university of groningen

faculty of science and engineering

Applying Graph Learning For Technical Debt Detection

Andrei-Ștefan Istudor

university of groningen

faculty of science and engineering

University of Groningen

Applying Graph Learning For Technical Debt Detection

Bachelor's Thesis

To fulfill the requirements for the degree of
Bachelor of Science in Computing Science
at the University of Groningen under the supervision of
Dr. D. (Daniel) Feitosa (Computing Science, University of Groningen)
and
J. (Jesse) Maarleveld, MSc. (Computing Science, University of Groningen)

Andrei-Ștefan Istudor (S4675908)

July 21, 2024

Abstract

In the domain of software engineering, the increasing presence of Technical Debt (TD) poses a significant challenge for the quality and long-term maintenance of modern software systems. Technical Debt refers to the extra cost and effort required, due to early sub-optimal decisions in software development. This research project seeks to apply graph learning techniques in order to identify Technical Debt within software projects. We developed a pipeline called 'Debtective' for preprocessing code samples and for model training. This allowed us to investigate the effectiveness of different model architectures in detecting Technical Debt from a chosen dataset. We selected and transformed code samples for two SonarQube code smells into Code Property Graphs (CPGs), and we used them in training and evaluating a series of models that we created. The results show that some models perform better for a code smell compared to the other, and some models perform great generally, for both code smells. Overall, the results of our study confirm the effectiveness of graph-based models in detecting Technical Debt (TD), and highlight its potential for future TD detection research. Through this proposed approach, we hope to enhance the identification of TD and offer new methodologies for its detection, potentially benefiting both the industry practices and the academic research in this area.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisors, Dr. Daniel Feitosa and Jesse Maarleveld. Without your guidance, this research project would not have been possible.

I would like to express my heartfelt appreciation to my family: my parents, Vasile and Lavinia, and my sister, Ana. Your unconditional support and unfailing love have been the cornerstone of my academic journey.

To my girlfriend, Petya, your encouragement and support throughout my degree has been essential.

Finally, I would like to thank all of my friends, who have contributed to my journey in various ways. Your support has been greatly appreciated.

Contents

List of Figures

List of Tables

1 | Introduction

In the rapidly evolving field of software development, the concept of Technical Debt (TD) has emerged as a critical consideration for both developers and researchers [12]. In software-intensive systems, Technical Debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible [3]. Technical Debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability [3]. As software systems become more complex, the need for developing tools for the management of Technical Debt increased [3].

Graph Neural Networks (GNNs) have demonstrated significant success across various domains due to their capability to model complex relational structures. As an example, they have improved tasks such as node classification, link prediction, and community detection, showing superior performance and scalability compared to traditional methods [10]. This success is highlighted in the comprehensive survey on graph representation learning, which highlights the transformative potential of GNNs in processing and understanding structured data in the context of diverse applications [10].

The application of graph learning specifically for technical debt detection is an area open to exploration. There is a series of existing studies that have laid the groundwork by employing graph-based models for vulnerability detection and defect prediction, and these demonstrated an increased accuracy and efficiency compared to traditional methods [25, 24, 5]. For example, the Bidirectional Graph Neural Network for Vulnerability Detection (BGNN4VD) approach uses advanced graph learning to better identify whether a piece of code is vulnerable or non-vulnerable [5]. This suggests that graph-based models and graph neural networks are possibly suitable for detecting Technical Debt as well [5].

The motivation for this research project comes from the significant challenges that the software development community is facing in identifying and managing Technical Debt. As software systems grow in complexity, so does the TD, making its detection and management increasingly difficult. This problem is not only of academic interest, but also significantly concerning for the industry. It directly impacts the cost and effort needed for creating high-quality long-term maintainable software projects. Hence, there is a serious

need for innovative approaches that can enhance the detection of technical debt, offering better results, compared to traditional methods [25].

By exploring and applying graph learning techniques in a Technical Debt setting, this research produced a series of advancements in the detection and management of TD. This will not only benefit the research community by contributing to the current level of knowledge in this area, but it will also have a practical impact on the industry by providing tools and methodologies for more efficient Technical Debt detection. To conclude with, exploring this path might open up interdisciplinary research opportunities at the intersection of the field of software engineering and machine learning.

Contributions of this research project:

- Evaluation of Graph-Based Models on TD detection:
  Demonstrating the applicability and effectiveness in identifying TD for different graph learning model architectures that we used.
- Methodological Advancements:
  We built a pipeline called 'Debtective' for investigating the effectiveness of graph learning in TD detection, which will contribute with methodological advancements in TD detection and management.

In Chapter 2 we will be looking over the background (or state of the art). In Chapter 3 we will be discussing the methodology for this research project. This includes the dataset selection and preprocessing, the choice of graph representations and, lastly, the construction of the pipeline, or the tool that was created. Next, we will be analysing the results in Chapter 4, delving into discussion in Chapter 5, and, in the end, drawing the conclusions of this project in Chapter 6.

2 | Background

2.1 Technical Debt Management

The management of Technical Debt (TD) is an area with a growing importance [12], reflecting the trade-offs between rapid deployment and long-term maintainability of software. Developers and teams often need to deliver new software within strict deadlines, compromising quality and maintainability [21, 6]. With the ever increasing complexity of modern software, there is a need for tools and methodologies for detecting and managing Technical Debt. Too much TD causes software to become difficult to maintain, and produces future costs and complexities which affect the amount of additional effort in the maintenance process of existing software [21]. The technical quality of source code is an important determinant for software maintainability in the long run [4].

2.2 Graph Neural Networks

Graph Neural Networks (GNNs) are a sub-class of deep learning models, which are specifically designed to work with graph-structured data. GNNs are used for learning node embeddings based on neighboring information through a combination of message passing and aggregation mechanisms. They show significant expressive capacity in representing graph embeddings in an inductive learning manner [10]. The development of GNNs can be traced back to the 2000s, with some of the early models using recurrent neural network architectures, in order to learn node embeddings via recurrent layers. These recurrent GNNs (RGNNs) use the same weights in each hidden layer and operate recursively until they converge. This enables the model to capture dependencies within the structure of the graph representations. However, a limitation of RGNNs is their potential incapability of distinguishing between local and global structures because of the use of identical weights across layers [10].

The evolution of GNNs has led to the introduction of convolutional operators with different weights in each hidden layer. This has proven more efficient in capturing and distinguishing local and global structures [10]. This advancement led to the appearance of Convolutional Graph Neural Networks

(CGNNs), including their various variants, such as spectral CGNNs, spatial CGNNs, and attentive CGNNs [10]. Spectral CGNNs make use of the spectral domain for convolution operations, spatial CGNNs work directly on the graph domain, and attentive CGNNs incorporate attention mechanisms to focus on the most relevant parts of the graph representations given [10]. Despite their advancements and advanced use-cases, GNNs also face a series of limitations. One notable challenge is called the 'over-smoothing' problem, where the node representations become indistinguishable when multiple GNN layers are stacked together [10]. Additionally, GNNs can also suffer from noise introduced by neighbor nodes, which can degrade the quality of the learnt embeddings [10].

A series of interesting Graph Neural Networks for the scope of this study are Graph Convolutional Networks (GCNs) [14], Graph Attention Networks (GATs) [22], and Graph Sample and Aggregation (GraphSAGE) [9]. We will further discuss these three:

2.2.1 Graph Convolutional Networks (GCNs)

Graph Convolutional Networks (GCNs) [14] are a sub-type of Graph Neural Networks (GNNs), designed to perform convolution operations directly on graph data. This allows GCNs to effectively capture and use the structural information of graph representations, making them suitable for tasks that involve complex relational data [14], such as detecting Technical Debt (TD) from graph representations of code samples. By applying a layer-wise propagation rule, GCNs aggregate information from the neighbors of a node, transforming and combining it with the node's own features [14].

Mathematically, the layer-wise propagation rule [14] for GCNs is defined as follows:

$$H^{(l+1)} = \sigma(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(l)}W^{(l)}) \tag{2.1}$$

Here, $\tilde{A} = A + I$ is the adjacency matrix with added self-loops, $\tilde{D}$ is the degree matrix of $\tilde{A}$, $H^{(l)}$ is the input feature matrix at layer l, $W^{(l)}$ is the trainable weight matrix at layer l, and σ is the activation function [14]. The trainable parameters in GCNs are the weight matrices $W^{(l)}$ for each one of the layers. The node features are updated through this convolution operation, which aggregates and transforms the features from the neighbors [14].

The advantages of using GCNs lie in their ability to incorporate both node features and graph structure into the learning process. This leads to better performance on tasks such as node classification. GCNs are also less prone to overfitting on sparse data due to the regularization effects of neighborhood aggregation [14]. On the other hand, GCNs have several limitations, such as sensitivity to graph connectivity and potential issues with over-smoothing, where repeated application of graph convolutions can make node features

indistinguishable [14]. These advantages and limitations are also relevant to other graph-based models like GAT [22] and GraphSAGE [9].

2.2.2 Graph Attention Networks (GATs)

Graph Attention Networks (GATs) [22] use attention mechanisms to address the limitations of previously mentioned graph convolutional methods. GATs enhance node classification on graph-structured data by applying masked self-attentional layers. These layers assign different importance to the neighboring nodes of the node in question [22]. This is achieved through attention coefficients that weigh the significance of each neighbor's features, providing a more flexible and dynamic method for feature aggregation without requiring prior knowledge of the graph structure [22].

