retained the majority of the features. Consequently, only a small subset of redundant or less important features was removed, limiting the impact of the dimensionality reduction.

Table 5.2, presents the results obtained for the test of our model on unseen data. The results of the model using feature selection revealed an improvement over the model without using feature selection. The most notable improvement was observed in precision, which increased from 0.7500 to 0.7895, suggesting that the feature selection process helped the model make more accurate positive predictions. This enhancement in precision, along with slight improvements in accuracy and F1-score, demonstrates the positive impact of feature selection on the model's overall performance, despite recall remaining unchanged.

Metrics	Results without Feature Selection	Results with Feature Selection
Accuracy	0.9294	0.9336
Precision	0.7500	0.7895
AUC	0.7660	0.7684
F1-Score	0.6383	0.6522
Recall	0.5555	0.5555

Table 5.2: Results for the test set in the Experiment 1.

Table 5.3 summarizes the performance of the classification model with feature selection applied, allowing for a focused evaluation of the model's effectiveness for each specific class (presence and absence of the long method smell, labeled as 0 and 1). This report provides valuable insights into the model's overall performance and its ability to accurately identify instances of each class.

Overall, while the model shows impressive performance metrics, particularly for class 0, it struggles significantly with class 1, as evidenced by its low recall and F1-Score. The AUC score of 0.7660 indicates that the model has a good ability to distinguish between the two classes. However, the slight improvement to 0.7684 with feature selection suggests that while there is some enhancement in the model's discriminatory power, it remains insufficient for accurately identifying positive instances (class 1).

This implies that the model needs improvement in recognizing instances of class 1, which may be linked to a class imbalance problem, as the dataset contains more negative instances (class 0) than positive ones (class 1), as noted in section 4.3. However, addressing this imbalance is not straightforward. Oversampling could introduce new instances that inaccurately represent cases where a code smell is present, potentially degrading the model performance. On the other hand, undersampling would reduce the dataset size so significantly that the model might overfit, meaning it would likely memorize the few remaining instances instead of learning generalizable patterns, leading to poor performance on unseen data.

Therefore, creating a more balanced dataset could help improve these results by allowing the model to better differentiate between positive and negative instances, leading to a more reli-

able and accurate performance. Additionally, monitoring the AUC score during this process could provide insights into the model's ability to classify instances correctly. This ensures that improvements in class recognition do not negatively affect the model's overall ability to differentiate classes.

	Precision	Recall	F1-Score	Support
0	0.95	0.98	0.96	428
1	0.79	0.56	0.65	54
accuracy			0.93	482
macro avg	0.87	0.77	0.81	482
weighted avg	0.93	0.93	0.93	482

Table 5.3: Classification report in the method with feature selection in Experiment 1

Based on the confusion matrix in Figure 5.3, we can observe some improvement in the model's ability to predict real instances. However, while the TPs instances show slight improvement, there remains a significant number of FPs. This suggests that the model still struggles to accurately predict most instances of class 1, as previously mentioned, suggesting it has difficulty identifying when the long method smell is present.

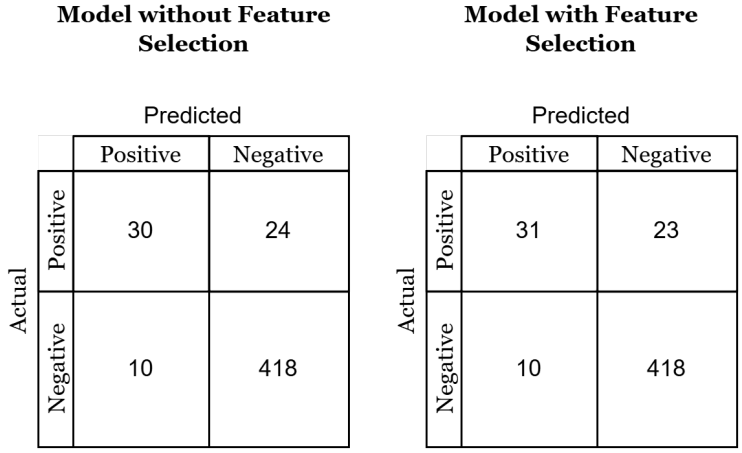

Figure 5.3: Confusion matrix for Experiment 1.

