ADDRESSING SUBJECTIVITY IN CODE SMELL DETECTION BY LEVERAGING HUMAN FEEDBACK IN A DEEP-LEARNING-BASED SOLUTION

by

Himesh Nandani

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
December 2023

To my parents and my sister, for without whom, I would be nothing I am today.

Table of Contents

List of Tables

List of Figures

Abstract

Code smells, being inherently subjective, can vary based on individual software developers' opinions and perspectives. Though many deep-learning-based models have been proposed for code smell detection, they often overlook the importance of considering each developer's subjective context, undermining the effectiveness of these methods. In this thesis, we address this limitation by (a) constructing a manually annotated dataset for three subjective code smells incorporating multiple annotations for each sample, (b) developing three state-of-the-art deep learning models with different architectures and comparing their performance on the dataset, and (c) conducting an extensive experiment using the best-performing deep-learning model to evaluate the impact of human feedback.

We start by building a dataset containing 10,267 annotations for 5,192 code snippets, targeting three code smells at different granularity levels: multifaceted abstraction, complex method, and long parameter list. Additionally, we create a supplementary dataset comprising 'definitely clean' and 'definitely smelly' samples, identified using the thresholds during dataset construction. To facilitate developers' involvement, we designed a plugin for IntelliJ IDEA and developed a container-based web server to offer services of our baseline deep-learning model within the IDE. This setup lets developers see code smells within the IDE and provide feedback (i.e., whether a snippet has the identified smell and whether a snippet has a smell not identified by our approach). In a controlled experiment, we collected feedback on code smells from 14 participants in two rounds: one with our baseline model and the second with a fine-tuned model. In the second round, we fine-tuned the model based on the participants' feedback and reevaluated the smell detection performance before and after adjustment. The results demonstrate that such calibration improves the smell detection model's performance by 15.49% in the F1 score on average across the experimental group participants. Notably, this improvement is achieved while maintaining a low Krippendorff score, indicating that the smell detection model's ability is enhanced while considering the subjectivity of the feedback.

Acknowledgements

I would like to express my deepest gratitude to Dalhousie University for providing me with the opportunity to pursue the Masters in Science course.

I am incredibly thankful to my Thesis supervisor Dr. Tushar Sharma for his unwavering support, guidance, valuable insights, and his valuable comments and critics throughout the course of this research. He taught me never to settle for anything less than perfect. He is a patient and thoughtful mentor. Their expertise and encouragement have been instrumental in shaping the direction of this thesis. I am proud to be one of his students.

Special thanks to my friend, research paper co-author, and SMART Lab co-member Mootez Saad, whose invaluable assistance has been crucial in making this research possible. His support and collaboration have been immensely beneficial.

I extend my heartfelt thanks to all the participants who generously contributed their time and efforts to provide feedback for the experiments conducted in this thesis. Their valuable input has been crucial in validating and refining the proposed approach.

I would also like to thank all the researchers, scholars, and developers in the software engineering community whose work and publications have been a source of knowledge and inspiration for this thesis.

Also, a big thank you to all my friends at the Dalhousie University. These years and people I have met are an integral part of my life and my thesis.

Last but not the least, I would like to express my heartfelt and special thanks to my parents Ila and Dinesh and my sister Sunisha who never stopped believing in me. I would never have managed any of this without their love and encouragement.

- Himesh

Chapter 1

Introduction

This chapter presents the context and background of the proposed research work. We introduce concepts such as code smells, technical debt, and the relationship between smells and technical debt. We present the problems with the current approaches in detecting code smells using deep learning and how they miss the subjectivity of code smells when trained on datasets created by multiple participants. We present an overview of the proposed research and our contributions.

1.1 Code smells and their subjectivity

Software development is a complex and error-prone process that demands meticulous attention to detail to produce high-quality code. As projects evolve, accommodating changes becomes increasingly challenging.

Code smells are a set of recurring, indicative patterns in source code that signal potential design or implementation problems. Code smells have been widely recognized as a form of technical debt, representing violations of software design principles [14, 25]. Fowler delineated a range of smells such as "Long Method", "Large Class", and "Duplicated Code" [15]. These descriptions provided a shared vocabulary for developers to communicate and identify problematic areas in their codebase.

```
class Organization {
    public static boolean isSeniorEmployee(Employee emp){
        if((emp.salary > 1000 && emp.salary < 5000)
            ||(emp.position.contains("senior") &&
            emp.position.doesNotContain("junior")))
        return true;
    return false;
    }
```

```
    // 30 more methods
}
```

In the above code snippet, two code smells, namely *complex conditional* and *insufficient modularization*, are evident. The "isSeniorEmployee" method in the Organization class exhibits a complex expression smell through its convoluted conditional logic. This complexity arises from the compound logical expression combining salary range checks and string containment evaluations in a single if statement. Such complexity makes the code hard to read and maintain. Additionally, the snippet also demonstrates *insufficient modularization*. Having too many methods, 30 in this case, within the same file is considered a bad coding practice and indicates that this class can further be modularized.

Technical debt refers to the cost of additional rework in software development caused by choosing an easy or limited solution now instead of using a better approach that would take longer [65]. Technical debt accumulates interest over time, making future changes more costly and complex. Code smells are intimately related to technical debt. They are early indicators of this debt, signaling underlying issues in the code that, if not addressed, can compound over time, leading to increased maintenance costs, decreased code quality, and reduced software agility.

Code smells and manifestations of common programming issues provide valuable insights into potential design flaws, maintainability problems, and other factors that may lead to future complications. Detecting code smells early in the development process enables developers to address them before the technical debt piles and code becomes more erroneous and expensive to rectify.

1.2 Research gap

Traditional methods of code smell identification have historically relied on metrics and heuristics-based analysis of source code [34, 54]. However, these approaches often produce a significant number of false positives because they are based on fixed rules and do not consider the context and subjectivity involved in smell detection [54].

Code smells inherently possess a degree of subjectivity. Different developers, based on their experiences, expertise, and contextual knowledge, may perceive and interpret these smells differently. This subjectivity can lead to varying opinions on what

constitutes a code smell and how severe it is. For example, one developer might view a particularly long method as a code smell indicating a need for refactoring, while another might consider it acceptable due to the complexity of the task it performs. This variation in perception complicates the process of accurately detecting and addressing code smells, as it relies heavily on individual judgment rather than strict, objective criteria.

To tackle the challenge posed by the detection of code smells, the research community has proposed various machine learning (ML) and deep learning (DL)-based approaches [9, 22, 28]. These methods attempt to leverage the power of data-driven techniques to better detect the code smells.

However, one key limitation of applying DL approaches for code smell detection lies in the reliance on code smell datasets to train the models. Existing code smell datasets suffer from constraints such as limited size, lack of filtering for definitively benign or smelly code snippets, and inadequate coverage of different types of smells. This limitation can hinder the generalization and effectiveness of the trained models.

Moreover, when using a DL-based approach to detect code smells based on datasets created by multiple participants, it can overlook the subjectivity inherent in identifying smells at the individual developer or team level [37]. Different developers may disagree about smells in the same code snippet, leading to discrepancies in the model's predictions. Consequently, a generic DL-based approach to detect code smells may not be as effective when applied universally.

1.3 Problem statement

Within the domain of software engineering, code smell detection assumes a critical role in ensuring the quality and sustainability of software systems. Despite numerous efforts to employ machine learning models for identifying code smells, these approaches often overlook the intrinsic subjectivity inherent in these smells. This oversight leads to generalized predictions, disconnecting developers from their nuanced perspective on these smells.

To address this issue, the proposed research seeks to create a personalized deep-learning system that takes into account the unique perspectives of each developer,

thus accounting for the subjective nature of code smells. To achieve this, we propose harnessing human feedback on the outcomes of the deep learning model and subsequently refining the model using the gathered feedback. We aim to establish a robust framework that confronts the challenges of subjectivity, contributing to the advancement of state-of-the-art in code smell detection.

1.4 Research questions

This thesis aims to address the following research questions:

RQ1 *Whether and to what extent does user feedback improve the accuracy of deep learning-based code smell detection?*

User feedback plays a crucial role in fine-tuning and refining DL models [60]. With this research question, we aim to validate that user feedback can enhance the accuracy of DL-based code smell detection and to understand the extent to which this improvement can be observed.

RQ2 *Whether and to what extent does user feedback influence the accuracy of deep learning-based detection for individual code smells?*

Code smells exhibit distinct attributes that differentiate them from one another due to the variations in their characteristics, patterns, and severity. With this research question, we aim to examine whether the improvement in accuracy, if any, through user feedback is consistent across all considered smells or if it varies for each smell. Furthermore, we seek to quantify the extent of this variation to understand the degree of improvement achieved.

RQ3 *Do feedback loops improve the accuracy of predicted smells at the cost of the subjectivity of the model?*

Deep learning models typically rely on a large dataset to learn and generalize patterns. However, during the fine-tuning process with extensive data, the subjectivity of individual users can be lost, resulting in a more generalized model that may not capture the unique perspectives and preferences of each user.

The research question aims to validate that by leveraging user feedback, we can achieve accuracy improvements in the models while ensuring that the subjectivity of each user for each code smell remains intact. By considering the specific feedback provided by users, the models can be customized to better align with their individual preferences, expertise, and contextual understanding.

1.5 Proposed approach

This thesis aims to fill the research gap in detecting code smells, keeping subjectivity in mind by suggesting a three-phase approach.

First, we propose a manually annotated dataset of code smells called *Dataset of Code Smells* (DACOS). We carefully filtered the code snippets to focus on potentially subjective cases to create an effective dataset. We removed snippets that were either definitely clean or smelly, allowing us to better utilize annotators' efforts where they are most needed. The dataset includes annotated code snippets for three specific code smells: *multifaceted abstraction* [49,57], *complex method* [50], and *long parameter list* [14]. Additionally, we provide the DACOSX dataset, which contains a large number of snippets that are either definitely clean or smelly. To facilitate the annotation process, we developed a web application called TAGMAN, which allows annotators to view one snippet at a time and indicate whether a smell is present in the code.

For the second step, we propose the creation of three state-of-the-art deep learning models with different architectures. These models are trained using the DACOS dataset we created in the first phase. After training, we compare the performances of these models to understand their strengths and weaknesses in code smell detection. We then choose the best-performing model for the subsequent phase.

In the third phase, we propose a novel approach that leverages machine learning techniques and a server-based architecture for code smell detection. This approach consists of two key components: a server and an IntelliJ plugin called TAGCODER.

The server component is developed as a Docker container using the Django framework in Python. It serves as the core of our code smell detection system, responsible for storing and processing the code metrics obtained from Designite, a code quality analysis tool. The server exposes four endpoints to handle different functionalities:

accepting class and method metrics from Designite [45], receiving code snippets for code smell prediction, collecting user feedback on the predicted smells, and triggering model retraining. By adopting a containerized architecture, we ensure minimal dependencies and facilitate easy deployment and scalability of the server.

The TagCoder IntelliJ plugin provides a seamless user experience for code smell detection within the IDE. When a user opens a project in IntelliJ IDEA, TagCoder automatically runs Designite in the background to analyze the project and sends the results to the server. Subsequently, when a user opens a file in the IDE, TagCoder sends the corresponding class and method information to the server for code smell prediction. The predictions received from the server are displayed in the gutter on the left pane of the IntelliJ IDEA editor, allowing users to identify potential code smells easily. Users can provide feedback on the predictions directly through the plugin, and this feedback is sent back to the server for analysis and model refinement. Additionally, TagCoder offers an option to explicitly trigger model retraining when a sufficient amount of feedback has been accumulated.

Following the implementation of the proposed tools, we intend to collect user feedback on code smell predictions. By gathering user feedback, we aim to refine and fine-tune the selected best-performing model for each specific user. The collected feedback will serve as valuable input to personalize the model's predictions according to the individual developer's coding practices and preferences.

1.6 Research contributions

The main contributions of our research are as follows:

1. Development of a web-server-based approach for code smell detection, providing a scalable and efficient solution for processing code metrics and predicting code smells.

2. Design and implementation of TagCoder, an IntelliJ plugin that seamlessly integrates code smell detection within the IDE, allowing developers to conveniently view and provide feedback on predicted code smells.

3. Complementary code smell dataset construction through the collection of user

feedback, addressing the limitations of existing datasets, and expanding the coverage of actively researched smells.

4. Evaluation of the effectiveness of our approach in terms of code smell detection accuracy and usability, demonstrating its potential to improve software maintainability and developer productivity.

1.7 Publications based on thesis

This research work has been the object of the following publications:

1. Nandani, H., Saad, M., & Sharma, T. (2023). DACOS—A Manually Annotated Dataset of Code Smells. In Proceedings of the IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), pp. 446-450. doi: 10.1109/MSR59073.2023.00067.

2. Nandani, H., Saad, M., & Sharma, T. (2023). Calibrating Deep Learning-based Code Smell Detection using Human Feedback. In Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2023).

3. Nandani, H., Saad, M., & Sharma, T. (2023). Calibrating Deep Learning-based Code Smell Detection using Human Feedback. Accepted at the Replication in Software Engineering (ROSE) track at the 23rd IEEE International Conference on Software Maintenance and Evolution (ICSME 2023). Won the **best artifact award**.