The attention mechanism in GATs [22] can be mathematically represented as:

$$e_{ij} = \text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_j]) \tag{2.2}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})} \tag{2.3}$$

$$\mathbf{h}'_i = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W}\mathbf{h}_j\right) \tag{2.4}$$

Here, $\mathbf{a}$ is a trainable weight vector, $\mathbf{W}$ is a trainable weight matrix, e_{ij} is the attention score between node i and node j, α_{ij} is the normalized attention coefficient, and σ is the activation function [22].

The main advantages of using Graph Attention Networks include the ability to handle graphs with varying neighborhood sizes and their efficiency due to the parallelizable nature of their attention mechanisms [22]. GATs are computationally expensive to scale to large graphs while maintaining computational efficiency and may require specialized hardware for optimal performance [22]. Despite these challenges, GATs can be effective for tasks such as detecting technical debt in graph representations of code by focusing on the most relevant nodes and relationships. Similar to GCNs and GraphSAGE, these benefits and limitations are shared [22].

2.2.3 Graph Sample and Aggregation (GraphSAGE)

GraphSAGE (Graph Sample and Aggregation) [9] represents an inductive framework designed for scalable and generalizable graph representation learning on large graphs. Unlike other methods that require the entire graphs during training, GraphSAGE can generate embeddings for unseen nodes by sampling and aggregating features from a node's local neighboring nodes [9].

This is done through different aggregating functions such as mean, LSTM, and pooling. By learning an embedding function that can be applied to new nodes, GraphSAGE enables efficient and scalable learning on dynamic and large-scale graphs [9].

The aggregation function in GraphSAGE [9] can be represented as:

$$\mathbf{h}_v^{(k)} = \sigma\left(\mathbf{W}^{(k)} \cdot \text{AGGREGATE}^{(k)}\left(\left\{\mathbf{h}_u^{(k-1)}, \forall u \in \mathcal{N}(v)\right\}\right)\right) \tag{2.5}$$

Here, $\mathbf{h}_v^{(k)}$ is the representation of node v at layer k, $\mathbf{W}^{(k)}$ is the trainable weight matrix at layer k, and $\text{AGGREGATE}^{(k)}$ is a function that aggregates the features of the neighbors u of node v [9].

The advantages of using GraphSAGE are mainly its ability to handle large and dynamic graphs and its flexibility in using different aggregation methods to capture various types of neighborhood information [9]. On the other hand, one of the challenges with GraphSAGE is the potential loss of global structural information due to its local sampling approach [9]. By analyzing and aggregating local neighborhood information in code graphs, GraphSAGE can uncover patterns and dependencies that might indicate areas of Technical Debt.

2.2.4 Dropout

With large neural networks, however, the obvious idea of averaging the outputs of many separately trained nets is prohibitively expensive. Combining several models is most helpful when the individual models are different from each other and in order to make neural net models different, they should either have different architectures or be trained on different data. Training many different architectures is hard because finding optimal hyperparameters for each architecture is a daunting task and training each large network requires a lot of computation. Moreover, large networks normally require large amounts of training data and there may not be enough data available to train different networks on different subsets of the data. Even if one was able to train many different large networks, using them all at test time is infeasible in applications where it is important to respond quickly [19].

Dropout is a technique that addresses both these issues. It prevents overfitting and provides a way of approximately combining exponentially many different neural network architectures efficiently. The term "dropout" refers to dropping out units (hidden and visible) in a neural network. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections, as shown in Figure 1. The choice of which units to drop is random. In the simplest case, each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or can simply be set at 0.5, which seems to be close to

optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5 [19].

Applying dropout to a neural network amounts to sampling a "thinned" network from it. The thinned network consists of all the units that survived dropout. A neural net with n units, can be seen as a collection of 2n possible thinned neural networks. These networks all share weights so that the total number of parameters is still $O(n^2)$, or less. For each presentation of each training case, a new thinned network is sampled and trained. So training a neural network with dropout can be seen as training a collection of 2n thinned networks with extensive weight sharing, where each thinned network gets trained very rarely, if at all [19].

2.2.5 Rectified Linear Units (ReLU)

ReLU is an activation function which has strong biological and mathematical underpinning. In 2011, it was demonstrated to further improve training of deep neural networks. It works by thresholding values at 0, i.e. $f(x) = max(0, x)$. Simply put, it outputs 0 when $x < 0$, and conversely, it outputs a linear function when $x \geq 0$ [1]. ReLU is conventionally used as an activation function for neural networks, with softmax being their classification function. Then, such networks use the softmax cross-entropy function to learn the weight parameters θ of the neural network [1].

2.3 Graph Representations

The employment of various graph representations plays a fundamental role in the detection of Technical Debt using graph learning. Abstract Syntax Trees (ASTs) [20], for instance, provide a tree-based representation of the syntactic structure of the source code. This makes them invaluable in the analysis and identification of potential technical debt. In a similar way, Control Flow Graphs (CFGs) [2] and Data Flow Graphs (DFGs) [11] offer insights into the flow of control and data within software, further enriching the context for graph-based analysis. Lastly, composite graphs, which integrate ASTs, CFGs, and DFGs, offer a holistic view of the program's structure, allowing for a more comprehensive analysis. Such graph representations are crucial for capturing the multifaceted nature of Technical Debt.

An intermediate program representation, called the program dependence graph (PDG), makes explicit both the data and control dependencies for each operation in a given program. Data dependencies have been used to represent only the relevant data flow relationships of a program. Control dependencies are introduced to analogously represent only the essential control flow relationships of a program. Control dependencies are derived from the usual control flow graph [8].

- **Abstract Syntax Trees (ASTs) [20]:**
  - **Description:** Represent the syntactic structure of source code in a tree format, where each node denotes a construct occurring in the source code.
  - **Nodes:** Represent programming constructs such as expressions, statements, and declarations.
  - **Edges:** Represent the hierarchical syntactic relationships between these constructs.
- **Control Flow Graphs (CFGs) [2]:**
  - **Description:** They depict the order in which individual instructions or statements of a program are executed.
  - **Nodes:** Represent basic blocks, which are code sequences with no branches except at the entry and exit.
  - **Edges:** Represent the flow of control between these basic blocks.
- **Data Flow Graphs (DFGs) [11]:**
  - **Description:** They show the flow of data within the program, focusing on the dependencies between data-producing and data-consuming operations.
  - **Nodes:** Represent operations or computations in the program.
  - **Edges:** Represent data dependencies, indicating which operation's output is used as input for another operation.
- **Program Dependence Graphs (PDGs) [8]:**
  - **Description:** Explicitly represent both data and control dependencies for each operation in a program.
  - **Nodes:** Represent individual operations or instructions in the program.
  - **Edges:** Represent both data and control dependencies between operations.

Control dependence refers to the execution of one instruction which depends on the outcome of a previous control instruction, like a conditional statement or a loop. This differs from control flow, which refers to the sequence in which all instructions execute, typically shown in a Control Flow Graph (CFG) [2]. In a similar manner, data dependence indicates that one instruction needs data produced by another instruction, whereas data flow is portraying how data moves and changes throughout the program, as it is depicted in Data Flow Graphs (DFGs) [11]. In short, control dependence fo-

cuses on decision points in the code, while control flow outlines their execution order. Data dependence focuses on data requirements, while data flow follows the data movement.

One specific graph representation that we would like to focus our attention on is Code Property Graphs (CPGs) [23]. They were first introduced by F. Yamaguchi et al. in 2014 [23]. CPGs are a complex representation of source code that combines elements of three classical code analysis representations: Abstract Syntax Trees (ASTs) [20], Control Flow Graphs (CFGs) [2], and Program Dependence Graphs (PDGs) [8]. Their unified structure allows for modeling of various patterns through graph traversals. This integrated approach enables the detection of potential code smells with high accuracy. Moreover, CPGs can be implemented using existing graph database technologies, which allows for efficient handling and querying of large codebases [23]. This efficiency is demonstrated by the successful identification of previously unknown vulnerabilities in complex software like the Linux kernel [23]. CPGs also come with a series of limitations. One significant disadvantage is the complexity of constructing and maintaining these graph representations, especially for very large and evolving codebases [23]. Another limitation is the reliance on static code analysis, which may not capture all runtime behaviours and interactions. This can lead to potentially missing vulnerabilities that only manifest under specific conditions [23].

Code Property Graphs allow us to use a single graph representation that captures both structure and semantics. This proves to be versatile for our project, since we need to run multiple analysis tasks for different types of code smells. The example of such utilisation can be seen in Figures 2.1 and 2.2, from "Modeling and Discovering Vulnerabilities with Code Property Graphs", by F. Yamaguchi et al. [23].

```
void foo()                          1
{                                   2
    int x = source();               3
    if (x < MAX)                    4
    {                               5
        int y = 2 * x;              6
        sink(y);                    7
    }                               8
}                                   9
```

Figure 2.1: Example code sample [23].