5.4 Results for Experiment 2

Experiment 2 was conducted using a cluster count of 8 for executing the MFCMS algorithm, as stated in 5.2. In line with Experiment 1, the threshold value τ was set to 2.0 as the initial criterion for selecting features based on their discrimination values within each cluster.

Similarly to Experiment 1, this threshold value was selected for its high discrimination index across features in each cluster, aiming to prioritize features with strong discriminative power.

Table 5.4 shows the dimensionality reduction achieved in Experiment 2. Using a threshold of $\tau = 0.6999$, the algorithm selected 20 features out of the original 26, indicating a substantial reduction. However, it is important to note that for this selection mechanism, we had to prioritize accuracy over precision, as precision did not improve during the selection phase in

training. This required us to follow the traditional selection method outlined by the authors in [58].

τ	Total Number of Features	Number of Features Selected
0.6999	26	20

Table 5.4: Number of features selected for the best threshold value τ in Experiment 2.

Table 5.5 presents the test set results from the second experiment, showing a slight improvement in the model using feature selection compared to the model trained on the full dataset. Although the improvement in accuracy and precision are minimal, it is worth noting that the model performed slightly better despite a significant reduction in features. This suggests that PCA may have contributed by removing correlations between features, allowing the model to focus on the most important information. As a result, PCA likely enhanced the feature selection process, leading to a more efficient model that achieved slightly better performance with fewer features, suggesting improved generalization and predictive capabilities.

Metrics	Results without Feature Selection	Results with Feature Selection
Accuracy	0.9294	0.9315
Precision	0.7500	0.7561
AUC	0.7660	0.7753
F1-Score	0.6383	0.6526
Recall	0.5555	0.5741

Table 5.5: Results for the test set in the Experiment 2.

In more detail, the table 5.6 summarizes the classification of the model with feature selection, allowing us to focus on each class and analyze the prediction capability of each one. Similarly, to the results of Experiment 1 presented in section 5.3, while the model shows impressive results for the absence of the long method smell (class 0), it continues to struggle with the detection of the presence of the long method smell (class 1), as evidenced by its low Recall and F1-Score.

The AUC score of 0.7753 underscores the model's moderate ability to distinguish between the two classes. This shows a slight improvement over the model without feature selection (AUC of 0.7660), but it also highlights that the model still has difficulty to accurately identifying class 1 instances, as evidenced by its low recall. Higher AUC values – closer to 1 – would indicate stronger class discrimination. Although progress is evident, the relatively minor improvement suggests that additional enhancements are needed to more effectively classify positive instances.

Following the same reasoning as in Experiment 1, the low recall and F1-Score indicate that the model struggles to accurately recognize instances where the long method smell is present. One way to address this issue is by applying data balancing techniques. In this case, undersampling would be a straightforward approach to balance the dataset by reducing the number of class 0 instances. However, as previously mentioned, undersampling could drastically reduce the

dataset size, increasing the risk of overfitting. This would hinder the model's ability to generalize and result in poor predictions on unseen data.

Given these limitations, a more effective solution would be to collect additional data to create a balanced dataset, as relying on synthetic data could introduce inaccuracies and lead to incorrect instances. This approach would help ensure that the model is trained on a more representative distribution of both classes, enabling it to better detect both the presence and absence of code smells. Ultimately, balancing the dataset without compromising its size is key to improving the model's ability to generalize and make accurate predictions across all instances.

	Precision	Recall	F1-Score	Support
0	0.95	0.98	0.96	428
1	0.76	0.57	0.65	54
accuracy			0.93	482
macro avg	0.85	0.78	0.81	482
weighted avg	0.93	0.93	0.93	482

Table 5.6: Classification report in the method with feature selection in Experiment 2

Based on figure 5.4, we can analyze the number of instances in the test set that were accurately classified compared to those that were not. In contrast with the Experiment 1, the model trained on the dataset with feature selection did not show an improvement in the classification of positive instances within the test set. However, it did improve the classification of the instances where code smells were absent. Additionally, similar to the confusion matrix from Experiment 1, a significant number of misclassified instances remain, particularly among those where the long method code smell is present.