Chapter 2

Background

This chapter presents the context of the research and literature relevant to this thesis. It begins with a general introduction to code smells. Subsequently, we discuss related work that is pertinent to the main research of this thesis. Specifically, we highlight a gap in the literature regarding feedback learning for code smell detection.

2.1 Code smells

Code smells have garnered significant attention in software engineering research over the years, with numerous studies focusing on their identification, detection, and mitigation. A code smell is a surface indication that usually corresponds to a deeper problem in the system [14]. These smells serve as indicators of potential issues within software systems, highlighting areas that might benefit from refactoring or restructuring. Researchers have classified code smells into different categories based on the scope of their impact and the nature of the underlying problems they signify. We elaborate on the different categories of smells in the following sections.

2.1.1 Implementation smells

Implementation smells are typically confined to a limited scope, often affecting individual methods or code blocks. They often result from poor coding practices or suboptimal implementation choices. “Complex method”, “long method” and ‘magic number” are some examples of common implementation smells. Complex Method indicates that a method has become excessively convoluted, making it hard to understand, test, and maintain. It may involve intricate control flow, nested conditionals, or excessive code duplication [15]. Similarly, long methods are characterized by an excessive number of lines of code, making them difficult to comprehend and modify. They can hinder code readability and increase the risk of introducing bugs [15]

2.1.2 Design smells

Design smells are certain structures in the design that indicate violations of fundamental design principles and negatively impact design quality [57]. They often indicate violations of established design principles and can result in reduced maintainability, flexibility, and understandability. Examples of design smell include "Insufficient modularization" and "Multifaceted Abstraction". Insufficient modularization arises when an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both [57]. Multifaceted Abstraction smell arises when an abstraction has more than one responsibility assigned to it. It leads to confusion and complexity, making it harder to understand and modify the code [57].

2.1.3 Architecture smells

Architecture smells refer to the issues that affect the overall structure and organization of a software system. Architecture smells highlight flaws in the architectural granularity that can lead to reduced maintainability, extensibility, or scalability. An example of architectural smell is "God Component" [53]. God component occurs when a component is excessively large, either in terms of LOC or the number of classes. It typically centralizes the functionality of the application, leading to a design where a single component manages multiple aspects of the system that could be more effectively and safely handled by multiple smaller components. God component code smell violates the Single Responsibility Principle (SRP), a core concept in object-oriented programming. Another example of an architectural smell is "Scattered Functionality". This smell occurs when related functionality is spread across multiple modules or components, making it challenging to identify and modify the relevant code. It hampers the system's comprehensibility and can impede future enhancements [16].

2.2 Code smell detection

Code smell detection has been a topic of interest for a long time in the field of software engineering [7, 21, 40]. As such, there have been many attempts to develop

effective techniques and tools for identifying and addressing code smells. Early research in this area primarily focused on defining and cataloging specific code smells. Rule-based approaches were commonly employed, where predefined patterns or anti-patterns associated with code smells were checked against the source code. However, with the increasing complexity of software systems, more advanced techniques have been explored. These include static analysis-based approaches, machine learning-based approaches, and hybrid methods combining both [6,20]. The following sections provide a literature review of different approaches used in code smell detection.

2.2.1 Metrics-based approaches

Metrics-based approaches involve quantifying specific code characteristics or metrics and using predefined thresholds or rules to identify potential code smells. These approaches rely on analyzing code quality metrics such as cyclomatic complexity, lack of cohesion in methods, and code duplication. Tools such as PMD [41], Designite [45], and SonarQube [55] utilize metrics-based approaches by examining code metrics and providing suggestions for refactoring. In such methods, the source code is processed to create a code model, metrics capturing code characteristics are calculated, and then these metrics are compared against predefined thresholds to detect code smells. For example, the *God class* smell can be detected using metrics such as Weighted Methods per Class (WMC), Access To Foreign Data (ATFD), and Tight Class Cohesion (TCC) [32] [61]. These metrics are compared against predefined thresholds and combined using logical operators.

In addition to metrics-based approaches, another traditional approach for code smell detection is rules-based detection. Rules-based smell detection methods define specific rules or heuristics to identify code smells. These methods take the source code model as input and, in some cases, additional software metrics. Code smells are detected when the defined rules or heuristics are satisfied. By applying these rules, potential issues can be identified and flagged as code smells. For instance, the *cyclic hierarchy* smell can be detected by implementing a rule that examines whether a class is referencing its subclasses [57]. When this condition is met, it indicates the presence of a *cyclic hierarchy* smell. Rules or heuristics are often combined with metrics to improve the effectiveness of smell detection.

2.2.2 Machine-learning-based approaches

Machine Learning for code smell detection has gained a lot of momentum in recent years. Khomh *et al.* [22] use Bayesian Networks to predict *blob*, *functional decomposition*, and *spaghetti code* in two open-source projects. Maiga *et al.* [30] proposed SVMDetect, an approach to detect anti-patterns based on support vector machines. It predicts *functional decomposition, blob, swiss army knife* and *spaghetti code*. They have used Azureus, Xerces, and ArgoUML as the source code repositories. In a study conducted by Saeys *et al.* [43], hybrid feature selection techniques such as recursive feature selection with *random forest* and *support vector machine* were employed. The performance measures of single and ensemble feature selection were compared, and it was found that hybrid feature selection outperformed the other methods. Jiarpakdee *et al.* [19] examined 12 feature selection techniques on 14 open-source datasets and concluded that feature selection had an impact of up to 9% on prediction and that wrapper methods were expensive to implement.

2.2.3 Deep Learning-based approaches

Deep learning approaches, particularly those utilizing recurrent neural networks (RNNs) such as LSTMs [18], are effective in capturing long-term dependencies in sequential data. These methods have been applied to source code, either for improving semantic representations [5] or for solving downstream tasks.

Alternative approaches to mining source code have employed CNNs to learn features from various representations of code. Li *et al.* [27] have used single-dimension CNNs to learn semantic and structural features of programs by working at the AST level of granularity and combining the learned features with traditional hand-crafted features to predict software defects. Their method, however, incorporates hand-crafted features in the learning process and is not proven to yield transferable results. Similarly, a one-dimensional CNN-based architecture has been used by Allamanis *et al.* [3] to detect patterns in source code and identify "interesting" locations where attention should be focused. Similarly, Ren *et al.* [42] use a CNN-based neural network to identify self-admitted technical debt. Sharma *et al.* [48] used CNNs, RNNs, and Autoencoders (AEs) to detect code smells without explicitly specifying code features. They showed that DL models can detect smells in direct and transfer learning contexts.

2.3 Problems with machine learning and deep learning-based approaches

The main problem with ML-based methods for code smell detection is the high degree of disagreement on what constitutes a code smell among developers [26]. Hence, if a model performs well on a dataset annotated by a set of developers, it might perform poorly when evaluated by another set of developers. This deficiency makes these models unusable in real life within an industrial software development environment.

2.4 Available datasets for machine learning and deep learning-based approaches

Software engineering literature offers a small number of manually annotated datasets for code smells. Palomba *et al.* [39] offered a dataset "Landfill" containing annotations for five types of code smells—*divergent change, shotgun surgery, parallel inheritance, blob,* and *feature envy*. They offered annotations for 243 snippets. They also developed an online portal where contributors can annotate code for smells, However, as of the time of writing this thesis (November 2023), the portal is not accessible. Madeyski *et al.* [29] proposed MLCQ— a code smell manually annotated dataset. The dataset contains 14.7 thousand annotations for $4,770$ samples. The dataset considered four smells—*blob, data class, long method,* and *feature envy*. Both of the datasets mentioned above do not consider the subjectiveness of a code snippet; hence, most of the snippets might not add any new information for the machine-learning classifier when used in training. Also, we chose the code smells that are not covered by any existing code smell dataset and, hence complement the existing datasets. There are some code smells datasets such as QScored [51]. Though the QScored dataset is large, the samples are not manually annotated and hence lack the required capturing of context.

2.5 Human feedback in code smell detection

Feedback loops play a vital role in machine learning, enabling systems to continuously learn and adapt based on previous outputs [60]. In the context of software engineering, feedback loops have been widely explored for various purposes. Aguiar *et al.* [1]

present a use case for feedback learning in live programming, demonstrating how real-time feedback can enhance the programming experience. Balzer's work [8] focuses on live coding and feedback learning, investigating how feedback loops can facilitate code development and improve programming efficiency. Brun *et al.* [10] explore the application of feedback loops in self-adaptive systems, where the system dynamically adjusts its behavior in response to changing environments and emerging requirements.

Despite the existing literature on feedback loops in software engineering, to the best of our knowledge, no previous work has specifically investigated the utilization of human feedback for DL-based code smell detection. In our research, we propose an approach that incorporates human feedback to enhance the performance of code smell detection models.

Chapter 3

Building the Dataset - *Focus on Subjective Smells*

This chapter presents the dataset created for three subjective smells—*Multifaceted Abstraction, Long Parameter List, and Complex Method*. First, we illustrate the process of constructing the dataset in detail. The description includes TAGMAN, a tool we developed to help us build the dataset. Finally, we provide details on the characteristics of the created dataset.

3.1 Overview

Figure 3.1 provides an overview of the dataset construction process. In constructing our code smell dataset, we perform several steps. First, we have utilized the `searchgithubrepo` Python package [47], which leverages the GitHub GraphQL API [38], to filter and download repositories based on criteria such as stars, lines of code, recent activity, and code quality scores. Then, we split the repositories into methods and classes using the CODESPLITJAVA tool [46]. Next, we analyzed code quality metrics and code smells using the DESIGNITEJAVA tool [52]. To streamline the annotation process, we have developed the TAGMAN web-based tool for presenting code snippets to annotators. The manual annotation process has been divided into two phases, with the first phase used to identify potentially subjective snippets based on metric thresholds. In the second phase, we presented the filtered snippets to annotators for annotation.

3.2 Methods

In the following section, we provide detailed description of each part of the process we followed in constructing our code smell dataset.

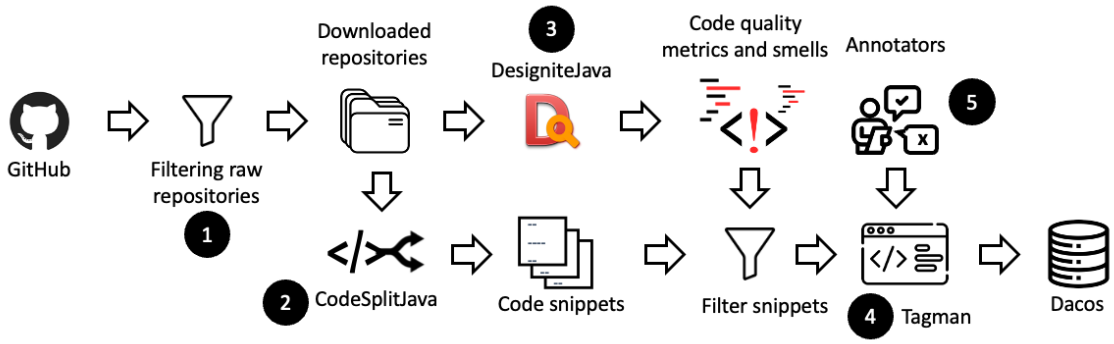

Figure 3.1: Dataset construction process

3.2.1 Downloading repositories

In step 1 from Figure 3.1, we perform the following tasks to identify and download repositories.

- We use `searchgithubrepo` [47] python package, which in turn uses the GitHub GraphQL API [38] to filter GitHub repositories.
- To identify high-quality Java repositories, we select repositories with more than or equal to 13 thousand stars and more than ten thousand lines of code.
- Also, we discard the repositories that have not been modified in the last year.
- In addition, we use QScored [51] to filter out repositories based on their code quality score. QScored assigns a weighted quality score based on the detected smells at various granularities. We select repositories with a quality score of less than ten (the higher the score, the poorer the quality).
- Finally, we obtained ten repositories after applying the filtering criteria. We download the selected repositories.

3.2.2 Dividing the repositories into classes and methods

We need to split a repository into individual methods and classes so that Tagman can show individual snippets individually to an annotator. We use CodeSplitJava [46] in step 2 to split each repository into individual methods and classes.

3.2.3 Analyzing repositories

In step 3, we employ a metrics-based filtering process in phase 2 of manual annotation. We use DesigniteJava [52] to compute code quality metrics. DesigniteJava computes a variety of code quality metrics and detects smells; it has been used in various studies [2, 11, 36, 53, 58]. We elaborate on the process to filter out non-subjective samples in the manual annotation step.

3.2.4 Tagman

Tagman is a web-based tool developed to streamline the code smell annotation process in software engineering research. Intending to enhance the efficiency and accuracy of code smell identification, Tagman offers a user-friendly interface for annotating code snippets with various smells. The tool employs a combination of front-end and back-end technologies to provide a seamless user experience.