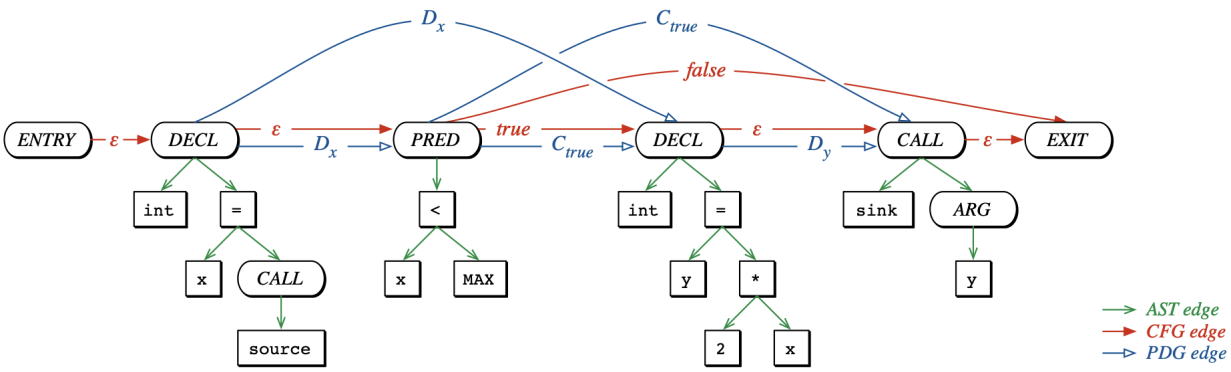

Figure 2.2: Code property graph for the example code sample [23].

3 | Methodology

3.1 Objectives & Research Questions

The goal of this project is to implement graph learning techniques from the relevant literature, apply them in a technical debt setting, and check whether these techniques are successful in detecting technical debt in existing technical debt datasets. Therefore, summarising our goal, the objectives were finding a suitable dataset, preprocessing the dataset, building a pipeline, and using the pipeline for running our analysis and extracting results. These objectives provide the backbone of our project, and in the end we were able to analyse a series of different model architectures that were able to detect technical debt from samples in the form of graph representations.

The key requirements include identifying and obtaining a suitable dataset, based on a series of criteria explained in the next subsection, preprocessing the data to ensure its usability and relevance, transforming sample code files to graph representation, constructing an analysis pipeline, and developing a model capable of detecting technical debt through graph representations. By clearly defining these requirements, we ensure that our approach is systematic and aligns with the goal of the project.

To delve deeper into our investigation, we will pose the following research question:

- **RQ: How effective are GCN, GAT and GraphSAGE graph-based models in detecting technical debt in existing technical debt datasets?**

This research question refers to the general applicability of the GCN, GAT and GraphSAGE graph-based models in this scenario and measuring the effectiveness in detecting technical debt in some use-cases on which the training is done. It is also the common part, or the collaboration point between this research project and my colleague's. By addressing this research question, the aim is to provide a comprehensive evaluation of the graph-based models' capabilities in a technical debt setting. Such insights are very important in developing more accurate and efficient technical debt detection methods, which can ultimately lead to better maintenance and management of software systems.

3.2 Outline

In this section we will be explaining the process of making a tool capable of detecting Technical Debt (TD) with the application of graph learning. Together with my colleague, Mădălina Gavăt, we drew the study design of this research project. There are five main points that constitute the core of our study, which are listed here and will be discussed in depth in the next parts of this chapter:

- Data collection
  The content of the selected dataset was defining the whole process of creating our tool and for the findings of our study.
- Dataset preprocessing
  Preprocessing the chosen dataset was crucial for 'modelling' the data in a meaningful and useful manner for the pipeline.
- Use of graph representations for transforming code samples.
  On this point, after gathering good knowledge about the available data and finishing the preprocessing, we employed graph representation in order to transform the code samples.
- Graph learning models Discussing the model architectures that we chose for this study (GCN, GAT, GraphSAGE and Hybrid GAT).
- The creation of our tool
  This represents the creation of the tool that was used in finding viable answers for the posed research question.

In the structure of the methodology chapter, these steps correspond to different sections. First, the dataset selection is included in the data collection section. Next, the dataset preprocessing and the graph representations selection are correlated with the Data Analysis. Lastly, we will discuss the final step, our pipeline, in the instrumentation section.

With these steps listed above, there were only a number of things that needed to be set up before starting to work. These consist of the frameworks that we planned to use, the availability of the libraries/modules/resources that we needed, and, lastly, the ease of implementation.

Traditionally, Python[1] is considered one of the best and most useful scripting languages. The reasons why we chose this programming language is its use within machine learning, data science, automation, but also the extensive availability of modules and libraries that we could easily use. Some notable use cases in our project are the necessity of training models, handling vast amounts of data and the ease of integrating different external and self-made components.

[1]Python: https://www.python.org

Another important part is played by Django[2], a Python framework which is used in building reliable and easily-deployable backend web applications. In order to run and test the pipeline that we are building and find answers to the research questions that we posed, we need to deploy what we created. Also, as an extra point on our list, we would like to make the tool publicly available online, so everyone can access and use it, if the results are encouraging. More details about the deployment can be found in the next chapter.

Lastly, it is also worth mentioning that the entirety of the implementation took place on a personal laptop, where we also ran the small-scale testing of the functionality of the tool being constructed. Instead, Hábrók[3] (the HPC cluster of the RUG) was used for full training and evaluation of the model, since there was a great amount of data used. The tool for version control that we used is GitHub, and our repository is available there.

Now that we created the general picture of the methodology, we can move to the next parts of this chapter.

3.3 Data Collection

From the relevant literature, three articles stood out, each one containing the dataset that was used in conducting the research. This dataset selection represents the second step of the project.

The article by T. Amanatidis et al. [21] investigated through case study fifty open source projects in two programming languages, Java and Javascript. The criteria for selecting these projects included popularity (with more than 3,000 stars on GitHub), active maintenance, and a range of project sizes [21]. The datasets for the Java and JavaScript projects contain technical debt data points measured in minutes by three different tools: SonarQube, Squore, and CAST [21]. These data points are collected for each file within those included projects. Unfortunately, the amount of Technical Debt data in this dataset was small, and was not enough for training our own model and achieving meaningful results. While this study was proposing a framework for capturing diversity of TD, and applying three leading available TD tools [21], it appeared that there wasn't enough information about the projects that might help support our objectives.

Next, we also analysed the 'QScored' dataset, the work of T. Sharma et al. [18], which incorporates a lot of useful information about code quality aspects for more than 86 thousand projects, containing more than 1.1 billion lines of code. At the project level, metrics such as smell density and code duplication give an overview of the project's overall health and maintainability. These metrics help in identifying the proportion of code elements affected by code smells and the percentage of duplicated code within the projects, which

[2]Django: https://www.djangoproject.com
[3]Hábrók: https://wiki.hpc.rug.nl/habrok/

are critical indicators of technical debt. On the other hand, an issue would be the dataset's bias towards larger projects, as it excludes repositories with fewer than 1,000 lines of code [18]. This exclusion could overlook smaller, yet significant, projects that are relevant to technical debt studies. Lastly, what influenced our decision to not choose this dataset is the preprocessing of the 'QScored' dataset [18] for graph learning, due to the volume and complexity of the data. The dataset includes detailed metrics for a large number of repositories, making data handling and preprocessing computationally intensive and time-consuming. Normalising and standardising these metrics across different repositories and languages is another challenge, as it is essential to ensure consistency throughout this research project. In short,the dataset does not contain clearly identifiable TD items, but derived metrics, which makes it difficult to work with. Additionally, repositories may have incomplete data for certain metrics due to differences in codebase structure or limitations of the analysis tools used [18].

Lastly, we analysed the publication of V. Lenarduzzi et al.[15]. 'The Technical Debt Dataset' was introduced as a meticulously curated set of measurement data from 33 Java projects from the Apache Software Foundation. This dataset includes an analysis of all commits from specified time frames using SonarQube to gather Technical Debt information and Ptidej to identify code smells[15]. The authors apply the SZZ algorithm to determine fault-inducing and fault-fixing commits [15]. The analysis encompasses 78,000 commits across the selected projects, uncovering 1.8 million SonarQube issues, 62,000 code smells, 28,000 faults, and 57,000 refactorings [15]. What proved to be a great advantage towards our research project was the fact that the dataset is accessible through an SQLite database, facilitating efficient data queries and scripting for finding information. It is worth mentioning that this dataset is also very light, since it doesn't include any source code, but only information about the projects in questions. This appeared to us as a perfect starting point for our dataset preprocessing, which is the third step of the project, as per the plan presented in the introduction of the Methodology chapter.

Important information can be drawn from the 'SZZ_FAULT_INDUCING_COMMITS' table (Figure 3.1), since it contains valuable details about the fault inducing and fault fixing commits. This represented a solid starting point in the preprocessing of the data (which will be discussed more in-depth in the next section). Because the goal was to apply graph learning, having both the TD containing sample and its solving pair, meant that we could already collect both negative and positive samples for the training by using the information from there. The dataset by itself only contains TD items, or positive samples for the large variety of SonarQube code smells analysed. For gathering enough negative samples for a selected code smell, we used negative samples that we collected ourselves and positive samples from other code smells.