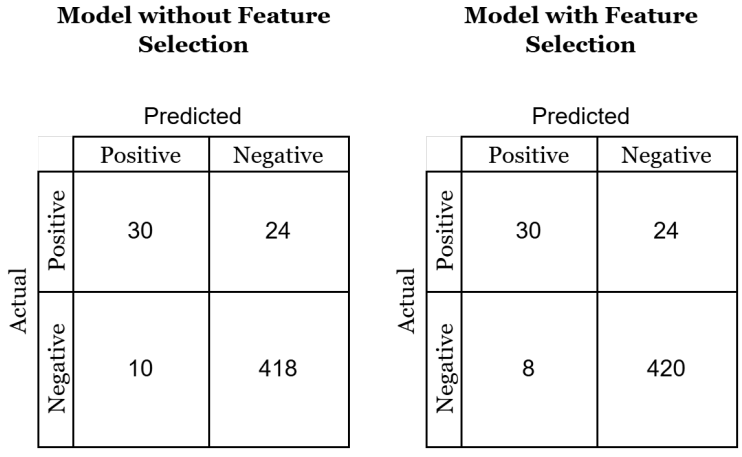

Figure 5.4: Confusion matrix for Experiment 2.

5.5 Experiment 1 versus Experiment 2

The Experiment 1, which used the original features without applying decorrelation, resulted in a small reduction in dimensionality. This was likely due to the high correlation between

features, making it difficult for the selection algorithm to effectively identify their unique contributions. In contrast, the Experiment 2 was designed to determine whether transforming features into uncorrelated components would allow for a more significant reduction in dimensionality by capturing essential information with fewer components than in the Experiment 1. The implementation of PCA in Experiment 2 revealed a substantial contribution to the dimensionality reduction, showing that the feature selection process in the Experiment 1 was limited by the presence of correlated features, while the Experiment 2 benefited from the use of uncorrelated components.

In terms of classification, Experiment 1 with feature selection yielded better results for accuracy and precision compared to Experiment 2. However, when considering the AUC score, F1-Score, and recall, Experiment 2 with feature selection produced slightly better outcomes for these metrics.

Both experiments revealed poor classification performance for instances with the long method smell present (class 1), as shown by the low recall and F1-Score in the classification reports. This pointed out the need for new techniques, such as data balancing, to address this issue. In particular, since there are considerably fewer positive instances (cases with long method smell present) compared to negative ones, balancing is crucial. However, data balancing poses challenges: undersampling might cause the model to overfit, while oversampling risks adding instances that do not accurately represent scenarios where the long method smell is present, potentially introducing noise and bias into the model. Creating a well-balanced dataset may improve the model's accuracy and precision, enhancing its ability to detect this code smell with higher AUC, F1-Score and recall.

5.6 Conclusion

This chapter analyzed and compared the results of two experiments to evaluate the effectiveness of different approaches for the MFCMS in feature selection. Both experiments showed that incorporating feature selection led to an improvement in model performance compared to models that did not use this technique. Additionally, the implementation of PCA in Experiment 2, as compared to Experiment 1, was found to be beneficial for reducing the dataset's dimensionality.

Despite these advancements, some areas for improvement were identified, particularly in addressing the model's low recall and F1-Score for detecting the long method smell. These challenges, which could be primarily driven by a class imbalance problem, highlight the need for further refinements to improve the model's performance in real-world applications.

In conclusion, while both experiments provided valuable insights into the model's performance and revealed the positive impact of feature selection and dimensionality reduction, further refinement is needed. Addressing these challenges will be key to optimizing the model for practical deployment.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Code smells play a crucial role in software development, as they indicate potential issues that could negatively impact the code of a given software system. They can affect the overall maintainability and comprehensibility of the code. Furthermore, code smells can result in the emergence of bugs, emphasizing the importance of addressing them quickly. In summary, identifying and solving code smells is crucial for an effective software development process. It ensures that the software remains robust, easy to understand, and efficient, resulting in a system that is both readable and easy to maintain. Therefore, detecting code smells is essential to proactively identify and address potential issues, ultimately leading to the elimination of these defects from the software.