The front end of Tagman is developed using Thymeleaf, a Java-based templating engine, along with HTML and CSS. This combination enables the creation of a visually appealing and interactive interface where users can easily navigate and annotate code snippets. The back end of the tool is built on SpringBoot, a popular Java framework, ensuring robustness and scalability. The data collected during the annotation process is stored in a MYSQL database, facilitating efficient retrieval and analysis. The entity-relationship diagram of Tagman is shown in figure 3.2.

To initiate the code smell annotation cycle, Tagman utilizes a CSV file containing the names and URLs of selected GitHub repositories. This file is uploaded to the tool, enabling the back end to utilize a set of Python scripts. These scripts perform several tasks, including downloading the GitHub repositories, splitting the code into separate class and method files, and running DesigniteJava, a code analysis tool. By leveraging these scripts, Tagman automates the initial stages of data collection and prepares the tool for the annotation phase.

Once the data import process is complete, Tagman becomes ready to accept annotations from users. The first step for users is to log in or sign up for the application, providing a personalized experience and ensuring the security and privacy of user data. After logging in, users are presented with detailed instructions that encompass the definitions and characteristics of different code smells. These instructions serve

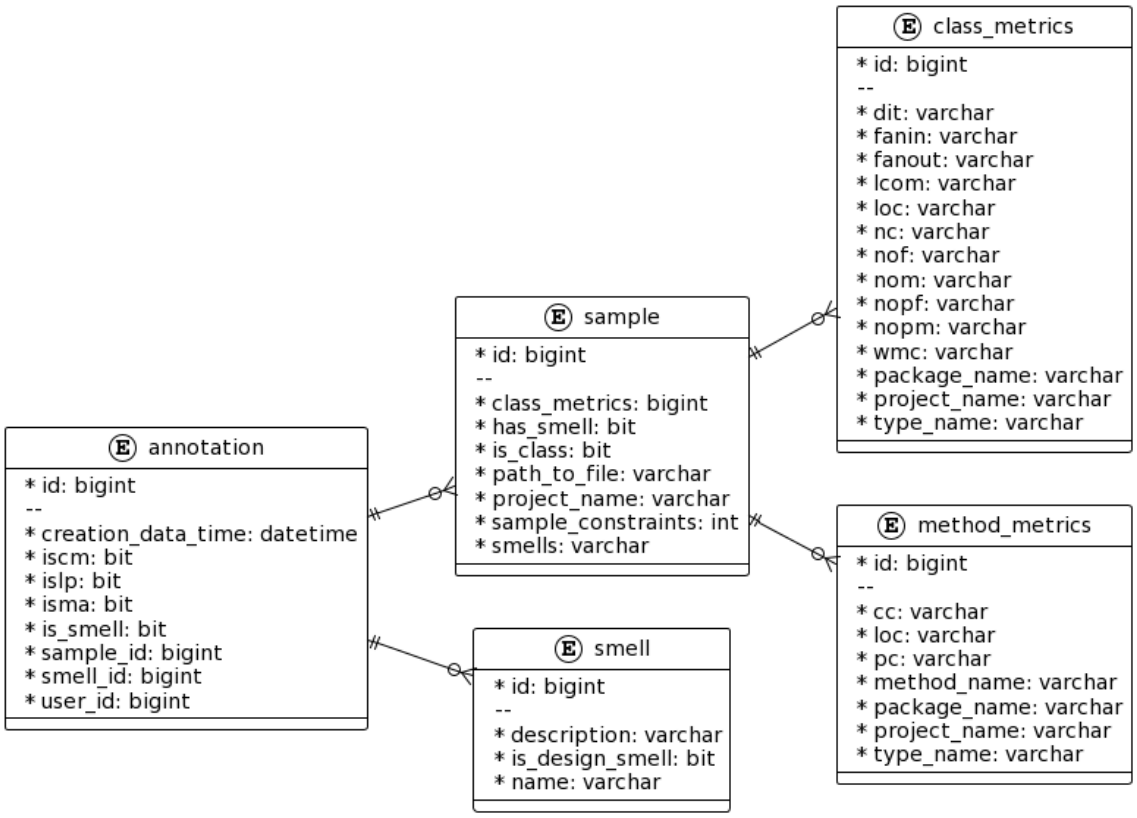

Figure 3.2: Schema of the DACOS database

as a reference guide to ensure a standardized and consistent annotation process.

With the instructions provided, users can proceed to annotate the presented code snippets. TAGMAN presents users with code snippets extracted from the previously downloaded GitHub repositories. Users can examine the code snippets and associate them with appropriate code smells based on their understanding and expertise. This process enables the collection of labeled data, which is valuable for training machine learning models, conducting statistical analyses, and gaining insights into the prevalence and impact of different code smells.

Figure 3.3 shows a screenshot of the application showing a code snippet and an option to annotate the snippet with a smell.

3.2.5 Manual annotation

We employ a snippet selection mechanism to identify potentially subjective snippets *w.r.t.* a code smell. We do so to improve the effectiveness of the resultant dataset

Figure 3.3: Annotation user interface of tagman

by only including manual annotations for potentially subjective code snippets. Also, such a strategy helps us better utilize the available annotators' time. A potentially subjective snippet is a code snippet that may get classified as benign or smelly based on context. The rest of the snippets that are not identified as potentially subjective snippets are either definitely clean or smelly snippets. For example, *cyclomatic complexity* (CC) [33] is commonly used to detect *complex method* smell. A code snippet is definitely benign if CC is very low (*e.g.,* CC=1); similarly, a snippet is definitely smelly if CC is very high for a method (*e.g.,* CC=30). We divide our annotation process into two phases. In the first phase, we show all snippets, *i.e.,* without any filtering, to annotators to identify metrics thresholds to determine whether a snippet is potentially subjective or not. The second phase uses the identified metrics thresholds and shows the filtered code snippets to annotators.

Phase-1

In the first phase, we show code snippets without any filtering to annotators. TAGMAN presents one snippet at a time to the annotators and collects their response on whether the shown snippet has a code smell or not. We show a code snippet to two randomly chosen annotators and record their responses. The annotators recruited,

on a volunteer basis, for this phase were graduate students of Computer Science enrolled in a software engineering course (during summer 2022) that covered code smells extensively. A total of 110 annotators participated in this phase.

We compute the minimum and maximum threshold for metrics that are used to decide the presence of a code smell based on the collected responses in Phase 1. We received a total of 17,869 responses in this phase. We compute the lowest metric value (t_l) where the smell is identified, for each smell individually, to obtain the threshold at the lower side. Similarly, we extract the highest metric value (t_h) where the smell is *not* identified. Then, we compute the standard deviation (`sd`) of the metric value for the samples where the smell is identified. Finally, we obtain the low threshold using `max(`m_l`, `t_l` - sd)` and high threshold using `min(`m_h`, `t_h` + sd)` for subjective snippet identification. Here, m_l and m_h represent the lowest and highest possible values of a metric. Table 3.1 summarizes the quality metrics corresponding to each code smell and their thresholds for identifying subjective snippets. For instance, for *cyclomatic complexity* metric, we obtain 4 and 7 from the above calculation after rounding the values to the nearest integer.

Table 3.1: Code quality metrics used for code smells and their low and high thresholds for subjective snippet identification

Code smells	**Code Quality Metric**	**Metric Threshold**
Complex method	Cyclomatic complexity	4–7
Long parameter list	Parameter count	3–6
Multifaceted abstraction	Lack of cohesion among methods (LCOM)	0.4–0.8

Phase-2

We configure our filtering mechanism based on the thresholds obtained from Phase 1 and invite annotators by advertising the link to Tagman installation on social media platforms such as Twitter and LinkedIn. The invitation was open to all software developers, software engineering students, and researchers who understand Java programming language and at least basic object-oriented concepts. We kept the invitation open for six weeks during Dec-Jan 2022-23. A total of 82 annotators participated in this phase. Tagman showed snippets that have metric values falling between the

low and high thresholds (inclusive). We configured TAGMAN to get two annotations for each sample to improve the reliability of the annotations.

3.3 Dataset characteristics

The dataset metadata, as shown in Table 3.2, provides an overview of the code smell annotations and samples in our dataset. The DACOS subset comprises samples that fell within the thresholds identified during phase 1, indicating that they were neither definitively classified as smelly nor clean. These samples were then annotated by at least two users to determine their code smell status. After the completion of phase 2, we received a total of 10,267 annotations for 5,192 samples from 86 annotators. Specifically, for the DACOS subset, we collected annotations for three specific code smells: Complex method, Long parameter list, and Multifaceted abstraction. The Complex method code smell received 4,349 annotations, corresponding to 2,197 samples. The Long parameter list code smell was annotated 3,221 times, representing 1,634 samples. Similarly, the Multifaceted abstraction code smell received 2,697 annotations, corresponding to 1,361 samples. In total, the DACOS subset comprises 10,267 annotations and 5,192 samples.

Table 3.2: Dataset metadata

Dataset	Code smell	#Annotations	#Samples
DACOS	Complex method	4,349	2,197
	Long parameter list	3,221	1,634
	Multifaceted abstraction	2,697	1,361
	Total	**10,267**	**5,192**
DACOSX	Complex method	–	94,489
	Long parameter list	–	93,442
	Multifaceted abstraction	–	19,674
	Total	–	**207,605**

On the other hand, the DACOSX subset consists of samples that were identified as definitely smelly or definitely clean based on the metric thresholds established during phase 1. These samples did not require user annotation, as they were filtered out using the predetermined method. The DACOSX subset includes a larger number

of samples but lacks specific annotations for the code smells. The Complex method code smell accounts for 94,489 samples, the Long parameter list code smell has 93,442 samples, and the Multifaceted abstraction code smell has 19,674 samples. In total, the DacosX subset consists of 207,605 samples.

Chapter 4

Building the Deep Learning Models

This chapter explains the methods used to construct the three deep learning models that were trained on the dataset we created. We present the architecture of the models, the process of building and training them, and an analysis of the results of each model considered. The analysis and comparison of these models help us select the best-performing one, which will serve as the base model for the next part of our approach - the controlled experiment.

4.1 Approach

Figure 4.1 presents an overview of the process used for training deep learning models. We explain each part of the process below.

4.1.1 Code smells dataset

For the initial model training and evaluation, we leverage the dataset we curated for code smell detection. The key advantage of using the DACOS dataset is that it collected annotations for non-trivial potentially subjective code snippets helping the machine learning-based classifiers to learn to segregate smelly and benign snippets with much more ease.

4.1.2 Splitting the samples

For the classification task, we create a dataset containing both smelly and benign samples for each code smell. We use a 70:30 split for training and testing. Table 4.1 illustrates statistics of the dataset.

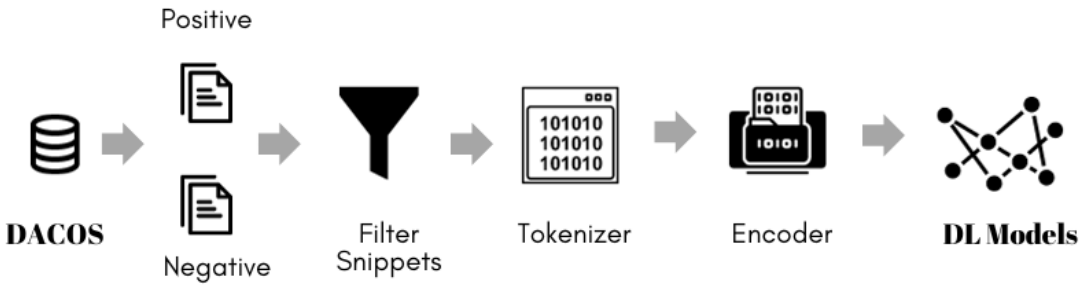

Figure 4.1: Building the Deep Learning Models

Table 4.1: Dataset statistics

Code smells	**#Training samples**	**#Test samples**	**Total samples**
Complex method	1,535	658	2,193
Long parameter list	1,137	487	1,624
Multifaceted abstraction	952	408	1,360

4.1.3 Filtering the snippets

Allamanis [4] shows that duplicate samples can lead to inflated and misleading results during testing. To avoid the issues potentially caused by duplicate samples, we perform data de-duplication using a hash function to compute a unique hash value for each code instance and by comparing the hash values to identify any duplicates.

4.1.4 Tokenizing learning data

To facilitate effective feature extraction for machine learning tasks, it is crucial to present the input data in a format suitable for analysis. In the case of source code, which is inherently text-based, it becomes necessary to convert it into a numerical representation that can be effectively processed by machine learning algorithms. In our approach, we employ the tokenizer associated with the respective code encoder to accomplish this task. By applying the tokenizer, we were able to transform the source code into a sequence of tokens, each representing a specific element or construct within the code.

The tokenization process yields a collection of tokens for each code file, forming the basis of our numerical representation. These tokens are subsequently converted into NumPy arrays, a commonly used data structure in numerical computing. Before further processing, we apply preprocessing and filtering techniques to ensure the quality and relevance of the data. Our preprocessing steps are kept minimal to preserve the essential characteristics of the code snippets while removing any redundant or extraneous information.

By converting the source code into a tokenized representation, organizing it into NumPy arrays, and conducting necessary preprocessing and filtering steps, we effectively transform the raw code data into a structured and suitable format for subsequent machine learning analysis.