Also, each table contains the key 'projectID' (Figure 3.1), which can easily allow us to join data together from multiple tables, when querying. Therefore,

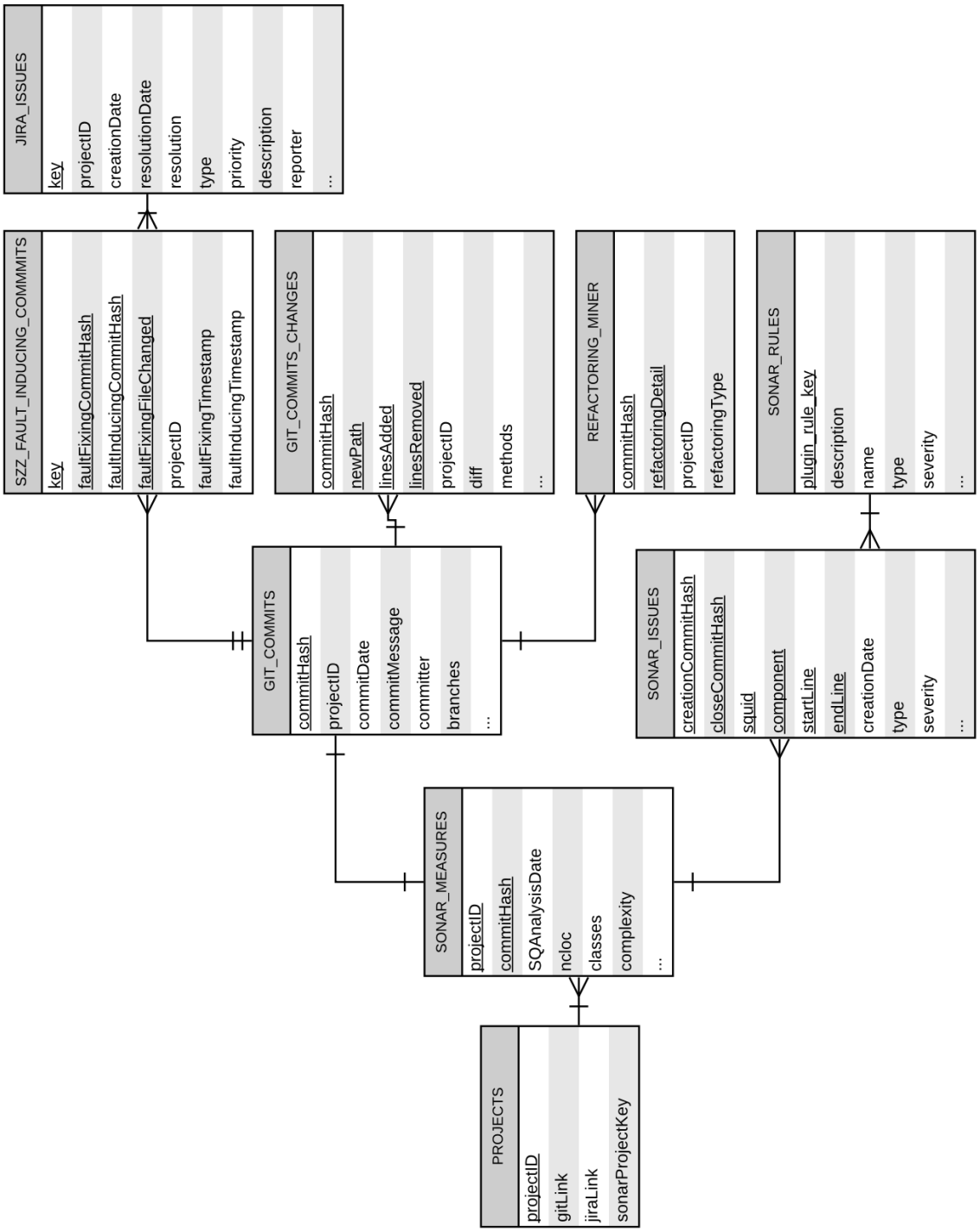

Figure 3.1: Entity Relationship Diagram of the 'Technical Debt Dataset'. [15]

Project Name	Git	Jira		Refactoring Miner		Sonar Qube		Ptidej		Analysis Timeframe		
	#Commits	#Issues	#Faults	#Commits	#Refactorings	#Commits	#Introduced TD items	#Commits	#Introduced Code Smells [10]	Git / Refactoring Miner	Jira	SonarQube / Ptidej
Accumulo	10,122	4,744	2,250	42	368	2,641	1,377,049	2,641	9,249	10/11 - 7/19	10/11 - 7/19	10/11 - 10/13
Ambari	24,579	25,041	17,722	58	342	13,397	40,698	13,397	7,738	08/11 - 7/19	09/11 - 7/19	09/11 - 06/15
Atlas	2,794	3,284	1,990	793	5,029	2,336	34,997	2,336	3,535	11/14 - 7/19	05/15 - 7/19	12/14 - 06/18
Aurora	4,065	1,969	628	562	3,080	4,012	7,405	4,012	1,983	04/10 - 7/19	10/13 - 7/19	04/10 - 06/18
Batik	3,491	1,265	1,160	429	3,117	2,097	31,113	2,097	2,362	10/00 - 7/19	01/01 - 7/19	10/00 - 06/06
Beam	22,332	2,361	1,723	2,113	10,105	2,865	74,434	2,865	8,458	12/14 - 7/19	02/16 - 7/19	12/14 - 07/16
Cocoon	13,160	521	327	984	3,447	10,210	47,994	10,210	6,513	02/03 - 7/19	01/01 - 7/19	03/03 - 02/07
Commons BCEL	1,451	7,750	3,218	85	643	1,324	7,471	1,324	562	10/01 - 7/19	05/02 - 7/19	10/01 - 04/18
Commons BeanUtils	1,234	319	242	75	203	1,192	4,674	1,192	424	03/01 - 7/19	10/01 - 7/19	03/01 - 06/18
Commons CLI	921	669	346	51	145	896	30,300	896	3,779	06/02 - 7/19	06/02 - 7/19	06/02 - 02/18
Commons Codec	1,825	295	182	148	474	1,726	1,831	1,726	166	04/03 - 7/19	04/02 - 7/19	04/03 - 05/18
Commons Collections	3,135	256	135	382	2,663	2,982	9,566	2,982	1,175	04/01 - 7/19	05/01 - 7/19	04/01 - 09/18
Commons Configuration	3,077	108	73	488	1,585	2,895	4,334	2,895	869	12/03 - 7/19	01/03 - 7/19	12/03 - 05/18
Commons Daemon	1,087	297	190	7	11	980	371	980	13	09/03 - 7/19	08/03 - 7/19	09/03 - 08/11
Commons DBCP	2,010	598	284	174	600	1,861	5,390	1,861	265	04/01 - 7/19	02/02 - 7/19	04/01 - 06/18
Commons DbUtils	662	291	159	37	152	645	545	645	56	11/03 - 7/19	11/03 - 7/19	11/03 - 12/17
Commons Digester	2,145	305	149	136	405	2,145	6,336	2,145	926	05/01 - 7/19	02/01 - 7/19	05/01 - 08/17
Commons Exec	627	747	444	36	77	617	655	617	67	07/05 - 7/19	08/05 - 7/19	08/05 - 01/16
Commons FileUpload	962	402	282	21	101	922	666	922	62	03/02 - 7/19	10/02 - 7/19	03/02 - 10/17
Commons IO	2,180	540	368	140	494	2,118	5,381	2,118	381	01/02 - 7/19	04/03 - 7/19	01/02 - 06/18
Commons Jelly	1,939	142	56	171	455	1,939	6,189	1,939	764	02/02 - 7/19	07/02 - 7/19	02/02 - 09/17
Commons JEXL	1,655	191	119	344	1,896	1,551	33,694	1,551	1,107	04/02 - 7/19	06/03 - 7/19	04/02 - 05/18
Commons JXPath	598	455	265	66	396	597	4,550	597	355	08/01 - 7/19	05/02 - 7/19	08/01 - 11/15
Commons Net	2,117	719	438	91	325	2,088	35,565	2,088	3,738	04/02 - 7/19	02/02 - 7/19	04/02 - 08/17
Commons OGNL	615	6,056	3,415	49	460	608	4,483	608	362	05/11 - 7/19	11/05 - 7/19	05/11 - 09/13
Commons Validator	1,342	932	397	50	168	1,339	1,720	1,339	252	01/02 - 7/19	02/02 - 7/19	01/02 - 04/18
Commons VFS	2,288	133	84	293	1,015	2,067	3,111	2,067	549	07/02 - 7/19	03/04 - 7/19	07/02 - 05/18
Felix	15,427	194	147	1,585	6,610	596	10,370	596	772	07/05 - 7/19	07/05 - 7/19	08/05 - 10/06
HttpComponents Client	3,009	663	463	483	2,624	2,867	8,998	2,867	1,436	12/05 - 7/19	10/01 - 7/19	12/05 - 06/18
HttpComponents Core	3,288	258	188	572	3,435	1,941	7,531	1,941	1,255	02/05 - 7/19	07/02 - 7/19	02/05 - 08/17
MINA SSHD	1,787	566	285	512	3,993	1,370	7,724	1,370	1,002	12/08 - 7/19	12/08 - 7/19	12/08 - 06/18
Santuario Java	2,824	1,932	1,302	224	1,035	2,697	19,807	2,697	1,854	09/01 - 7/19	08/13 - 7/19	10/01 - 06/18
ZooKeeper	1,940	3,424	1,859	522	2,077	411	5,265	411	391	11/07 - 7/19	06/08 - 7/19	05/08 - 06/18
Total	134,812	67,427	40,890	11,723	57,530	77,932	1,840,217	77,932	62,420			

Figure 3.2: Description of the selected projects from the 'Technical Debt Dataset'. [15]

this gave us the possibility of creating individual tables containing project information, such as project name, GitHub link, fault inducing commit hash, fault fixing commit hash, rule and component file (Figure 3.1).