This dissertation provides an overview of the code smells realm and detection techniques used for this domain. It focused on the implementation of a feature selection mechanism that is not commonly applied in this context; however, it could be a valuable option due to its ability to handle complex interactions between features. Specifically, the MFCMS algorithm offers a promising alternative because it combines the strengths of fuzzy clustering with supervised learning, allowing it to effectively identify subtle patterns in the feature space. This method's unique capability to manage ambiguity and overlapping feature characteristics makes it particularly well-suited for code smell detection, where relationships between code metrics can be intricate and nonlinear. As a result, MFCMS served as a powerful tool for feature selection, optimizing both the efficiency and effectiveness of code smell detection models.

In our experiments, we used the MFCMS algorithm for feature selection in two distinct scenarios: one involved applying PCA to the dataset beforehand to decorrelate the features, while the other relied on the original feature set. This allowed us to evaluate the impact of decorrelating features through PCA on the effectiveness of MFCMS for feature selection. We assessed the performance of each approach by analyzing key metrics, allowing us to draw informed conclusions on the most effective setup for detecting code smells.

The results of both experiments showed that the MFCMS algorithm was effective in identifying the most critical features, particularly in the second experiment, where decorrelating the features through PCA significantly enhanced the algorithm's ability to discern important patterns. This underscores the value of MFCMS as an effective feature selection method, especially when paired with preprocessing steps that reduce feature correlation. To improve the classification of instances with the long method smell and further validate the feature

selection and detection mechanism for code smells, it is recommended to incorporate a new dataset in future experiments. This additional data would allow for a more comprehensive evaluation across different scenarios, supporting further refinement of the approach and improving its effectiveness in accurately detecting both the presence and absence of the long method smell.

Considering the limited time and computational resources, the results are satisfactory and align with the study's objectives, providing a solid foundation for future research. However, there is still potential for improvement, and further studies will be crucial to refine these findings and contribute to advancements in the field of code smell detection.

In essence, we believe that the work presented in this dissertation successfully achieves the objectives defined at the beginning and makes a meaningful contribution to the field of code smell detection. It lays a solid foundation for future research, particularly in exploring the application of feature selection mechanisms for code smell detection.

6.2 Future Work

Despite the results obtained in this dissertation, we encountered a recurring challenge observed in previous studies, particularly in classifying positive instances where the long method smell is present. It is crucial to investigate whether the inherent class imbalance in the dataset contributes to this issue. Therefore, creating a new balanced dataset is essential, while avoiding oversampling techniques that introduce artificial data, as they could lead to misclassified instances or fail to accurately represent cases with the long method smell. This will allow for a more comprehensive evaluation of our model's performance and provide a more robust validation of our methodology across different scenarios.

Additionally, in terms of hyperparameter optimization, we suggest exploring Bayesian optimization for our ML model. As highlighted in the literature, grid search can be computationally expensive, especially with large parameter spaces, as it requires retraining the model for each combination. Bayesian optimization offers a more efficient alternative, reducing computational time and enhancing the optimization process. By utilizing past performance data, it can accelerate the search for the optimal hyperparameter combination and improve the overall model performance.

Future research could expand the scope of this work by focusing on other types of code smells that are not commonly addressed by current detection approaches, providing a valuable opportunity to improve the comprehensiveness of code smell detection models and enhance their applicability to a broader range of software systems.

Bibliography

[1] M. Lee, "Software quality factors and software quality metrics to enhance software quality assurance," *British Journal of Applied Science and Technology*, vol. 4, pp. 3069–3095, 2014. 1

[2] A. Al, "Empirical studies on software refactoring techniques in the industrial setting," *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 12, pp. 1705–1716, 2021. 1

[3] T. Guggulothu and S. A. Moiz, "Code smell detection using multi-label classification approach," *Software Quality Journal*, vol. 28, pp. 1063–1086, 2020. 1

[4] M. Fowler and K. Beck, *Refactoring: improving the design of existing code.* USA: Addison-Wesley Longman Publishing Co., Inc., 1999. 1, 5, 6, 7