4.1.5 Endoding the samples

To enable a deep learning model to leverage the contextual information present in the code samples, we incorporate an embedding step in our approach. By embedding the code samples, we facilitate the generation of meaningful context representations for further processing.In this study, we leverage the *transformers*[1] implementation of two Large Language Models pre-trained on code—*Code*BERT [12] and *Code*T5 [63].

CodeBERT is a bimodal pre-trained language model that has been specifically designed to support various Natural Language-Programming Language (NL-PL) applications, including code search and code documentation generation [13]. It employs a Multilayer Transformer architecture, which is widely used in large pre-trained models like BERT [59]. CodeBERT has been trained on a massive dataset comprising over 6 million projects from GitHub, encompassing diverse programming languages such as Java. To ensure its bimodal functionalities, CodeBERT has been trained using a hybrid objective function based on replaced token detection, incorporating both bimodal NL-PL data (consisting of source code paired with its corresponding documentation) and unimodal data (including NL and PL sequences).

On the other hand, CodeT5 is a unified pre-trained encoder-decoder Transformer model that capitalizes on the code semantics conveyed through developer-assigned

[1] https://github.com/huggingface/transformers

identifiers [64]. The architecture of CodeT5 is pre-trained with three distinct objective functions, namely Masked Span Prediction, Identifier Tagging, and Masked Identifier Prediction. These objectives serve as feedback signals to fine-tune the model parameters and enhance the code understanding capabilities.

By incorporating code embedding techniques using CodeBERT and CodeT5, we enable our models to grasp the nuances of the code samples and effectively utilize the contextual information within them. This empowers the models to make more informed predictions and enhances their overall performance in code smell detection tasks. This data is then ready to be passed to the deep learning models.

4.2 Architecture of deep learning models

In this section, we describe the architecture of the deep learning models we used in our experiment.

4.2.1 Autoencoder with multilayer perceptron

Figure 4.2 represents the architecture of the model. We divide the model into two parts.

Our approach employs an Autoencoder, which belongs to a class of feed-forward neural networks designed to reconstruct the input data. Autoencoders possess the capability to compress the input data into a lower-dimensional representation, known as the code, and subsequently reconstruct the output from this compressed representation. This process involves two key components: an encoder and a decoder.

The *Input* layer is the first layer of Autoencoder, configured to match the shape of the input vectors. Following the input layer, we incorporate a *Dense* layer, which is a type of neural network layer characterized by dense interconnections. Specifically, each neuron in the Dense layer receives input from all neurons in the preceding layer. The output of this *Dense* layer is a vector of dimensionality 'm'. To enhance the training stability and reduce the number of required epochs, we apply a normalization layer known as *Batch Normalization*. This technique standardizes the inputs to a layer for each mini-batch, contributing to the overall stability and efficiency of the training process.

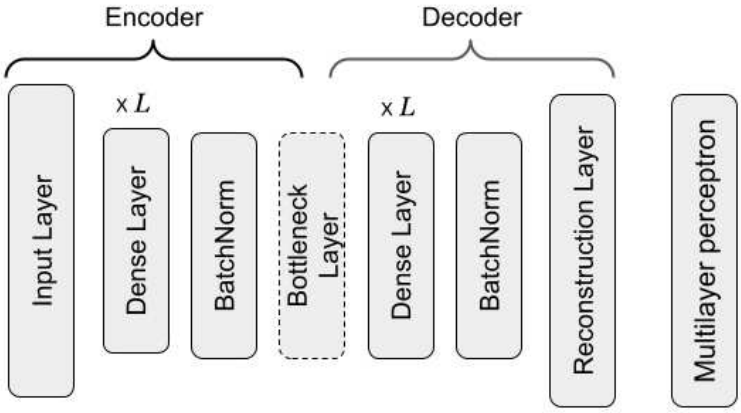

Figure 4.2: AutoEncoder with multilayer perceptron

Subsequently, we replicate these layers in reverse order to construct the decoder, which is responsible for reconstructing the input data from the compressed representation. During training, we divide the code samples into batches of size 16, and the model is trained using these batches over 200 epochs. It is important to note that once the training of the Autoencoder is completed, we discard the decoder component and retain only the encoder. The encoder is utilized to compress new instances of input data into vectors output by the bottleneck layer.

To further analyze the compressed code representations, we pass the input of the encoder to a multilayer perceptron model. Multilayer perceptron is a well-known feed-forward neural network widely used in machine learning tasks. By feeding the encoded input to this multilayer perceptron model, we aim to leverage its capability for pattern recognition and classification. The multilayer perceptron model, with its multiple layers of interconnected nodes, is adept at capturing and learning from non-linear relationships in data.

In order to further analyze the compressed code representations, we utilize a multilayer perceptron model. A multilayer perceptron is a feed-forward neural network that is widely used in various machine learning tasks. By feeding the encoded input to this model, our aim is to take advantage of its capability for pattern recognition

and classification. The multilayer perceptron model is effective in capturing and learning from non-linear relationships in the data, thanks to its multiple layers of interconnected nodes.

Our approach combines the strengths of the Autoencoder and the multilayer perceptron model to offer an effective framework for code smell detection. The Autoencoder learns to compress and reconstruct the input code, while the multilayer perceptron model harnesses the encoded representations to perform classification tasks.

4.2.2 Autoencoder with LSTM

The architecture of our proposed Autoencoder with LSTM model is depicted in Figure 4.3. This hybrid model is inspired by state-of-the-art approaches in Autoencoder-based architectures, customized to cater specifically to code smell detection.

The Autoencoder component of our model begins with an input layer that mirrors the shape of the input vectors. Subsequently, a Dense layer, characterized by comprehensive interconnections between neurons, follows. Each neuron in the Dense layer receives input from all neurons in the previous layer. This layer produces an 'm'-dimensional vector, which is then passed through a normalization layer. In our case, we employ Batch Normalization, a widely adopted technique in deep learning, to standardize the inputs for each mini-batch. This normalization step contributes to training stability and reduces the number of required epochs to achieve optimal performance. To construct the decoder, we replicate the layers in reverse order.

During training, we partition the code samples into batches of size 16, and the entire model is trained using these batches for 200 epochs. Once the training of the Autoencoder is completed, we discard the decoder component and retain only the encoder. This encoder is responsible for compressing new instances of input data into vectors that are output by the bottleneck layer.

Next, we introduce an LSTM model, which is a form of recurrent neural network (RNN). LSTMs were introduced by Hochreiter and Schmidhuber in 1997 and have gained significant popularity due to their ability to learn long-term dependencies. LSTMs excel in capturing and retaining information over extended sequences, mitigating the challenges associated with the vanishing or exploding gradient problem.

In contrast to standard RNNs, LSTMs are explicitly designed to overcome the

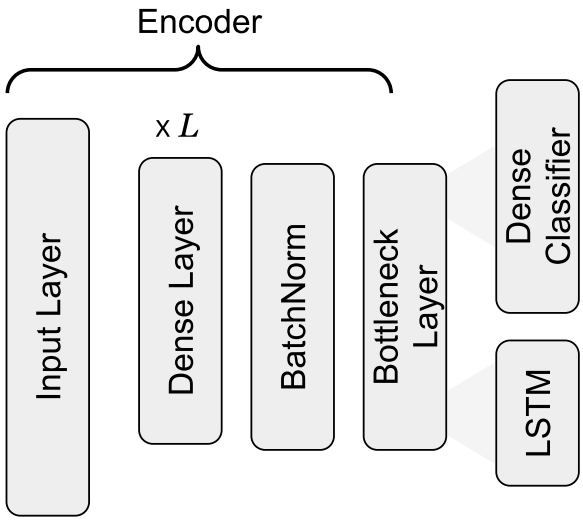

Figure 4.3: Using Encoder of the Autoencoder for training the dense classifier and LSTM model

issue of long-term dependency. They possess the capacity to retain relevant information for prolonged periods without struggling to learn it. To leverage the strengths of LSTMs for prediction, our model begins with an input layer followed by a bidirectional layer. The bidirectional layer allows the input to flow in both directions, enabling the model to incorporate information from both the past and future context. Subsequently, we incorporate a dropout layer with a dropout rate of 0.2, which aids in regularizing the model and preventing overfitting. Finally, a dense layer with sigmoid activation is employed to generate predictions. The model is then trained on the available data and becomes ready for prediction tasks.

Unlike standard RNNs, LSTMs are explicitly designed to solve the problem of long-term dependency. They can retain relevant information for extended periods without difficulty in learning it. To make the most of LSTMs for prediction, our model starts with an input layer followed by a bidirectional layer. The bidirectional layer allows the input to flow in both directions, incorporating information from both the past and future context. We then add a dropout layer with a dropout rate of 0.2 to regularize the model and prevent overfitting. Finally, we employ a dense layer with sigmoid activation to generate predictions. The model is trained on the data available and becomes ready for prediction tasks.

Our model combines the power of Autoencoders and the temporal understanding of LSTMs, making it a robust framework for code smell detection. The Autoencoder

compresses the input code effectively, while the LSTM model captures the temporal dependencies within the compressed representations.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2$$

By optimizing the network's parameters to minimize this loss, the autoencoder learns to encode and decode the input data effectively, capturing the most salient features in the process.

4.2.3 Autoencoders

The architecture of autoencoders can vary in complexity, with simple models typically employing dense layers to compress the input by reducing the number of units in intermediate hidden layers. Alternatively, compression can be achieved through the imposition of sparsity constraints on the activated hidden units, which can be accomplished using regularization techniques that introduce penalty terms in the loss function.

Table 4.2 provides an overview of the hyperparameters used for the DL models, including the Autoencoder. The hyperparameters include the number of encoder and decoder layers (Autoencoder), the loss functions used (MSE for the Autoencoder, CrossEntropy for MLP and LSTM), batch sizes (for all models), and the number of epochs.

Table 4.2: Hyperparameters for the DL models

Hyperparameter	**Values**
Encoder Layers (Autoencoder)	1,2
Decoder Layers (Autoencoder)	1,2
Loss functions	MSE (AE), CrossEntropy (MLP, LSTM)
Batch Size (all)	32,64
Epochs (all)	5,10,15,20

During training, the autoencoder minimizes the reconstruction error between the input and the output. We use the Mean Squared Error (MSE) loss function. The MSE loss calculates the average squared difference between the input and reconstructed output as shown in the following equation.

In the context of code smell detection, we construct simple sparse autoencoder models using dense layers, where the number of units in the intermediate layers is reduced and the loss function is penalized using L1-regularization [48]. The objective is to train the model to learn encodings specifically for negative instances, i.e., clean samples. Subsequently, the trained model is tested on both clean and smelly data.

Similar to the autoencoder, the VAE consists of an encoder and a decoder. The encoder transforms the input x into a latent representation z, while the decoder reconstructs the original input data ($\hat{x}$) based on this latent vector. The model's joint distribution is defined as follows:

$$p_\theta(x, z) = p_\theta(x|z) \cdot p_\theta(z)$$

The encoder, denoted as $q_\phi(z|x)$, where ϕ represents its parameters, produces estimates of the mean and variance variables of a Gaussian distribution. Using these estimated parameters, the VAE generates a latent vector z by sampling from the distribution. The decoder, denoted as $p_\theta(x|z)$, then reconstructs the original input by mapping the latent vector z to the output space. The decoder's parameters are represented as θ. The VAE aims to find the maximum likelihood by optimizing the following expression:

$$\sum_{i=1}^{n} \log p_\Theta(x_i)$$

Where Θ represents the parameter of the encoder and decoder, and $\log p_\Theta(x_i)$ can be expressed as:

$$\log p_\Theta(x_i) = D_{KL}\left(q_\Phi(z|x_i) \| p_\Theta(z)\right) + L(\Theta; \Phi; x_i)$$

Where D_{KL} is the Kullback-Leibler divergence between the posterior and prior distributions $L(\Theta; \Phi; x_i)$ and is called the evidence variational lower bound (ELBO). We train a VAE for each smell, similar to the two previous classifiers. Specifically, we train the VAE on the positive training samples. To perform the classification, we set a threshold α: if the loss measured is greater than the threshold, then it is classified as negative. The reason we do this is since the VAE has been trained on one class, it would have learned the salient features of that particular class, minimizing the reconstruction error after epochs of training. Hence, a high loss entails that the VAE was exposed to an outlier, *i.e.,* a sample from a different class. The value of α is

chosen after experimenting with multiple loss intervals with various steps, we report the value that yielded the highest predictive performance.

By employing this approach, we leverage the capacity of autoencoders to capture the underlying structure of the input code and learn representations that effectively discriminate between clean and smelly samples.

4.3 Results

Table 4.3 provides an overview of the classification results obtained using the CodeBERT and CodeT5 encoders for initial representation generation and the corresponding classifiers. For each encoder, we experimented with three combinations of Autoencoder (AE) and classifiers. The performance metrics of precision, recall, and F1-score are reported for each combination.

Using CodeBERT as an encoder

The results obtained for each code smell detection task when we used CodeBERT as an encoder are as follows. For the smell, the AE-MLP classifier achieved a precision of 0.56, recall of 0.80, and an F1-score of 0.66. The AE-LSTM classifier demonstrated slightly better results with a precision of 0.70, recall of 0.73, and an F1-score of 0.71. However, the VAE classifier outperformed both with a precision of 0.79, recall of 0.91, and an F1-score of 0.85. These results indicate that the VAE, leveraging the latent space representation, captured the distinguishing features of the smell more effectively.