If we take a look at the analysis timeframe from the 'Technical Debt Dataset' (Figure 3.2), we can see that this data collection contains information about the aforementioned projects on a long interval of time. If we also take a look at the number of commits (Figure 3.2), we can see that there is a great number of them, and some projects even have tens of thousands of commits. This ensures that the number of samples necessary for training is obtainable in large numbers and we should have variety as well.

3.4 Data Analysis

In this section we will be analysing the data that we have, and we will be discussing the selection of SonarQube rules available in the current dataset. Afterwards, we will be delving into the queries that were performed, and, lastly, we will be talking about the graph representations that were chosen for this study.

3.4.1 SonarQube rules

In SonarQube, for Java projects, there is a total of 698 rules used in analysing code smells and vulnerability problems, according to the Sonarsource website[4]. Due to concerns about the amount of data that would be available in the dataset for training, the first step that we took was to make a selection of the top 100 most frequently occurring rules among the hundreds of thousands of entries. This can be

```
SELECT
    RULE,
    TYPE,
    COUNT(RULE) AS RULE_COUNT
FROM
    SONAR_ISSUES
GROUP BY
    RULE,
    TYPE
ORDER BY
    RULE_COUNT DESC LIMIT 100;
```

Listing 3.1: Top 100 SonarQube Rules: Query performed on the Technical Debt Dataset

The results of Query 3.1 can be found in the Additional Data, at the end of this thesis (Table: A.1). The next step, after we obtained these entries, was to try and narrow the selection down as much as we could. In order to do that we needed to find a suitable criteria on which we could categorise the SonarQube rules.

This lead us to decide that we would be using 'single-file' or 'file-based' rules. This entails that we would be selecting the SonarQube rules that would allow us to use a file directly as sample, instead of snippets of code from the files or entire directories. By doing so, we could ensure a lower complexity in the preprocessing, since the Technical Debt Dataset[15] makes references to specific files from the repositories. Also, it is worth mentioning that it is easier to tackle 'single-file' SonarQube rules, since the TD that could be detected doesn't span multiple parts of the whole project, but it is only reduced to that specific file containing the code in question. That's how we reduced the list to only 15 rules. The full table with each rule and its individual description can be found in the Additional Data section, at the end of this thesis (Table: A.2). In the following table you can see only the names of those 15 selected SonarQube rules.

Rule Name	Final Selection
Cyclomatic Complexity	✓
Class Data Should Be Private	

[4]Sonarsource: https://rules.sonarsource.com/java/

Rule Name	Final Selection
Duplicated Blocks	✓
Long Method	✓
Complex Class	✓
Unused Private Method	
Base Class Should Not Use Derived Class Functions	
Method With Boolean Parameter	
Redundant Throws Declaration	
Variable Declaration Distance	
Modifiers Order	
Switch Cases Without Break	
Method Should Not Have Too Many Parameters	✓
Class Variable Visibility Check	
Lazy Class	✓

Table 3.1: 15 Selected Rules from the SonarQube Analysis from the Technical Debt Dataset

As you can notice in Table 3.1, six of the fifteen selected rules have a check-mark in the 'Final Selection' column. The reasons why those were chosen can be found in the Table 3.2, along with the possible graph representations that could help in identifying the TD related to the rule.

Rule Name	Representation	Information
Cyclomatic Complexity[1]	Control Flow Graph (CFG)	High cyclomatic complexity is indicated by numerous branching and merging points in the CFG, making functions error-prone and hard to maintain. Simplification reduces technical debt.
Duplicated Blocks[2]	Subgraph Isomorphism	Duplicated code segments appear as isomorphic subgraphs within a graph representation, indicating redundant code that increases maintenance effort and error risk. Removing duplicates reduces technical debt.

Rule Name	Representation	Information
Long Method[3]	Abstract Syntax Tree (AST)	Long methods manifest as nodes with high complexity and depth in the AST part of the CPG, indicating the need for simplification into smaller, more manageable functions to improve maintainability and reduce technical debt.
Complex Class[4]	Class Dependency Graph	Complex classes are shown by dense connections and interactions in the Class Dependency Graph, suggesting the need for simplification into smaller classes to enhance maintainability and scalability, reducing technical debt.
Method Should Not Have Too Many Parameters[5]	Abstract Syntax Tree (AST)	Methods with many parameters are identifiable by nodes with numerous parameter nodes in the AST part of the CPG, indicating the need for refactoring to reduce parameter counts and improve readability, thereby reducing technical debt.
Lazy Class[6]	Class Dependency Graph	Large, monolithic classes appear as extensive nodes with numerous edges in the Class Dependency Graph, suggesting decomposition into smaller, focused classes to improve maintainability and reduce technical debt.

Table 3.2: 6 Selected Rules from the SonarQube Analysis

[1] SonarQube: https://docs.sonarsource.com/sonarqube/latest/user-guide/metric-definitions/#complexity
[2] SonarQube: https://docs.sonarsource.com/sonarqube/latest/user-guide/metric-definitions/#duplications
[3] SonarQube: https://sonarsource.atlassian.net/browse/RSPEC-138
[4] [6] SonarQube: https://github.com/davidetaibi/sonarqube-anti-patterns-co blob/master/README.md
[5] SonarQube: https://rules.sonarsource.com/java/RSPEC-107/?search=long%20parameter%20list

This is the selection of SonarQube rules that we focused on. From Table 3.2,

we can notice that there are already 'leads' on what graph representations we could use in identifying those code smells, along with a resolution of these issues. Graph representations will be covered more in depth in Subsection 3.4.3. It is also worth specifying that SonarQube changes the names and ids of the some rules from version to version, so that is a reason why we could not find information in the same place about all of the selected rules. The links providing information differ, but they refer to great resources used in SonarQube analysis studies or projects. It was especially difficult to align the rules from the Technical Debt Dataset, since their names did not correspond with the new names used in newer versions of SonarQube.

Currently, for the purpose of this study and the creation of our tool, we decided to cover only two code smells: 'Long parameter list' and 'Long method'. The reasons behind choosing those two are that they have great impact on software maintenance and the 'accumulation' of technical debt. Long methods usually obscure the logic and flow of the program, making it difficult for developers to understand and modify the code easily. This can lead to increased error rates and longer debugging times, which results in escalating maintenance costs and efforts. Additionally, methods with long parameter lists can complicate the interface and usage of those methods, which in turn affects the maintainability of software projects and leads to a complicated maintenance process.

On the other hand, identifying lazy classes can help optimize the codebase, the impact on immediate readability and maintainability is smaller than addressing long methods and parameter lists. Breaking down long methods naturally simplifies complex classes, achieving similar goals indirectly. Also, removing duplicated blocks reduces redundancy, but does not directly enhance the understanding of individual code pieces as effectively as focusing on method and parameter list length. Lastly, reducing cyclomatic complexity often overlaps with the benefits of breaking down long methods.

3.4.2 Dataset preprocessing

In the previous section we covered the selection of the final SonarQube rules, which we based our study on. Now, we will be describing the preprocessing of the available dataset. First, let's narrow the entries only to the scope of the six selected rules. We did that through the following query (Listing 3.2).

```
CREATE TABLE SELECTED_RULES AS
SELECT PROJECT_ID, COMPONENT, RULE
FROM SONAR_ISSUES
WHERE RULE="squid:MethodCyclomaticComplexity"
   OR RULE="common-java:DuplicatedBlocks"
   OR RULE="code_smells:long_method"
   OR RULE="code_smells:long_parameter_list"
   OR RULE="code_smells:complex_class"
   OR RULE="code_smells:lazy_class"
ORDER BY RULE;
```

Listing 3.2: Query for creating a table containing all the data for the selected SonarQube rules

Now, in this SELECTED_RULES table, we have all the entries from the SONAR_ISSUES table for the six selected rules. If we analyise Figure 3.1 again, we can see that we can join together information from multiple tables, mainly through the key represented by the PROJECT_ID.

Thus, the new tables that we are trying to build now for each one of the selected rules will have the structure presented in Table 3.3. We need the GitHub link so that we can download the repository. Next, we will be using the fault inducing and fault fixing commit hashes to obtain negative and, respectively, positive TD samples. Lastly, we need to know which file in question is the entry about, so we need the 'COMPONENT' as well. These details should be enough for us to gather a significant amount of positive and negative samples, that should suffice for the training of the models that we will be using.

Column Name	**Table of Origin**
PROJECT_ID	PROJECTS
GIT_LINK	PROJECTS
FAULT_INDUCING_COMMIT_HASH	SZZ_FAULT_INDUCING_COMMITS
FAULT_FIXING_COMMIT_HASH	SZZ_FAULT_INDUCING_COMMITS
RULE	selected_rules
COMPONENT	selected_rules

Table 3.3: Columns that contains useful information for our study from the Technical Debt Dataset 3.1

Based on this table, we created queries for each one of the selected rules. The following code snippet contains the query for the 'long_parameter_list' rule:

```
CREATE TABLE manyParameters AS
SELECT PROJECTS.PROJECT_ID, PROJECTS.GIT_LINK,
       SZZ_FAULT_INDUCING_COMMITS.FAULT_INDUCING_COMMIT_HASH,
           SZZ_FAULT_INDUCING_COMMITS.
          FAULT_FIXING_COMMIT_HASH,
       selected_rules.RULE, selected_rules.COMPONENT
FROM PROJECTS
INNER JOIN selected_rules ON PROJECTS.PROJECT_ID =
   selected_rules.PROJECT_ID
INNER JOIN SZZ_FAULT_INDUCING_COMMITS ON PROJECTS.PROJECT_ID
   = SZZ_FAULT_INDUCING_COMMITS.PROJECT_ID
WHERE selected_rules.RULE = "code_smells:long_parameter_list"
LIMIT 5000;
```

Listing 3.3: Query for the 'long_parameter_list' rule

We are creating new tables where we join information from the columns mentioned in Table 3.3, such as in the example from Listing 3.3. The full query, also containing the splits for training, validation and testing, can be found in the Appendix in Additional Data (Listing A.2). The results of this query can also be found in the Appendix, in Figure A.1.