[5] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow, "Code smells detection and visualization: A systematic literature review," *Archives of Computational Methods in Engineering*, vol. 29, no. 1, p. 47–94, Mar 2021. 1, 7, 8, 9

[6] G. Santos, A. Santana, G. Vale, and E. Figueiredo, "Yet another model! a study on model's similarities for defect and code smells," in *Fundamental Approaches to Software Engineering*, L. Lambers and S. Uchitel, Eds. Cham: Springer Nature Switzerland, 2023, pp. 282–305. 1

[7] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, "A large empirical assessment of the role of data balancing in machine-learning-based code smell detection," *Journal of Systems and Software*, vol. 169, p. 110693, 2020. 1

[8] U. Mansoor, M. Kessentini, S. Bechikh, and K. Deb, "Code-smells detection using good and bad software design examples," 2013. 5

[9] B. Bafandeh Mayvan, A. Rasoolzadegan, and A. Jafari, "Bad smell detection using quality metrics and refactoring opportunities," *Journal of Software: Evolution and Process*, vol. 32, p. e2255, 02 2020. 5

[10] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, 2003, pp. 381–384. 6

[11] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, no. 3, p. 395–431, 2006. 6

[12] Refactoring.Guru. Code smells. Last accessed on October 11, 2023. [Online]. Available: https://refactoring.guru/refactoring/smells 6, 7

[13] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, p. 1188–1221, Aug 2017. 6

[14] R. S. Menshawy, A. H. Yousef, and A. Salem, "Code smells and detection techniques: A survey," in *2021 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*, 2021, pp. 78–83. 7

[15] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014. 7, 8, 9

[16] 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. 7

[17] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 350–359. 8

[18] G. Langelier, H. Sahraoui, and P. Poulin, "Visualization-based analysis of quality for large-scale software systems," in *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, Nov. 2005, pp. 214–223. 8

[19] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. Ben Chikha, "Competitive coevolutionary code-smells detection," in *Search Based Software Engineering*, G. Ruhe and Y. Zhang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 50–65. 9

[20] 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)*, 2018, pp. 612–621. 9

[21] F. Arcelli 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, p. 1143–1191, Jun 2015. 9, 22

[22] F. Li, K. Zou, J. W. Keung, X. Yu, S. Feng, and Y. Xiao, "On the relative value of imbalanced learning for code smell detection," *Software: Practice and Experience*, vol. 53, no. 10, p. 1902–1927, Jun 2023. 9, 10

[23] A. AbuHassan, M. Alshayeb, and L. Ghouti, "Software smell detection techniques: A systematic literature review," *J. Software: Evol. Process*, vol. 33, no. 3, p. e2320, mar 2021. 9

[24] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, "A systematic literature review on the code smells datasets and validation mechanisms," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–48, jul 2023. 9

[25] O. Qasim and Z. Algamal, "Feature selection using different transfer functions for binary bat algorithm," *International Journal of Mathematical, Engineering and Management Sciences*, vol. 5, pp. 697–706, Aug 2020. 10

[26] A. Khalid, G. Badshah, N. Ayub, M. Shiraz, and M. Ghouse, "Software Defect Prediction Analysis Using Machine Learning Techniques," *Sustainability*, vol. 15, no. 6, p. 5517, mar 2023. 10

[27] M. Crabtree. What is Machine Learning? Definition, Types, Tools & More. Last accessed on November 10, 2023. [Online]. Available: https://www.datacamp.com/blog/what-is-machine-learning 10, 11

[28] E. Alpaydin, *Introduction to Machine Learning*. USA: The MIT Press, 2014. 10, 11

[29] G. Cloud. What is unsupervised learning? Last accessed on November 10, 2023. [Online]. Available: https://cloud.google.com/discover/what-is-unsupervised-learning#:~:text=Unsupervised%20learning%20in%20artificial%20intelligence,any%20explicit%20guidance%20or%20instruction. 10

[30] V. Kanade. What Is Reinforcement Learning? Working, Algorithms, and Uses. Last accessed on November 13, 2023. [Online]. Available: https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-reinforcement-learning/ 11