For the *complex method* smell, the VAE classifier achieved the best performance among the three models with a precision of 0.80, recall of 0.99, and an F1-score of 0.89. This suggests that the VAE, by leveraging the probabilistic modeling and the threshold-based classification, effectively distinguished the *complex method* smell.

For the *long parameter list* smell, the AE-MLP classifier achieved a precision of 0.62, recall of 0.78, and an F1-score of 0.69. The AE-LSTM classifier exhibited a precision of 0.66, recall of 0.97, and an F1-score of 0.79. However, the VAE classifier achieved the best results with a precision of 0.81, recall of 0.90, and an F1-score of 0.85. The VAE's ability to capture the underlying probabilistic distribution of the *long parameter list* smell seemed to contribute to its superior performance.

Encoder	Model	Metric	Complex Method	Multifaceted Abstraction	Long Parameter List
CodeBERT	AE-MLP	Precision	0.60	0.56	0.62
		Recall	0.75	0.80	0.78
		F1	0.67	0.66	0.69
	AE-LSTM	Precision	0.68	0.70	0.66
		Recall	0.97	0.73	0.97
		F1	0.79	0.71	0.79
	VAE	Precision	0.80	0.79	0.81
		Recall	0.99	0.91	0.90
		F1	**0.89**	**0.85**	**0.85**
CodeT5	AE-MLP	Precision	0.69	0.60	0.60
		Recall	0.64	0.72	0.72
		F1	0.64	0.66	0.65
	AE-LSTM	Precision	0.84	0.45	0.80
		Recall	0.59	0.99	0.57
		F1	0.64	0.62	0.67
	VAE	Precision	0.76	0.77	0.83
		Recall	0.80	0.89	0.79
		F1	**0.78**	**0.83**	**0.81**

Table 4.3: Classification results for each type of smell using CodeBERT and CodeT5 with different classifiers.

Overall, the VAE consistently outperformed the AE-MLP and AE-LSTM classifiers across all three code smells. This can be attributed to the VAE's ability to model the latent space and capture the underlying probabilistic distribution. Leveraging the threshold-based classification, the VAE effectively distinguished positive and negative cases, resulting in higher precision and recall. In contrast, the AE-MLP and AE-LSTM classifiers demonstrated lower performance, potentially due to their limited capacity to capture complex patterns and dependencies in the data.

Using CodeT5 as an encoder

When using CodeT5 as an encoder, the results varied across different code smells. For the smell, the CodeT5 model achieved a precision of 0.60, recall of 0.72, and an F1-score of 0.66 when using the AE-MLP model. When using the AE-LSTM model, the precision was 0.45, the recall was 0.99, and an F1-score was 0.62. For the VAE model, the precision was 0.77, the recall was 0.89, and the F1-score was 0.83.

For the *complex method* smell, the CodeT5 model achieved a precision of 0.69, recall of 0.64, and an F1-score of 0.64 when using the AE-MLP model. When using the AE-LSTM model, the precision was 0.84, the recall was 0.59, and the F1-score was 0.64. For the VAE model, the precision was 0.76, the recall was 0.80, and the F1-score was 0.78.

For the *long parameter list* smell, the CodeT5 model achieved a precision of 0.60, recall of 0.72, and an F1-score of 0.65 when using the AE-MLP model. When using the AE-LSTM model, the precision was 0.80, the recall was 0.57, and the F1-score was 0.67. For the VAE model, the precision was 0.83, the recall was 0.79, and the F1-score was 0.81.

Overall, while CodeT5 exhibits varying performance across different code smells, the VAE consistently outperforms the AE-MLP and AE-LSTM classifiers. However, the performance of CodeT5 as an encoder is inferior compared to CodeBERT. CodeT5 shows lower recall scores for certain code smell types, indicating a higher rate of false negatives. This implies that CodeT5 has a more conservative approach to detecting code smells and may miss instances of code smells.

4.4 Discussion and selection of the best performing model

The VAE consistently outperformed the AE-MLP and AE-LSTM classifiers across all three code smells. This can be attributed to the VAE's ability to model the latent space and capture the underlying probabilistic distribution. Leveraging the threshold-based classification, the VAE effectively distinguished positive and negative cases, resulting in higher precision and recall. In contrast, the AE-MLP and AE-LSTM classifiers demonstrated lower performance, potentially due to their limited capacity to capture complex patterns and dependencies in the data.

For CodeT5 as well, VAE consistently outperforms AE-MLP and AE-LSTM across all the code smells. However, the performance of CodeT5 as an encoder is inferior compared to CodeBERT. CodeT5 shows lower recall scores for certain code smell types, indicating a higher rate of false negatives. This implies that CodeT5 has a more conservative approach to detecting code smells and may miss instances of code smells.

Based on the findings presented above, it is evident that the Variational Autoencoder (VAE) consistently outperformed the other deep learning models in terms of precision, recall, and F1-score. Therefore, we selected the VAE as our choice of the deep learning network for code smell detection. Additionally, we found that the CodeBERT transformer yielded better results compared to CodeT5 across the different code smells. Hence, we decided to utilize the CodeBERT transformer as the encoder for our VAE-based code smell detection framework.

This combination of VAE and CodeBERT resulted in the highest F1 score, and the system thus obtained demonstrated significant efficacy in detecting positive samples of code smells. Based on these observations, we selected this combination as the base model for our experiment.

Chapter 5

The Controlled Experiment

In this chapter, we outline the controlled experiment performed using the deep learning model we previously built. We first present the tools we created to conduct the experiment. Then, we present the procedure for the experiment. Finally, we evaluate the impact of fine-tuning the selected deep-learning model individually for each participant using the feedback gathered from them.

5.1 Overview

In the last phase, we carry out an experiment to determine the most efficient model. The main objective is to verify whether the inclusion of human feedback in the model's process enhances its accuracy in detecting code smells. This experimental setup allows for a clear comparison before and after, enabling us to quantify the model's performance improvement resulting from the integration of human insights.

Figure 5.1 illustrates an overview of our experiment. For our controlled experiment, we conducted our study with the participation of users who were divided into two groups: the control group and the experiment group. In the initial phase, participants from both groups are presented with code samples, where we utilize the *TagCoder* plugin to capture their feedback.

In the second phase, we introduce a feedback loop for the experiment group. We

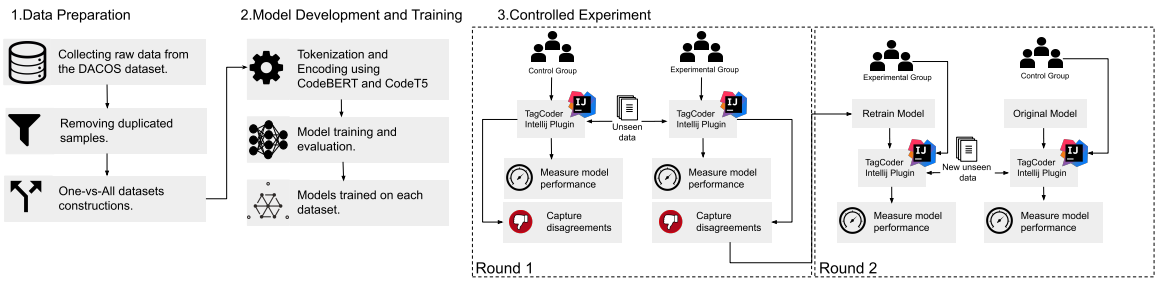

Figure 5.1: Overview of the approach

fine-tune the model using the user feedback captured from this group, tailoring the model's predictions according to their input. In contrast, the control group continues to interact with the same base model as in the first phase, without any fine-tuning based on user feedback. We then show predictions to both groups and collect and compare their feedback.

5.2 Methods

In this section, we describe the tools and the protocol that we developed to capture human feedback, and the results we obtained.

To conduct our experiment, we have developed two key software components—a web server serving the model to perform inference and a plugin for popular IntelliJ IDEA Integrated Development Environment (IDE) for displaying classification results and capturing developers' feedback. In this section, we describe each software component in detail.

5.2.1 A web-server

The primary objective of the web server is to support the inference from our DL model that is decoupled from the user feedback collection system. We minimize the dependencies required to run the server by encapsulating the server as a Docker container.

For the implementation of the server, we utilize the Django framework, which is based on the Python programming language. At the beginning of the experiment, the server accepts method and class metrics generated by running DesigniteJava [45] on the project. These metrics are stored in memory for subsequent processing. The server receives requests along with required data (such as metrics) from the client (in our case, our plugin for IntelliJ IDEA) and parses them based on predefined thresholds from the considered dataset. These thresholds are mentioned in the table 3.1.

The server uses four endpoints to connect with the plugin mentioned in the following sections. The communication between the server and the plugin is shown in figure 5.2. The server offers the following four endpoints:

- *Metrics endpoint*: This endpoint accepts a POST request with two CSV files—one

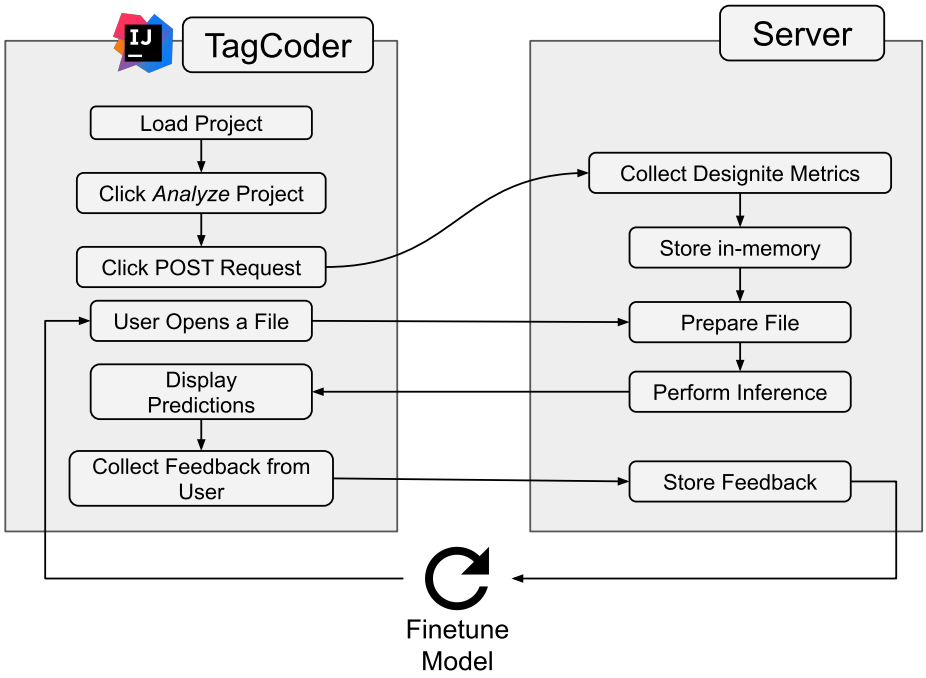

Figure 5.2: Interaction between TagCoder and local web server

containing class metrics and the other containing method metrics. These files are generated by DesigniteJava and are recorded using our IntelliJ IDEA plugin.

- *Prediction endpoint*: A POST endpoint that accepts a source code file, along with a boolean value indicating whether it pertains to a class or a method. The file is passed to the DL model for inference.

- *Feedback endpoint*: Another POST endpoint that allows users to provide feedback on the model's predictions for a specific file. When a user is presented with a prediction, their feedback regarding the correctness (according to them) of the prediction is collected through this endpoint. The endpoint records the file and the user's feedback. Once a preset number of feedback instances have been collected, the model is fine-tuned.

- *Fine-tuning endpoint*: A GET request that enables users to explicitly trigger model fine-tuning whenever desired.

To simplify the deployment and setup process, we have also provided a Docker script. This script automatically downloads all the necessary dependencies, including Python and the required DL libraries, ensuring a streamlined deployment of the server.

To expedite request processing and reduce server load, we leverage the metrics collected by DesigniteJava along with the rationale related to subjectivity earlier established when constructing DACOS. We identify whether a method or a class is definitely smelly based on metrics values higher than a threshold. Similarly, when the web server receives a code snippet for prediction, we first look up the corresponding metrics. Based on preset thresholds as used by Nandani *et al.* [35], we can quickly identify whether the sample is definitively smelly or definitely benign. For instance, if a method has eight parameters and the threshold for the "Parameter Count" metric is set between two and four, lower and higher thresholds respectively, we can conclude that the method exhibits the *long parameter list* code smell. If the sample falls within the predefined threshold, we use the model to infer whether a smell is present or not and return the inference back to the plugin. If not, we pass the sample to the DL model for further analysis.

To simplify the deployment and setup process, we use a Docker script. This script automatically downloads all the necessary dependencies, including Python and the required DL libraries, ensuring a streamlined deployment of the server. We store the user feedback for the presented code samples on a configurable volume within the Docker container. This ensures that user feedback is not lost even if the container is shut down or restarted.