3.4.3 Graph Representations

With the dataset sample extraction complete, the next step is discussing the graph representations that we employed. This is one of the most important parts of this project, since it influences the transformation of the samples for the training of the models that we will be using.

Earlier, in the SonarQube rules Section 3.4.1, we introduced for each one of the rules the graph representations which can be used (Table 3.2). This section delves into the choices that we made for our study. First, we will list all of those graph representations from the table of rules with observations (Table 3.2), and we will provide more details about their use and properties.

For the purpose of our study, we chose to use Code Property Graphs (CPGs) and investigate how they can impact the effectiveness of the specified graph learning models in the scope of techincal debt detection. After looking at Table 3.4, we can see what the requirements for detecting the SonarQube code smells are. CPGs, fortunately, have a series of features which can help meet those requirements.

In creating our instrumentation, we decided to use Joern[5]. This is a very powerful tool for transforming code into graph representations, including Code Property Graphs (CPGs). It provides coverage for multiple programming languages, amongst which the maturity level of the tool for Java projects is one of the highest. Joern features robust parsing characteristics and has extensive documentation about generating CPGs and querying information. Another advantage of using joern is that it can be easily integrated in other projects or tools.

For the 'Long parameter list' code smell, after performing the transformations using Joern, we had in total a number of 1619 samples (50% positive samples, 50% negative samples). These samples are further allowed us to split the dataset as follows: 80% (1296) samples for training, 10% (162) for validation, and 10% (161) for testing. It's worth mentioning that all of these proportions maintain 50% positives and 50% negatives.

For the 'Long method' code smell, after the Joern transformations, we had a total of 1108 samples, with half comprising of positive samples and the other half of negative samples. In a similar manner as the previously mentioned code smell, the training-validation-testing split is still the same, with 80%-10%-10% (or 887-110-111) samples. Again, we kept the proportion of

[5]Joern.io: https://joern.io

positives and negatives equal.

Representation	**Observations**
Control Flow Graph (CFG)	• Represents the flow of control in a program. • Indicates number of linearly independent paths. • High complexity suggests error-prone and hard to maintain code.
Subgraph Isomorphism in Graph Representations	• Identifies isomorphic subgraphs within the code. • Useful for detecting duplicated code segments. Duplicates can also include pieces of code which are very similar, not only identical ones. • Removing duplicates can reduce maintenance effort and error risk.
Abstract Syntax Tree (AST)	• Represents the hierarchical structure of source code. • Nodes with high complexity and depth indicate long methods. • Nodes with numerous parameter nodes suggest methods with too many parameters.
Class Dependency Graph	• Shows dependencies between classes in a program. • Dense connections and interactions indicate complex classes. • Large nodes with numerous edges suggest monolithic classes.

Table 3.4: Graph representations and observations

3.4.4 Model Architectures Employed

For these analysis tasks, we employed a number of eight different models that we created. Four are explained in this study, and another number of four are covered by my colleague in another thesis. In our research project, we tried to cover a large variety of Graph Neural Networks (GNNs) types, in order to check their effectiveness in technical debt detection. In this thesis, we cover one node sampling model (GraphSAGE), a spectral convolution model (GCN), a spatial attentional model (GAT), and a hybrid model combining GAT and GCN, providing architecture diversity. Those are as follows:

- **Model 1: GraphSAGECustom**
  - Uses `SAGEConv` layers.
  - Input channels: 768.
  - Hidden channels: 256.
  - Output channels: 2.
  - Dropout rate: 0.3.
  - Architecture:
    - 2 SAGEConv layers (256 hidden channels), each followed by ReLU activation and dropout.
    - Global mean pooling.
    - Final fully connected layer.

  The GraphSAGE model that we created employs two layers of SAGEConv. We chose this architecture because SAGEConv is effective at capturing complex patterns in graph data through its aggregation methods. Each layer is followed by ReLU activation and dropout, which helps in preventing overfitting and enhances the model's generalization capabilities. The use of global mean pooling aggregates the node features into a single graph-level representation before passing it to a fully connected layer for final classification. The selection of 256 hidden channels balances model capacity and computational efficiency, ensuring sufficient complexity to learn meaningful representations without overburdening the training process. The number of 256 was reached after lowering the amount of hidden channels. Initially, there were too many hidden channels, and this lead to overfitting. A dropout rate of 0.3 was chosen to prevent overfitting and maintain model performance.

- **Model 2: HybridGAT**
  - Uses `GATConv` and `GCNConv` layers.
  - Input channels: 768.

  - Hidden channels: 16.
  - Output channels: 2.
  - Dropout rate: 0.5.
  - Architecture:
    * First layer: GATConv with 8 heads (16 hidden channels), followed by ReLU activation and dropout.
    * Second layer: GCNConv (16×8 hidden channels), followed by ReLU activation and dropout.
    * Global mean pooling.
    * Final fully connected layer.

  The HybridGAT model that we built combines both GATConv and GCNConv layers in order to use the strengths of both graph attention and convolutional mechanisms. The first layer uses GATConv with multiple heads to focus on the importance of different nodes from the CPGs, capturing node relationships dynamically. The second layer employs GCNConv to aggregate the node features, providing a good feature extraction process. We selected 16 hidden channels to create a lightweight and effective model, ensuring quicker training and reduced computational cost. The dropout rate of 0.5 was chosen to combat overfitting. This can also improve generalisation on diverse graph structures within CPGs.

- **Model 3: GCN**
  - Uses `GCNConv` layers.
  - Input channels: 768.
  - Hidden channels: 128.
  - Output channels: 2.
  - Dropout rate: 0.5.
  - Architecture:
    * 2 GCNConv layers (128 hidden channels each), each followed by ReLU activation and dropout.
    * Global mean pooling.
    * Final fully connected layer.

  This GCN model is relatively simple, featuring only two GCNConv layers with ReLU activation and dropout. We chose this architecture be-

cause GCNConv layers are effective for learning from graph-structured data, providing a balance between simplicity and performance. The use of 128 hidden channels was selected to maintain model capacity while ensuring computational efficiency, particularly useful for scenarios where the graph structures are not overly complex. The dropout rate of 0.5 was chosen to prevent overfitting, especially important given the moderate hidden layer size, allowing the model to generalize well to unseen data.

- **Model 4: GAT**
  - Uses `GATConv` layers.
  - Input channels: 768.
  - Hidden channels: 32.
  - Output channels: 2.
  - Dropout rate: 0.5.
  - Architecture:
    - 2 GATConv layers with 8 heads, each followed by ReLU activation and dropout.
    - Global mean pooling.
    - Final fully connected layer.

  The GAT model that we created employs two GATConv layers with multiple heads, focusing on using the graph attention mechanism to learn the importance of different nodes and their features. Each layer is followed by ReLU activation and dropout, in order to enhance learning and prevent the overfitting of the data. The attention mechanism in GAT allows the model to weigh the contributions of different nodes differently, which can be particularly useful in graphs with varying node importance. Global mean pooling aggregates the learnt node features into a single vector, which is then classified by a fully connected layer. We chose 32 hidden channels to balance the model complexity and performance, ensuring it can learn effectively without becoming too computationally expensive. The dropout rate of 0.5 was selected to mitigate the risk of overfitting, crucial for maintaining the model's ability to generalize from the training data.

The choices that we made for each model were considered based on their effectiveness and relevance to the problem of technical debt detection on graph representations. The SAGEConv layers that we used in the GraphSAGE model were selected for their ability to capture complex patterns through efficient node aggregation, while the GAT and GCN layers that we used in the other models employed the strengths of both attention and convolu-

tional mechanisms. These specific model architecture configurations, such as the number of layers, hidden channels, and dropout rates were determined mainly through empirical tuning. We optimised these configurations in order to balance model complexity and performance, while addressing issues such as overfitting. Initially, we started with a higher amount of layers and hidden channels, but that lead to models being too complex.

Additionally, the batch size and learning rates have also been empirically tuned. Generally, smaller batch sizes lead to better results and greater generalisation[6], and that is the reason why we settled for a batch size of 16 for the models that we employed (as it can be seen in Chapter 4). In terms of learning rates, for each one of the model architectures, they were determined through experimentation. We opted for a fixed learning rate, since it is ideal for simple or baseline models, and works well with datasets that are not very large[7]. We also considered the use of a decaying learning rate that could address the possible fluctuations, but it required some tuning of the decaying rate and predefining steps.

3.5 Instrumentation

For effectively running the data gathering, data preprocessing and model training, we came to the conclusion that we needed to build a pipeline for streamlining the process. Thus, we created the tool called 'Debtective'. In this section we will be focusing on the backend of the tool, which represents the work that was done in scope of this project.

We will be going through the different groups of components that we have, starting from the repository gathering based on the information from the dataset. Next, we will be discussing the transformations to graph representations that were performed, and their labeling process. In the end, we will be covering the models that we used, and how those were built, before we jump in the final discussion about model training.