[31] S. Jaiswal. What is Normalization in Machine Learning? A Comprehensive Guide to Data Rescaling. Last accessed on December 12, 2023. [Online]. Available: https://www.datacamp.com/tutorial/normalization-in-machine-learning 11

[32] J. Ibitola. Data Normalization Demystified: A Guide to Cleaner Data. Last accessed on January 2, 2024. [Online]. Available: https://www.flagright.com/post/data-normalization-demystified-a-guide-to-cleaner-data 11

[33] H2O.ai. Feature Selection. Last accessed on January 3, 2024. [Online]. Available: https://h2o.ai/wiki/feature-selection/#:~:text=In%20the%20machine%20learning%20process,why%20feature%20selection%20is%20important. 11

[34] Heavy.ai. Feature Selection. Last accessed on January 3, 2024. [Online]. Available: https://www.heavy.ai/technical-glossary/feature-selection 11

[35] P. Charonyktakis. Do We Really Need Feature Selection in a Data Analysis Pipeline? Last accessed on January 3, 2024. [Online]. Available: https://towardsdatascience.com/do-we-really-need-feature-selection-in-a-data-analysis-pipeline-dc8401621c6c 12

[36] A. Klein, M. Seeger, and C. Archambeau. Hyperparameter Optimization. Last accessed on January 3, 2024. [Online]. Available: https://d2l.ai/chapter_hyperparameter-optimization/ 12

[37] A. Bonnet. Fine-tuning Models: Hyperparameter Optimization. Last accessed on January 3, 2024. [Online]. Available: https://encord.com/blog/fine-tuning-models-hyperparameter-optimization/#:~:text=Hyperparameter%20optimization%20is%20a%20key,learned%20during%20the%20training%20phase. 12, 13

[38] S. Paul. Hyperparameter Optimization in Machine Learning Models. Last accessed on January 3, 2024. [Online]. Available: https://www.datacamp.com/tutorial/parameter-optimization-machine-learning-models 13

[39] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "Code smell detection using ensemble machine learning algorithms," *Appl. Sci.*, vol. 12, no. 20, p. 10321, oct 2022. 13

[40] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Inf. Softw. Technol.*, vol. 108, pp. 115–138, 2019. 13

[41] Coursera. What Is Random Forest? Last accessed on January 3, 2024. [Online]. Available: https://www.coursera.org/articles/random-forest 13, 14

[42] IBM. What is random forest? Last accessed on January 3, 2024. [Online]. Available: https://www.ibm.com/topics/random-forest#:~:text=Random%20forest%20is%20a%20commonly,both%20classification%20and%20regression%20problems. 13, 14

[43] D. R. Yehoshua. Random Forests. Last accessed on January 3, 2024. [Online]. Available: https://medium.com/@roiyeho/random-forests-98892261dc49 14

[44] M. Fahmy Amin, "Confusion matrix in binary classification problems: A step-by-step tutorial," *Journal of Engineering Research*, vol. 6, no. 5, 2022. 15, 16

[45] N. Aryha Ahmed. What is A Confusion Matrix in Machine Learning? The Model Evaluation Tool Explained. Last accessed on January 3, 2024. [Online]. Available: https://www.datacamp.com/tutorial/what-is-a-confusion-matrix-in-machine-learning 15

[46] Google. Accuracy. Last accessed on January 3, 2024. [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/accuracy 16

[47] Deepgram. Precision and Recall. Last accessed on January 3, 2024. [Online]. Available: https://deepgram.com/ai-glossary/precision-and-recall 16

[48] C3.ai. What is the F1 Score? Last accessed on January 3, 2024. [Online]. Available: https://c3.ai/glossary/data-science/f1-score/ 16

[49] Google. ROC Curve and AUC. Last accessed on January 3, 2024. [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc 17

[50] J. C. Bezdek, J. Ehrlich, and W. Full, "Fcm: The fuzzy c-means clustering algorithm," *Computers & Geosciences*, vol. 10, no. 2, pp. 191–203, 1984. 17, 18

[51] G. for Geeks. ML | Fuzzy Clustering. Last accessed on January 3, 2024. [Online]. Available: https://www.geeksforgeeks.org/ml-fuzzy-clustering/ 17, 18, 19