Additionally, we have implemented a mechanism to fine-tune the model based on the collected feedback. Once the number of feedback instances reaches a pre-defined, but configurable, threshold (currently set to 50), the web server invokes fine-tuning the model incorporating the new information. Although we have arbitrarily assigned this number, it is configurable, allowing users to adapt the model fine-tuning frequency to their specific needs.

We set up and configured the web server locally to avoid sending code snippets to a third-party server configured outside of an organization boundary. However, due to the flexibility of the containerized web server, one can choose to install it on their local machine, a server within their organization, or on public cloud infrastructure.

5.2.2 TagCoder—an IntelliJ IDEA plugin

To enhance the usability and convenience of our code smell detection system, we developed *TagCoder*—a plugin for IntelliJ IDEA, a widely-used Integrated Development Environment (IDE) for software development, particularly in Java. The objective of *TagCoder* is to provide users with a seamless experience in obtaining predicted code smells for their code and offering a straightforward mechanism for providing feedback on these predictions. When a user opens a project in the IDE, *TagCoder* automatically runs DesigniteJava in the background to analyze the project. The obtained results are then sent to our local web server, which serves as the core of our code smell detection system.

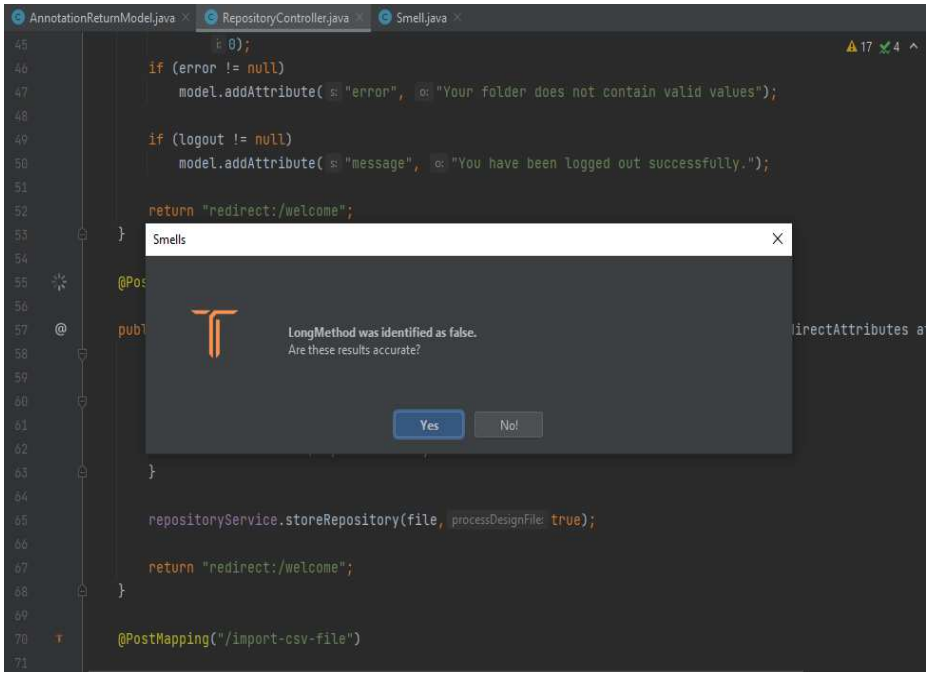

Figure 5.3: User feedback collection using TagCoder

After the initial analysis, whenever the user opens a file within the IDE, *TagCoder* automatically sends the corresponding class and method information to the web server for code smell prediction. The server processes the received code snippets and returns the predictions back to the plugin.

The plugin then displays the predictions in the gutter on the left side pane of the IntelliJ IDEA editor, allowing users to conveniently view the code smells associated with specific classes and methods. This integration within the IDE's interface enables

users to easily identify potential code smells without disrupting their workflow.

TagCoder supports recording users' feedback. Users can provide feedback on the identified smells directly from the plugin. This feedback is captured by the plugin and sent to the web server for documentation, analysis, and model refinement. This iterative feedback loop helps improve the accuracy and reliability of the code smell predictions over time.

Additionally, *TagCoder* offers an option to explicitly trigger model fine-tuning when users feel that a sufficient amount of feedback has been accumulated. This ensures that the model remains up-to-date and capable of capturing evolving code smells as the project progresses.

Figure 5.3 provides a snapshot of the *TagCoder* plugin, showcasing the user interface where code smell predictions are displayed in the gutter, along with the feedback capture dialog. By integrating *TagCoder* into IntelliJ IDEA, we aim to streamline the code smell detection process, providing developers with real-time insights into potential code quality issues and facilitating their active participation in improving the model's accuracy.

5.3 Experiment setup and design

The third step of Figure 5.1 illustrates the approach to conducting our controlled experiment. We elaborate on the experimental design and setup in the rest of the section.

5.3.1 Participants

We recruited 14 participants who have a background in Computer Science and are enrolled in a graduate, post-graduate, or doctoral degree program. They were solicited using a relevant internal mailing list. Participation was voluntary, but a small monetary reward was offered at the end of the experiments to compensate for their time. All participants were informed about the purpose of the study and were asked to provide consent to record and publish the anonymous data. They were also informed that all personal information (such as name and email) gathered will be confidential and only the researchers involved in the study will have access to the personal data

collected. The experiment took place at Dalhousie University spanning two days in May 2023.

5.3.2 Procedure

Before conducting the actual experiment, we performed a pilot study involving a participant to get their feedback regarding the *TagCoder* plugin. We do so to minimize any errors that could occur during the experimental procedure and get an estimate of the needed time to perform the annotation of at least 50 code snippets in each round. We then make necessary changes in the plugin as well as in the process based on the feedback we received. We conducted a second round of validation with another participant, however, we ignored the feedback this time as it was only related to one concern which was the latency of the model's inference ($\sim 3s$ per sample).

During the experimentation days, each participant was randomly assigned to a group (*experimental* or *control*) in a way that both groups have the same number of participants. We provide the same computer for each participant with IntelliJ IDE and *TagCoder* plugin installed. The source code project that was imported into the IDE can be found in the replication package.

We then present the same source code to all the participants and ask them to perform the following tasks:

- Open the IDE and navigate to the "Tools" menu. In the menu, select the option to analyze code using *TagCoder*.
- Open the source code files one by one. The methods and classes would have the *TagCoder* icon in the gutter of the editor on the left.
- Assess the smells detected initially by the model by clicking on the *TagCoder* icon. The plugin shows the kind of smell along with its description. They then provide their feedback (*i.e.,* agree or disagree with the detected smell) to the model from the same dialog box.

For the experimental group, after annotating at least 50 samples in Round 1, the model is fine-tuned. Upon fine-tuning the model, the web server notifies the plugin, and the plugin shows a popup notification to inform the user about completing the

fine-tuning process. However, a participant can also manually trigger the model's fine-tuning using the option present in the menu bar. The participants in the control group were presented with a modified version of the plugin. The modified plugin looks and behaves the same as the one presented to those in the experimental group except for a minor tweak-the model is not fine-tuned for the control group. Both the group members were unaware of their group and the difference in the plugin. For each session with a participant, we use a new copy of the original trained model and ensure that the fine-tuned models are saved for individuals and are not reused.

In Round 2, we asked both groups to follow the same set of tasks mentioned earlier. The end of this round marks the end of the session.

We conducted the experiments in succession; one participant after the other. For each session with a participant, we use a new copy of the original trained model and ensure that the fine-tuned models are saved for individuals and are not reused.

5.3.3 Data collection procedure

All participants were shown the same code repository[1]. On average, completing both rounds took every participant approximately 48 minutes. The average number of annotations collected per user was 101 over both rounds. We collected $1,421$ annotations from this experiment, where the number of annotations performed by the control group ranged from 81 to 135. In contrast, the number for the experiment group ranged from 73 to 135. For each participant, we store the model's prediction and their response and calculate a hash for the code snippet to identify it uniquely. The reason we track the actual annotated code snippets is to measure the inter-rater reliability later. The classification performance metrics were calculated based on our received data by treating the participants' responses as the ground truth.

5.4 Results and discussion

In this section, we illustrate the results of our experiments and discuss the implications.

[1] `https://github.com/SMART-Dal/Tagman`

5.4.1 RQ-1: Whether and to what extent does user feedback improve the accuracy of deep learning-based code smell detection?

User feedback plays a crucial role in fine-tuning and refining DL models [60]. With this research question we aim to validate that user feedback can enhance the accuracy of DL-based code smell detection and to understand the extent to which this improvement can be observed.

Approach

During the experiment, the participants were given software samples and asked to identify any code smells present. The experiment was conducted in two rounds. In the first round, the initial model was used, and in the second round, the model was retrained with additional data.

To evaluate the participant's performance, the responses for each sample in both rounds were curated. The user response was considered the ground truth, indicating whether a software sample was a code smell or clean code.

The evaluation metrics used were Precision, Recall, and F1-score, which are common metrics in binary classification tasks:

- True Positive (TP): Instances where the model predicted a code smell, and the user agreed with the prediction.
- False Negative (FN): Instances where the model predicted a sample as clean, but the user disagreed and believed it was a code smell.
- False Positive (FP): Instances where the model predicted a sample as a code smell, but the user disagreed and believed it was clean code.
- True Negative (TN): Instances where the model predicted a sample as clean, and the user agreed with the prediction.

We then calculated the precision and recall for each participant from the experiment and control group for the first and second rounds. The calculations of precision and recall are as follows:

Precision quantifies the ratio of correctly identified positive samples (true positives) to the total number of positive predictions made by the classifier. It serves as an indicator of the accuracy of positive predictions.

$$P = \frac{TP}{TP + FP}$$

Recall measures the proportion of true positive predictions with respect to all actual positive samples in the dataset. It assesses the ability of the classifier to identify positive samples correctly.

$$R = \frac{TP}{TP + FN}$$

Using these values, we calculated the F1-score of the participants before and after retraining the model.

The F1 measure is the harmonic mean of precision and recall:

$$F1 = 2 \cdot \frac{P \cdot R}{R + R}$$

It combines both precision and recall into a single metric that provides a balanced assessment of the classifier's performance

Results and Discussion

The difference in the F1 scores of participants in the control group and experiment group are shown in figures 5.4 and 5.5.

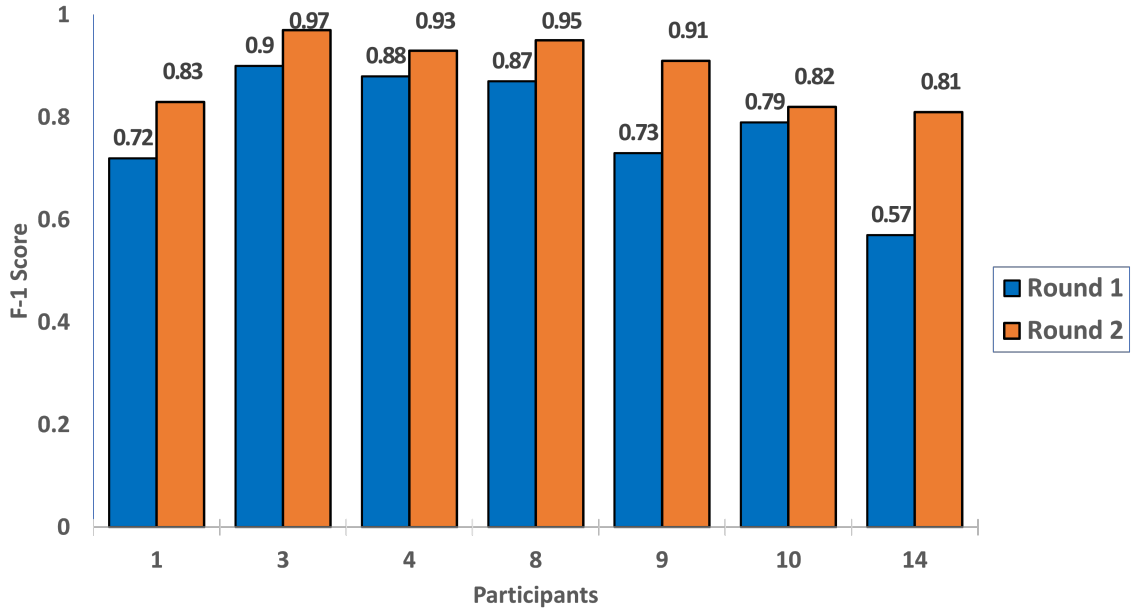

Figure 5.4: Before and after F1 scores for participants from experiment group. $M_{\text{Before}} = 0.78\,(\pm 0.1)$, $M_{\text{After}} = 0.88\,(\pm 0.06)$

In the control group, the participants' total code sizes ranged from 81 to 135. The initial F1 scores before the feedback loop ranged from 0.80 to 0.89. The feedback loop for the control group was a placebo and did not fine-tune the algorithm. After the feedback loop, the F1 scores showed slight variations, with values ranging from 0.82 to 0.90. The improvement in the F1 score ranged from 0.01 to 0.02.