The evaluation of the training will be covered in the results category, since those results represent the outcomes of this research project. The general software architecture of the 'Debtective' tool can be found in Figure 3.3.

3.5.1 Repository gathering

The repository gathering part of our tool represents the first group of components, which in the architecture of our tool (Figure 3.3) is called the 'Sampler'. Here, we get the necessary information from the 'Technical Debt Dataset' [15], and we are passing this to the 'Grabber' component. This 'Grabber' uses the 'GitHub Grabber' component that we used in order to download the repositories and revert back and forth using commit hashes, and in the

[6]Medium: https://medium.com/geekculture/why-small-batch-sizes-lead-to-greater-genera
[7]Medium: https://medium.com/thedeephub/learning-rate-and-its-strategies-in-neural-ne

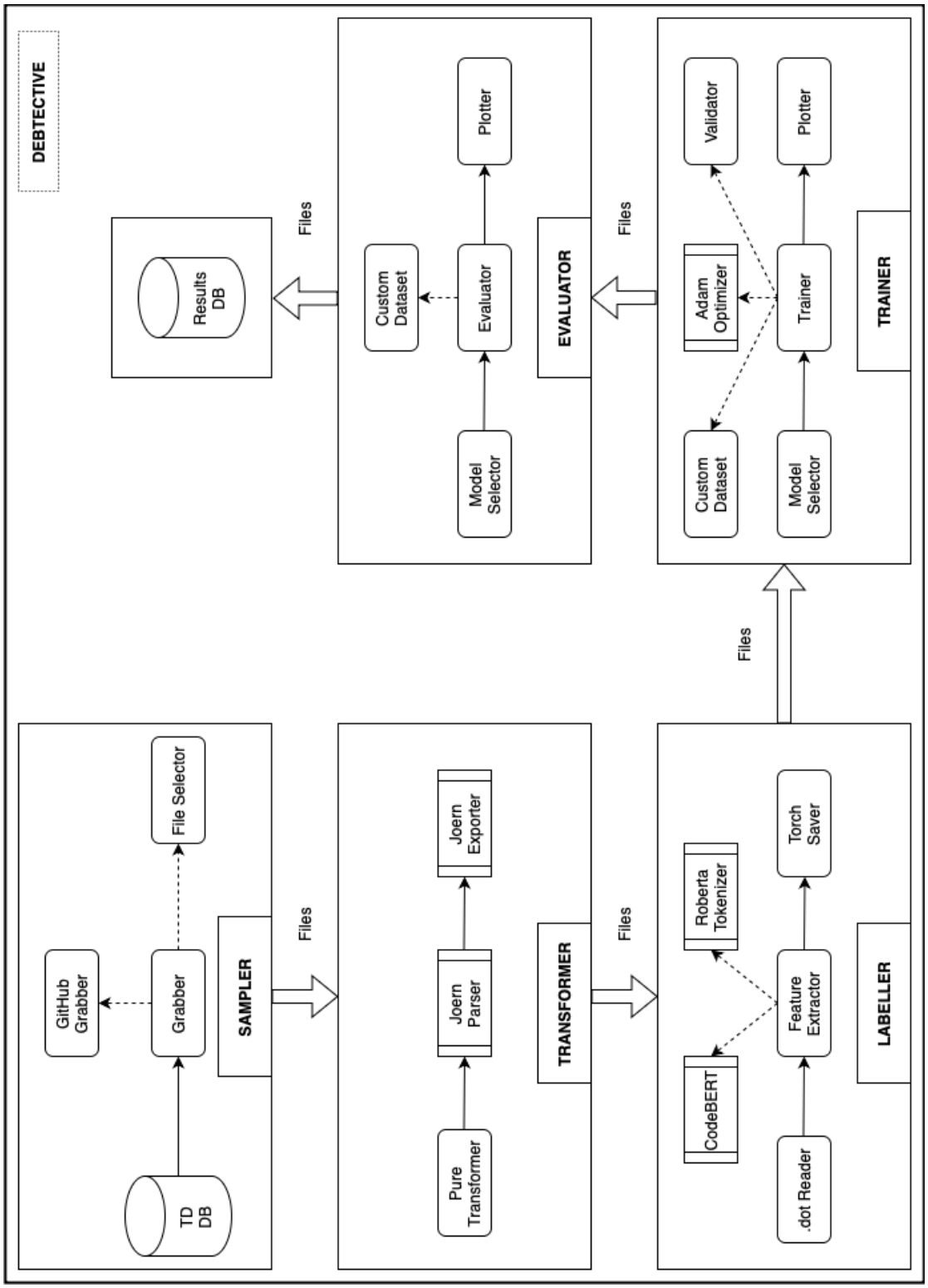

Figure 3.3: Software Architecture of the Debtective Tool

end it stores the repositories in a directory. Next, our 'Grabber' uses the 'File Selector' component to select the files that are marked in the 'Technical Debt Dataset' as having SonarQube code smells for a specific given rule.

After the whole operation is done, the selected files from the fault-inducing commits will be placed in the positive samples directory, and the ones selected from the fault-fixing commits will be placed in the negative samples directory. Before we move to the next step, we delete the repositories from the repository directory, so that we don't retain any redundant (non-TD containing) files. For the sampling of the data, we used the fields mentioned in Section 3.4.2, in Table 3.3.

3.5.2 Rules covered by our tool

Long Parameter List

This SonarQube code smell takes into account the number of parameters given to a method. Generally, methods with a long parameter list are difficult to use because maintainers must figure out the role of each parameter and keep track of their position, as stated in the SonarSource rules documentation[8]. In our case, we have to analyse the amount of parameter nodes in each one of our transformed samples, and allow the model to predict if a sample has too many parameters or not.

Examples:

```
public void createUser(String firstName, String lastName,
    String email) {
     User user = new User();
     user.setFirstName(firstName);
     user.setLastName(lastName);
     user.setEmail(email);
     // Save user to database
     userRepository.save(user);
}
```

Listing 3.4: Example of negative sample for the 'Long parameter list' code smell.

```
public void createUser(String firstName, String lastName,
    String email, String password, String address, String
    phoneNumber, Date birthDate) {
     User user = new User();
     user.setFirstName(firstName);
     user.setLastName(lastName);
     user.setEmail(email);
     user.setPassword(password);
     user.setAddress(address);
     user.setPhoneNumber(phoneNumber);
```

[8]SonarSource: https://rules.sonarsource.com/java/RSPEC-107/

```
    user.setBirthDate(birthDate);
    // Save user to database
    userRepository.save(user);
}
```

Listing 3.5: Example of positive sample for the 'Long parameter list' code smell.

Negative samples

Negative samples represent the code samples in which all methods have at most 4 parameters (Listing 3.4).

Positive samples

Positive samples represent the code samples that have at least a method with more than 5 parameters (Listing 3.5).

Long Method

This SonarQube code smell accounts for the size of the methods from the provided code samples. According to SonarSource, a method that grows too large tends to aggregate too many responsibilities[9]. Such methods inevitably become harder to understand and therefore harder to maintain in the long run[10]. In this case, when running the analysis we have to check the amount of nodes in the Code Property Graphs (CPGs) of our transformed samples for each method individually, and allow our model to predict if a method is too long or not.

Examples:

```
public void processOrder(Order order) {
    validateOrder(order);
    double total = calculateTotal(order);
    total = applyDiscounts(order, total);
    finalizeOrder(order, total);
}
```

Listing 3.6: Example of negative TD sample for the 'Long method' code smell.

```
public void processOrder(Order order) {
    // Validate order
    if (order == null || !order.isValid()) {
        throw new IllegalArgumentException("Invalid order");
    }

    // Calculate total
```

[9]SonarSource: https://rules.sonarsource.com/java/RSPEC-138/
[10]SonarSource: https://rules.sonarsource.com/java/RSPEC-138/

```
    double total = 0;
    for (OrderItem item : order.getItems()) {
        total += item.getPrice() * item.getQuantity();
    }

    /* 20 more lines of code here ... */

    // Apply discounts
    if (order.hasDiscount()) {
        total -= order.getDiscount();
    }

    // Finalize order
    order.setTotal(total);
    order.setStatus(OrderStatus.PROCESSED);
    orderRepository.save(order);
}
```

Listing 3.7: Example of positive TD sample for the 'Long method' code smell.

Negative samples

Negative samples represent the code samples in which all methods have less than 30 lines of code (Listing 3.6).

Positive samples

Positive samples represent the code samples in which there is at least a method that is longer than 30 lines of code (Listing 3.7).

3.5.3 Sample transformations

The next group of components that transforms the code samples into graph representations is called the 'Transformer'. As it can be seen in Figure 3.3, we are using three components to perform this operation. The 'Pure Transformer' that we wrote takes all of the positive and negative samples from the aforementioned directories, and passes the files to the 'Joern Parser'. Lastly, the graph representations of the code samples are exported by 'Joern Exporter' into .dot format and saved in two separate directories, one for the positive samples and one for the negative samples.

If we take a look into the process of transformation, we can see that Joern[11] performs the following operations:

The 'Joern Parser'[12] component converts Java source code into Code Property Graphs (CPGs) through a multi-step process. It begins with lexical and syntax analysis in order to tokenize the code and build a Parse Tree. The

[11] Joern.io: https://joern.io
[12] Joern.io documentation: https://docs.joern.io

Parse Tree is then then transformed into an Abstract Syntax Tree (AST). Next, in the semantic analysis step, type resolution and symbol table construction are performed, so that we can capture the binding and the scope of the identifiers. From here, this information is used to create an intermediate representation (IR) of the code, which serves as the foundation for generating a Control Flow Graph (CFG) that maps out the possible execution paths through the code file given.