[52] H. Yadav, J. Singh, and A. Gosain, "Experimental Analysis of Fuzzy Clustering Techniques for Outlier Detection," *Procedia Comput. Sci.*, vol. 218, pp. 959–968, jan 2023. 17

[53] K. Zhou, C. Fu, and S. Yang, "Fuzziness parameter selection in fuzzy c-means: The perspective of cluster validation," *Sci. China Inf. Sci.*, vol. 57, no. 11, pp. 1–8, nov 2014. 18

[54] V. Torra, "On the selection of m for fuzzy c-means," in *Proceedings of the 2015 Conference of the International Fuzzy Systems Association and the European Society for Fuzzy Logic and Technology*. Atlantis Press, 2015/06, pp. 1571–1577. 18

[55] S. Deng, "Clustering with fuzzy c-means and common challenges," *Journal of Physics: Conference Series*, vol. 1453, no. 1, p. 012137, jan 2020. 19

[56] P. Arumugam and P. Jose, "Efficient feature selection technique based on modified fuzzy c-means clustering with rough set theory," *International Journal of Advanced Research in Computer Science*, vol. 8, no. 7, p. 259–264, Aug 2017. 19

[57] W. Li, S. Zhai, W. Xu, W. Pedrycz, Y. Qian, W. Ding, and T. Zhan, "Feature selection approach based on improved fuzzy c-means with principle of refined justifiable granularity," *IEEE Transactions on Fuzzy Systems*, vol. 31, no. 7, pp. 2112–2126, 2023. 19

[58] F. Marcelloni, "Feature selection based on a modified fuzzy c-means algorithm with supervision," *Information Sciences*, vol. 151, pp. 201–226, 2003. 19, 24, 32, 35, 36, 50

[59] L. Madeyski and T. Lewowski, "Mlcq: Industry-relevant code smell data set," in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 342–347. 22, 23, 30

[60] A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, and G. Sladić, "Automatic detection of long method and god class code smells through neural source code embeddings," *Expert Systems with Applications*, vol. 204, p. 117607, 2022. 23, 24, 30, 31

[61] C. Team. Normalization. Last accessed on June 18, 2024. [Online]. Available: https://www.codecademy.com/article/normalization 30

[62] I. T. Jolliffe, *Principal Component Analysis*. New York, NY, USA: Springer, 1986. 31

[63] G. for Geeks. Principal Component Analysis (PCA) Intuition. Last accessed on June 19, 2024. [Online]. Available: https://www.geeksforgeeks.org/principal-component-analysis-pca 31

[64] U. D. of Mathematics. Cauchy's criterion for convergence. Last accessed on June 19, 2024. [Online]. Available: https://personal.math.ubc.ca/~cass/courses/m220-00/cauchy.pdf 37

[65] Y. Zhang, C. Ge, H. Liu, and K. Zheng, "Code smell detection based on supervised learning models: A survey," *Neurocomputing*, vol. 565, p. 127014, 2024. 39

[66] D. Berrar, "Cross-validation," in *Encyclopedia of Bioinformatics and Computational Biology*, S. Ranganathan, M. Gribskov, K. Nakai, and C. Schönbach, Eds. Oxford: Academic Press, 2019, pp. 542–545. 39

[67] Scikit-learn. KFold. Last accessed on June 19, 2024. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html 40

[68] N. Arya. Tuning Random Forest Hyperparameters. Last accessed on June 19, 2024. [Online]. Available: https://www.kdnuggets.com/2022/08/tuning-random-forest-hyperparameters.html 40

[69] P. Contreras, J. Orellana-Alvear, P. Muñoz, J. Bendix, and R. Célleri, "Influence of random forest hyperparameterization on short-term runoff forecasting in an andean mountain catchment," *Atmosphere*, vol. 12, no. 2, 2021. 40

[70] N. Zhu, C. Zhu, L. Zhou, Y. Zhu, and X. Zhang, "Optimization of the random forest hyperparameters for power industrial control systems intrusion detection using an improved grid search algorithm," *Applied Sciences*, vol. 12, no. 20, 2022. 40