For the experiment group, the participants' total code sizes ranged from 73 to 135. Before the user feedback loop, the F1 scores varied between 0.57 and 0.90. After the feedback loop, the F1 scores improved and ranged from 0.81 to 0.97. This indicates that the customization based on user feedback had a positive impact on code smell detection performance in the experiment group.

The participant with code size 79 in the experiment group had the lowest initial F1 score of 0.57, but after the user feedback loop, the F1 score improved significantly to 0.81. This indicates that even participants who initially had lower code smell detection performance were able to benefit from the customization based on user feedback.

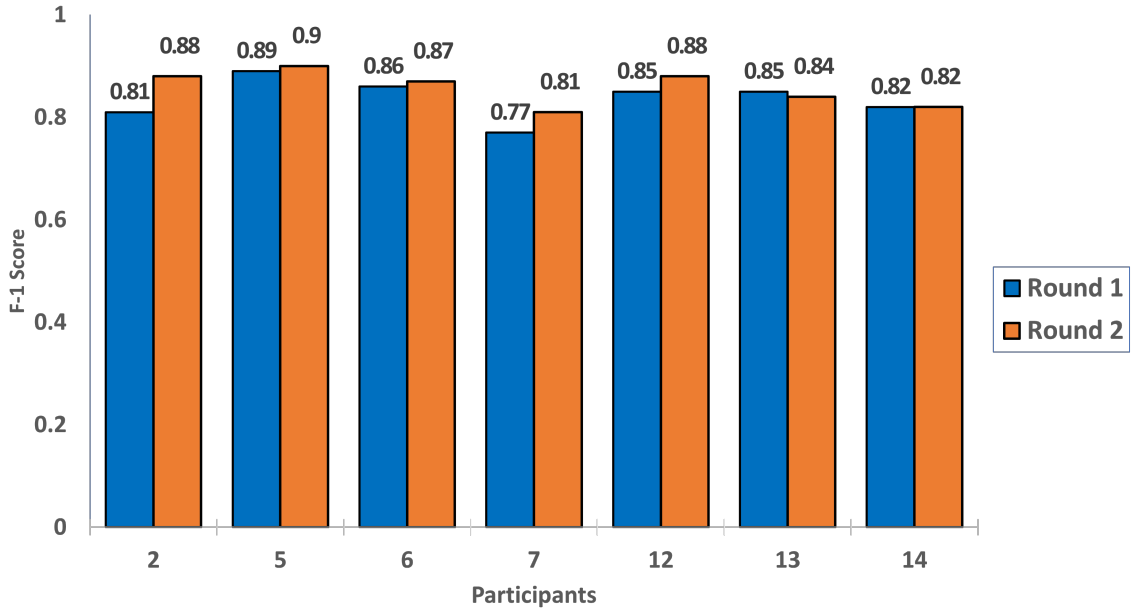

Figure 5.5: Before and after F1 scores for participants from control group.$M_{\text{Before}} = 0.78\,(\pm 0.1)$, $M_{\text{After}} = 0.88\,(\pm 0.06)$

We conduct statistical tests to determine the significance of the observed changes in the F1-score for both groups. Given the size of each group, we use a *Permutation test* for the parametric tests. We employ $n = 5,040$ permutations, representing all possible permutations, to ensure a robust analysis. The significance level is set at $\alpha = 0.05$ and we use *SciPy*'s [62] implementation for each test.

First, we examine the changes within the experimental group. The F1-scores obtained in Round 1 ($p = 0.367$) and Round 2 ($p = 0.197$) are found to follow a normal distribution based on the Shapiro-Wilk test [44]. Additionally, they satisfy the assumption of homoscedasticity, as determined by Levene's non-parametric test [31] ($p = 0.718$). Consequently, we perform a paired t-test to assess the significance of the F1-score changes within the experimental group. The null hypothesis (H_0) states that there is no significant increase in F1-scores between the two rounds for the experimental group. The test yields a p-value of $0.015 < \alpha = 0.05$, with a t-value of -3.79. Therefore, we reject the null hypothesis, indicating a significant difference in F1-scores between the rounds. In addition, Hedge's $g = 1.13$ suggests a substantial difference between the experimental group's performance before and after introducing human feedback. The effect size indicates that the introduction of human feedback

had a significant impact on the performance of the experimental group.

Furthermore, we explore the potential relationship between the number of annotations and the difference in F1-scores. We calculate Spearman's coefficient [56], resulting in $\rho = -0.03$ and $p = 0.963$. These findings indicate a negligible or near-nonexistent relationship between the two variables.

Similarly, the distributions of F1 measures for the control group after each phase were found to be normally distributed ($p = 0.805$ after Round 1 and $p = 0.466$ after Round 2) and exhibited homoscedasticity ($p = 0.6875$). However, the paired t-test yielded a p-value of $0.0625 > \alpha$ with a t-value of -2.497. With the t-test results, we cannot reject the null hypothesis of a significant F1-score change in the control group, *i.e.,* the change is not significant.

Our findings provide evidence that incorporating human feedback enhances the performance of DL models for code smell detection, as shown by the significant improvement in F1-scores. The absence of a significant change in the control group further supports the conclusion that the observed improvements in the experimental group can be attributed to the incorporated feedback.

5.4.2 RQ-2: Whether and to what extent does user feedback influence the accuracy of DL-based detection for individual code smells?

Code smells exhibit distinct attributes that differentiate them from one another due to the variations in their characteristics, patterns, and severity. With this research question, we aim to examine whether the improvement in accuracy, if any, through user feedback is consistent across all considered smells or if it varies for each smell. Furthermore, we seek to quantify the extent of this variation to understand the degree of improvement achieved.

Approach

For this research question, the focus was on evaluating the performance of the model in detecting specific code smells rather than aggregating the results per participant. The goal was to assess how well the model identifies individual code smells and compare its performance across different types of smells.

Similar to the previous research question, the responses for each sample were

curated, and the user response was considered the ground truth. The samples were then divided based on each of the three smells we considered. For each specific code smell, we calculate the following metrics: True Positive (TP), False Negative (FN), False Positive (FP), and True Negative (TN), as in the previous research question. After calculating the metrics for each specific smell, we proceed to compute Precision, Recall, and F1-score individually for each one.

Results and discussion

Table 5.1 summarizes the impact of the user feedback on the performance metrics for the *complex method*, *long parameter list* and smells. The findings demonstrate the effectiveness of the feedback in enhancing the smell detection performance of the trained model.

Table 5.1: Influence of the feedback on classifiers' performance for the considered code smells individually

Smell	**Round 1**			**Round 2**		
	Precision	Recall	F1	Precision	Recall	F1
Multifaceted Abstraction	0.91	0.88	0.88	0.93 ↑	0.90 ↑	0.91 ↑
Complex Method	0.86	0.77	0.81	0.86 –	0.81 ↑	0.83 ↑
Long Parameter List	0.69	0.88	0.77	0.74 ↑	0.95 ↑	0.83 ↑

Before incorporating the feedback loop, the model achieved a precision of 0.91, a recall of 0.88, and an F1-score of 0.88 for the code smell. Following the feedback loop, there was a slight improvement across all metrics, with the precision increasing to 0.93, recall to 0.90, and the F1-score to 0.91.

For the *complex method* code smell, the initial performance showed a precision of 0.86, a recall of 0.77, and an F1-score of 0.81. In the second round, there was a marginal enhancement in the recall and F1-score, while the precision remained unchanged at 0.86.

Similarly, the model's performance for the *long parameter list* smell demonstrated

a precision of 0.69, a recall of 0.88, and an F1-score of 0.77 for the base model. Subsequently, introducing human feedback yielded significant improvements, with the precision increasing to 0.74, recall to 0.95, and the F1-score to 0.83. The results indicate that incorporating human feedback positively influenced the models' code smell detection capabilities. The inclusion of human feedback resulted in improved F1 scores, indicating enhanced precision and recall tradeoffs in the models' detection of code smells. Notably, the *long parameter list* code smell exhibited the most substantial improvement, followed by and *complex method*. This suggests that the effectiveness of the feedback loop may vary depending on the specific code smell being detected.

The variation in performance after incorporating human feedback across different code smells can be attributed to several factors. The number of training samples for each smell influences the initial performance, with larger sample sizes potentially resulting in higher performance. The complexity and characteristics of each smell also play a role, with some smells being more straightforward to detect and classify accurately. For example, detecting *complex method* smell is considerably more difficult than *long parameter list* smell due to larger and more complex code snippets to process. Due to the complexity, it requires a greater number of training samples to learn to classify correctly.

5.4.3 RQ-3: Do feedback loops improve the accuracy of predicted smells at the cost of subjectivity of the model?

Deep learning models typically rely on a large dataset to learn and generalize patterns. However, during the fine-tuning process with extensive data, the subjectivity of individual users can be lost, resulting in a more generalized model that may not capture the unique perspectives and preferences of each user.

The research question aims to validate that by leveraging user feedback, we can achieve accuracy improvements in the models while ensuring that the subjectivity of each user for each code smell remains intact. By considering the specific feedback provided by users, the models can be customized to better align with their individual preferences, expertise, and contextual understanding.

Approach

The subjectivity of code smells lies in the fact that the threshold for a code snippet to become smelly is different for every developer. Statistical coefficients like the Kappa-Cohen score [23], Gwett's AC1/AC2 [17], and Krippendorff's Alpha [24] are used to measure inter-annotator agreements.

While all these metrics can be used to measure inter-annotator agreements, the Kappa-Cohen coefficient works best when used for comparison between two annotators [4]. Moreover, Gwett's AC1/AC2 and the Kappa-Cohen coefficient all possess a significant bias when there are a large number of non-random missing values [23].

For this reason, we selected Krippendorff's Alpha coefficient to measure the inter-annotator agreement values. Krippendorff's Alpha works well with any number of annotators and can handle the missing data well [3]. The formula for Krippendorff's Alpha is given as:

$$\text{Alpha} = 1 - \frac{D_o}{D_e}$$

where D_o is the observed disagreement and D_e is the expected disagreement.

We generated a matrix of all the samples annotated by two or more annotators before and after fine-tuning the model. We then constructed a two-dimensional matrix and passed it to the Krippendorff Python library.

Results and discussion

Table 5.2 presents the Krippendorff values before and after fine-tuning.

Table 5.2: Krippendorff's Alpha coefficient values

Round	Smell	Alpha value
Round 1	Complex Method	0.44
	Multifaceted Abstraction	–
	Long Parameter List	0.35
	Overall	**0.46**
Round 2	Complex Method	0.51
	Multifaceted Abstraction	–
	Long Parameter List	0.36
	Overall	**0.48**

In Round 1, the *complex method* had an Alpha value of 0.44, while we did not have enough samples with common annotations for . The *long parameter list* had an Alpha value of 0.35. The overall Alpha value for Round 1 was 0.46. In Round 2, the *complex method* had an Alpha value of 0.51, and the *long parameter list* had an Alpha value of 0.36. The overall Alpha value for Round 2 was 0.48. With these results, it is reasonable to state that the subjectivity is not *diluted* in the updated models. Despite the increase in the model's performance, the continued presence of relatively low Krippendorff's alpha values indicates that the subjective nature and variability among developers in their assessments of code smells persist. In addition, this complements our insight for Research Question 2; the fact that certain smell models showed more substantial improvements (*long parameter list* vs *complex method*) with the consistent ranking of subjectivity, indicates that they have become more attuned to the subjective assessments of developers for those specific smells.

5.5 Threats and validity

In this section, we address the potential threats to the validity of this thesis.

5.5.1 Internal validity

Internal validity threats concern the ability to conclude from our experimental results.

While building the dataset, in phase 2 of manual annotation, we invited volunteers with at least a basic understanding of Java programming language and object orientation concepts. We advertised the invitation on social media professional channels (Twitter and LinkedIn). Given the anonymity of the exercise, we do not have any mechanism to verify the assumption that the participants have sufficient knowledge to attempt the annotations. However, we offered all the major participants (with at least 50 annotations) to include them as contributors to the dataset; we perceive such a measure would have motivated the annotators to perform the annotations to the best of their abilities. Additionally, we configured TAGMAN to obtain two annotations per sample so that we can reduce the likelihood of a random annotation.

While conducting the experiment, to address potential *internal validity* threats, we employed random assignment of participants to the control and experimental groups. This helps mitigate selection bias by ensuring that any differences in the results between the groups are more likely due to the introduction of human feedback rather than pre-existing differences. Additionally, we controlled for the potential influence of maturation by limiting each experiment session to a maximum of 90 minutes. Moreover, to ensure the validity of the tools used to capture feedback, we conducted a pilot study to validate their effectiveness and reliability. This helped us to mitigate any potential biases or limitations associated with the data collection instruments.

5.5.2 External Validity

External threats are concerned with the ability to generalize our results. The proposed dataset is for snippets written in Java. However, our code annotation tool is generic and it can be used to annotate snippets from any programming language. Additionally, scripts used to generate individual snippets can be customized to use any other external tool for splitting the code into methods and classes. Furthermore, the thresholds used in the annotation process to filter snippets based on low and high thresholds of a metric are configurable.

During the experiment, we provided detailed information about the participants'

characteristics, such as being graduate students enrolled in Master's or Ph.D. programs in Computer Science, and the source of recruitment through the university's mailing lists. This helps readers assess the generalizability of the findings within the target population. We also described the study setting, being conducted in a university environment, and provided contextual information to aid readers in evaluating the transferability of the findings to similar settings. In addition, the provision of a replication package, including the data and code used in the study, contributes to the external validity of the research.

5.5.3 Conclusion validity

To manage *conclusion validity* threats, we aimed for an adequate sample size and performed statistical analysis using well-known statistical tests. By doing so, we aimed to minimize the risk of concluding a false effect. We controlled the significance level (alpha) to manage the risk of Type I errors. Furthermore, by achieving sufficient statistical power, we aimed to mitigate Type II errors. Finally, to address potential confounding variables, we employed randomization in the assignment of participants.

Chapter 6

Conclusions

This chapter summarizes the methodology, findings, and key contributions of the thesis. In addition, we propose potential future works that might complement this thesis for a better understanding of subjectivity in different code smells and their detection.

6.1 Summary of contribution

Deep learning models tend to ignore the subjective nature of code smells, and they fail to consider individual differences in how people perceive them. To improve the accuracy of these models while still retaining the subjective aspect, it's essential to leverage human feedback. In this thesis, we present an approach to address this issue..

To address the problem of a lack of a sizable dataset, We offer DACOS—a manually annotated code smell dataset containing $10,267$ annotations for $5,192$ subjective code snippets. We also provide a large DACOSX dataset containing definitely benign and definitely smelly snippets in addition to those present in DACOS. We offer TAGMAN, a code annotation application that can be reused in similar contexts.

We selected a relatively small set of code smells to consider in the dataset because having more annotations for a smell is better than having a small number of annotations per smell. Also, we chose a set of smells not covered by the existing code smells dataset. We configured TAGMAN to obtain two annotations per sample.

Next, we present a comprehensive framework that addresses the issue of subjectivity in code smell detection and enables personalized and accurate predictions. Our approach combines deep learning techniques, user feedback, and a containerized deployment architecture for a locally run web-server to address the subjectivity in the smell detection We train a baseline DL model using the previously created DACOS dataset. We integrate our DL model into a Docker container behind a web server to offer smell predictions and retrain the model quickly as and when required. Our

initial model predicts code smells that we show users in the IntelliJ IDEA environment with the help of our plugin *TagCoder*. *TagCoder* offers the smells to the users and collects their feedback. We train the deployed DL model using the collected user feedback. The fine-tuning allows the model to learn and adapt to individual user preferences and enhances the accuracy of smell detection. Our experiments to evaluate the proposed approach show that fine-tuning the DL model using collected user feedback outperforms the base model for all the participants. Moreover, the performance of the fine-tuned model is superior to the base model for all three smells considered. This performance improvement is achieved for each participant while considering their feedback and maintaining the customization of the model's behavior for each participant.

6.2 Limitations

We configured TAGMAN to obtain two annotations per sample. Though it improves the reliability of the dataset, one may argue that it may introduce a situation where the annotations are contradictory.

Moreover, the experiment was conducted with 14 participants. The results can be more grounded with a significantly larger pool of participants. The experiment was also conducted with samples from the project using Java language. Replicating the experiment with different languages could help discover the nuances of subjectivity related to the semantics of the particular language.

The thesis also observed variations in performance improvements across different code smells, suggesting that specific characteristics of code may influence the impact of the feedback loop smells. This highlights the need for further investigation into the factors contributing to these variations, potentially leading to more tailored and effective code smell detection approaches.

6.3 Directions of future research

In this section, we present some ideas for direction towards future research using our thesis as a base.

6.3.1 Language extension for the dataset

We have built the dataset using samples of Java language. However, TAGMAN is entirely configurable, and it would be possible to extend it to use the platform for collecting samples for other programming languages like Python or NodeJS.

6.3.2 Increasing the number of annotations

Increasing the number of annotations for each sample could help mitigate the scenario where there are conflicting numbers of samples for annotations.

6.3.3 Serverless architecture

For the experiment, we have built a server that needs to be run on the local machine to collect user feedback. This server requires a specific hardware configuration to run effectively. Further research can be undertaken to create an approach that works built into the IDE that would not require a server to run. This would help increase the number of participants as the hardware requirements would be lower.

6.3.4 Publishing the created tool to the IDE marketplace

Currently, the plugin created to capture user feedback has not been submitted to the marketplace for the IntelliJ tool. Publishing the plugin to the marketplace for different IDEs would allow more people to participate in the experiment.

6.3.5 Replicating the experiment with more participants

We conducted the experiment with 14 participants due to a lack of resources and time. Replicating the experiment with a higher number of participants from varying backgrounds could provide better insights into the findings of the experiment and could highlight potential differences between people from varying educational and professional backgrounds.

6.3.6 Expanding the number and types of code smells

Expanding the scope of the experiment to include a wider variety of code smells may provide more comprehensive insights. By incorporating various kinds of smells, especially those that are not currently featured in the dataset, the experiment may evaluate the model's effectiveness across a broader spectrum of smells. This diversification in code smells would enable a more thorough understanding of the model's capabilities and limitations, and how they vary across different categories of code smells.

Bibliography

[1] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. Live software development: Tightening the feedback loops. In *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*, Programming '19, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Mamdouh Alenezi and Mohammad Zarour. An empirical study of bad smells during software evolution using designite tool. *i-Manager's Journal on Software Engineering*, 12(4):12, 2018.

[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 378–389, 2016.

[4] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 143–153, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code, 2018.

[6] Francesca Arcelli Fontana and Marco Zanoni. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, 128:43–58, 2017.

[7] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138, 2019.

[8] R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, 1985.

[9] Antoine Barbez, Foutse Khomh, and Yann-Gaël Guéhéneuc. A machine-learning based ensemble method for anti-patterns detection. *Journal of Systems and Software*, 161:110486, 2020.

[10] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. *Engineering Self-Adaptive Systems through Feedback Loops*, pages 48–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[11] André Eposhi, Willian Oizumi, Alessandro Garcia, Leonardo Sousa, Roberto Oliveira, and Anderson Oliveira. Removal of design problems through refactorings: Are we looking at the right symptoms? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 148–153, 2019.

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[14] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley Professional, 1 edition, 1999.

[15] Martin Fowler. *Refactoring*. Addison-Wesley Professional, 2018.

[16] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a catalogue of architectural bad smells. In *Architectures for Adaptive Software Systems: 5th International Conference on the Quality of Software Architectures, QoSA 2009, East Stroudsburg, PA, USA, June 24-26, 2009 Proceedings 5*, pages 146–162. Springer, 2009.

[17] Kilem Li Gwet. Computing inter-rater reliability and its variance in the presence of high agreement. *British Journal of Mathematical and Statistical Psychology*, 61(1):29–48, 2008.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[19] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Christoph Treude. The impact of automated feature selection techniques on the interpretation of defect models. *Empirical Software Engineering*, 25(5):3590–3638, 2020.

[20] Amandeep Kaur, Sushma Jain, and Shivani Goel. A support vector machine based approach for code smell detection. In *2017 International Conference on Machine Learning and Data Science (MLDS)*, pages 9–14, 2017.

[21] Marouane Kessentini and Ali Ouni. Detecting android smells using multi-objective genetic programming. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 122–132, 2017.

[22] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. Bdtex: A gqm-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559–572, 2011. The Ninth International Conference on Quality Software.

[23] Helena C Kraemer. Kappa coefficient. *Wiley StatsRef: statistics reference online*, pages 1–4, 2014.

[24] Klaus Krippendorff. Computing krippendorff's alpha-reliability. 2011.

[25] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012.

[26] Tomasz Lewowski and Lech Madeyski. How far are we from reproducible research on code smell detection? a systematic literature review. *Information and Software Technology*, 144:106783, 2022.

[27] H. Li, Z. Liu, H. Zhu, H. Wang, and Z. Yang. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. *IEEE Transactions on Software Engineering*, 43(4):335–355, 2017.

[28] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. Deep learning based code smell detection. *IEEE transactions on Software Engineering*, 47(9):1811–1837, 2019.

[29] Lech Madeyski and Tomasz Lewowski. Mlcq: Industry-relevant code smell data set. In *Proceedings of the Evaluation and Assessment in Software Engineering*, EASE '20, page 342–347, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. Support vector machines for anti-pattern detection. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 278–281, 2012.

[31] Benoit Mandelbrot. Contributions to probability and statistics: Essays in honor of harold hotelling (ingram olkin, sudhist g. ghurye, wassily hoeffding, william g. madow, and henry b. mann, eds.). *SIAM Review*, 3(1):80–80, 1961.

[32] Radu Marinescu. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 701–704. IEEE, 2005.

[33] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[34] T Mens and T Tourwe. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, February 2004.

[35] Himesh Nandani, Mootez Saad, and Tushar Sharma. Dacos-a manually annotated dataset of code smells. *ArXiv*, abs/2303.08729, 2023.

[36] Willian Oizumi, Leonardo Sousa, Anderson Oliveira, Luiz Carvalho, Alessandro Garcia, Thelma Colanzi, and Roberto Oliveira. On the density and diversity of degradation symptoms in refactored classes: A multi-case study. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 346–357. IEEE, 2019.

[37] Daniel Oliveira, Wesley K. G. Assunção, Alessandro Garcia, Baldoino Fonseca, and Márcio Ribeiro. Developers' perception matters: machine learning to detect developer-sensitive smells. *Empirical Software Engineering*, 27(7), October 2022.

[38] Jeroen Ooms and Facebook, Inc. *graphql: A GraphQL Query Parser*, 2023. https://docs.ropensci.org/graphql/, https://graphql.org (upstream) https://github.com/ropensci/graphql (devel).

[39] Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Landfill: An open dataset of code smells with public evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 482–485, 2015.

[40] Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Landfill: An open dataset of code smells with public evaluation. May 2015.

[41] PMD. *PMD Source Code Analyzer*. PMD, 2021.

[42] R. Ren, C. Nistor, L. Schumacher, and B. Meyer. Identifying self-admitted technical debt: A machine learning approach. *Empirical Software Engineering*, 24(5):3204–3242, 2019.

[43] Yvan Saeys, Thomas Abeel, and Yves Van de Peer. Robust feature selection using ensemble feature selection techniques. In Walter Daelemans, Bart Goethals, and Katharina Morik, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 313–325, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[44] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[45] Tushar Sharma. Designite: A customizable tool for smell mining in c# repositories. 2017.

[46] Tushar Sharma. Codesplitjava, February 2019. https://github.com/tushartushar/CodeSplitJava.

[47] Tushar Sharma. searchgithubrepo - search github repositories. `https://github.com/tushartushar/search-repo`, 2022.

[48] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software*, 176:110936, 2021.

[49] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 189–200, New York, NY, USA, 2016. Association for Computing Machinery.

[50] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. House of cards: Code smells in open-source c# repositories. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 424–429, 2017.

[51] Tushar Sharma and Marouane Kessentini. Qscored: A large dataset of code smells and quality metrics. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 590–594, 2021.

[52] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. Designite - a software design quality assessment tool. In *2016 IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities (BRIDGE)*, pages 1–4, 2016.

[53] Tushar Sharma, Paramvir Singh, and Diomidis Spinellis. An empirical investigation on the relationship between design and architecture smells. *Empirical Software Engineering*, 25(5):4020–4068, 2020.

[54] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158 – 173, 2018.

[55] SonarSource. *SonarQube*. SonarSource, 2021.

[56] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 100(3/4):441–471, 1987.

[57] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[58] Anderson Uchôa, Caio Barbosa, Willian Oizumi, Publio Blenilio, Rafael Lima, Alessandro Garcia, and Carla Bezerra. How does modern code review impact software design degradation? an in-depth empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 511–522, 2020.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[60] Jennifer Wortman Vaughan. Making better use of the crowd: How crowdsourcing can advance machine learning research. *Journal of Machine Learning Research*, 18(193):1–46, 2018.

[61] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23:501–532, 2016.

[62] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[63] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[64] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *CoRR*, abs/2109.00859, 2021.

[65] Jesse Yli-Huumo, Andrey Maglyas, and Kari Smolander. The sources and approaches to management of technical debt: A case study of two product lines in a middle-size finnish software company. In Andreas Jedlitschka, Pasi Kuvaja, Marco Kuhrmann, Tomi Männistö, Jürgen Münch, and Mikko Raatikainen, editors, *Product-Focused Software Process Improvement*, pages 93–107, Cham, 2014. Springer International Publishing.

Appendices

Appendix A

Complementary Materials

This appendix presents the materials and tools used during this thesis.

A.1 Replication Package

The framework, tools, scripts, analysis, and generated anonymized data can be found online at:

- **https://github.com/SMART-Dal/DLFeedback**

  This repository contains all the code developed for this thesis. The repository is divided into small projects. Each project focuses on a different aspect of code smell detection used in the thesis. Each project contains a readme file for instructions to setup the project.

- **https://github.com/SMART-Dal/Tagman**

  This repository contains the code shown to all the users during both phases of the controlled experiment.

- **https://zenodo.org/record/7570428**

  This artefact contains the built dataset and instructions to replicate the dataset construction process.