After that, a Program Dependence Graph (PDG) is constructed by incorporating data flow information into the already-created CFG, capturing both control and data dependencies. The ASTs, CFGs, and PDGs are then integrated to form the Code Property Graph (CPG), which is a comprehensive representation that encapsulates the syntactic structure, control flow, and data dependencies of the given code samples.

Lastly, the in-memory CPG is serialized for storage by the 'Joern Exporter'[13], and, in the end, it is converted to a selected format. In our study, we decided to use the .dot format, since it offers the advantage of visualizing complex graph structures easily through tools like Graphviz, which also aids in understanding and debugging the possible problems. It also provides a straightforward, text-based format that is easily-readable and changeable.

An example of a resulting Code Property Graph in .dot format can be seen in Listing 3.8. It represents a negative (non-TD containing) sample transformed from a method called 'getCtrlKey'. This sample contains the information necessary for later labelling and analysis in the generated nodes and edges.

[13]Joern.io documentation: https://docs.joern.io

```
digraph "getCtrlKey" {
"2961" [label = <(METHOD,getCtrlKey)<SUB>75</SUB>> ]
"2962" [label = <(PARAM,this)<SUB>75</SUB>> ]
"2963" [label = <(BLOCK,&lt;empty&gt;,&lt;empty&gt;)<SUB>75</
   SUB>> ]
"2964" [label = <(RETURN,return ctrlKey;,return ctrlKey;)<SUB
   >76</SUB>> ]
"2965" [label = <(&lt;operator&gt;.fieldAccess,this.ctrlKey)<
   SUB>76</SUB>> ]
"2966" [label = <(IDENTIFIER,this,return ctrlKey;)> ]
"2967" [label = <(FIELD_IDENTIFIER,ctrlKey,ctrlKey)<SUB>76</
   SUB>> ]
"2968" [label = <(MODIFIER,PUBLIC)> ]
"2970" [label = <(METHOD_RETURN,boolean)<SUB>75</SUB>> ]
  "2961" -> "2962"  [ label = "AST:␣"]
  "2961" -> "2963"  [ label = "AST:␣"]
  "2963" -> "2964"  [ label = "AST:␣"]
  "2965" -> "2964"  [ label = "CFG:␣"]
  "2967" -> "2965"  [ label = "CFG:␣"]
  "2961" -> "2967"  [ label = "CFG:␣"]
  "2964" -> "2970"  [ label = "DDG:␣&lt;RET&gt;"]
  "2962" -> "2970"  [ label = "DDG:␣this"]
  "2961" -> "2962"  [ label = "DDG:␣"]
}
```

Listing 3.8: Example CPG in .dot format.

Each node in Listing 3.8 represents the different elements of the code. This includes methods, parameters, blocks, return statements, field access, identifiers, modifiers, etc. The label inside each node provides detailed information about the element. In this example, we have

$$"2961"[label =< (METHOD, getCtrlKey) < SUB > 75 < /SUB >>]$$

, which indicates a method named getCtrlKey located at line 75. In a similar way, nodes like

$"2964"[label =< (RETURN, returnctrlKey;, returnctrlKey;) < SUB > 76 < /SUB >>]$

denote return statements, while

$"2965"[label =< (< operator > .fieldAccess, this.ctrlKey) < SUB > 76 < /SUB >>]$

represents field access operations in the original code sample.

The edges between nodes define relationships such as Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Dependence Graph (DDG). These relationships help in understanding the structural and data dependencies within the code. For example,

$$"2961" - > "2962"[label = "AST : "]$$

shows an AST relationship, which suggests that the node "2962" is a child of "2961" in the syntax tree.

Samples filtering

After we transformed the code samples into CPGs using Joern, we noticed that we had an unequal number of positive and negative samples. Apart from that, there was also the possibility of having a number of negative CPGs among the positives.

For instance, if we work with code smells that refer to methods (as we do in this research project), many code samples contain more than one method. A sample is considered positive if at least one of its methods is affected by the long method or long parameter list smells. This means that we might have methods from positive code samples that are not containing any of these code smells.

The solution for this was performing filtering after finishing the Joern transformations. By doing so, we removed all the negative CPGs of methods from the positive samples directory.

Lastly, we also performed balancing, so that we can ensure that the number of positive and negative samples is equal. If the number of negative samples is smaller than the number of positive samples after filtering, we randomly remove positive samples until their numbers are equal. All of the previously mentioned actions are performed by the 'Pure Transformer'.

3.5.4 Sample labelling

The next operation performed by our 'Debtective' tool is the labelling of the samples, which is done by the group of components that we called 'Labeller', as seen in Figure 3.3. This group of components incorporates the '.dot Reader', the 'Feature Extractor', and the 'Torch Saver', which are created by us. It's important to note that for the feature extraction, we are using CodeBERT [7], a pretrained model for programming and natural languages, and we are using the Roberta Tokenizer [16] for tokenizing the input for CodeBERT.

When CodeBERT is used for feature extraction from Code Property Graphs (CPGs) of code samples, it encodes a series of important elements of the code. Each token (such as identifiers, keywords, and operators) is represented as a vector which captures the context of the token within the entire given code snippet [7]. These contextual embeddings reflect the relationships between the tokens and the structures within the given code. Additionally, CodeBERT embeddings also include structural information from the CPGs, such as control flow and dependencies between different parts of the code, including paths, branches, loops, and even function calls [7]. This representation allows for a better understanding of the execution flow and logic in the given sample.

The embeddings generated by CodeBERT also encode higher-level semantic relationships, capturing how functions relate to each other, how variables

are used across different scopes, and the interactions between various code sequences (control or data flow) [7]. This approach surpasses the limitations of performing a simple one-hot encoding [17], which, despite being interpretable, can lead to high dimensionality and scalability issues, while losing contextual information. Unlike one-hot encoding, the embeddings from CodeBERT are more expressive and efficient, encapsulating aspects which are more subtle from node types and their interactions within the code, making them highly suitable for complex tasks like code classification [7, 17].

The '.dot Reader' reads the negative and positive Code Property Graph samples that were generated by the previous group of components (also called 'Transformer' in our software architecture), and it passes them to the 'Feature Extractor' component. There, the input is tokenized by the Roberta Tokenizer [16], and it is transformed into tensors. These returned tensors are then passed to the CodeBERT model [7], which gives us the final output, for which, in the end, the 'Feature Extractor' component applies a target label of either 1 (positive), or 0 (negative). Lastly, the 'Torch Saver' component saves the positive and negative resulting tensors in separate directories, which will be then used for the training, the validation, or the evaluation of our given models.

It is important to note that the feature extraction happens for each one of the nodes, and CodeBERT [7] extracts 768-dimensional features for each one of the input nodes. This rich encapsulation helps in detecting Technical Debt by accurately identifying complex code patterns, dependencies, and anomalies that are indicative of SonarQube code smell presence. This is important for effectively achieving the desired performance with the models.

3.5.5 Model training

This group of components is called the 'Trainer'. As seen in Figure 3.3, our 'Trainer' comprises of the 'Model Selector', the actual 'Trainer' component and the 'Plotter'. This group ensures the complete training and validation of the models that we want to use. The 'Trainer' group performs the following steps:

First, based on the selected model, we create an instance of a model architecture with the given parameters for a specific run. The selector takes the model architecture that we want to use, and instantiates it with the given number of hidden channels. The models that can be selected need to be built individually and can be imported in the selector component.

Next, the 'Trainer' component takes parameters as criterion, batch size, learning rate, patience and number of epochs. It loads the data (both negative and positive samples) in a 'Custom Dataset', which is created based on the PyTorch Geometric Dataset [14]. The 'Trainer' uses the Adam Optimizer [13] for

[14]PyTorch Geometric Datasets: https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html.

stochastic optimization of the parameters of our selected model. During the training of the model, the 'Trainer' also performs validation after each epoch, using the 'Validator' component.

At the end of the training, the 'Trainer' has already found and saved the best model weights for three different parameters: accuracy, precision and f1-score. During the training, we check in every epoch if the current validation accuracy, precision or f1-score is greater than the previous maximum one, and we save the model replace the saved model weight with that one. This way, we can ensure that we have three different perspectives to investigate during the evaluation. The model weights are timestamped and also contain the SonarQube code smell in their name.

Finally, after the training is done, the results are plotted using the 'Plotter' component. This component uses the Matplotlib Python library [15] for plotting the metrics of the training, and it creates a series of graphs. In a typical figure created by this component, we can observe the graphs for loss, accuracy, precision, recall, and the f1-score. These include both training and validation metrics.

It is also worth mentioning that for the training, validation and testing, the dataset is split in proportions of 80%-10%-10%. The training

3.5.6 Model evaluation

The last group of components of our 'Debtective' tool is the 'Evaluator', as seen in Figure 3.3. It uses the same 'Custom Dataset', 'Model Selector' and 'Plotter' as the 'Trainer'. In this group of components, the only difference is made by using the 'Evaluator' component that we created.

The same model which was used in training is being selected for evaluation. Based on the given parameters, we can select the best training model weights for accuracy, precision or f1-score (as mentioned in Section 3.5.5). The metrics for choosing the best model weights are derived from validation, which is performed after each training step.

The 'Evaluator' is going to evaluate the selected model on an evaluation set, which, as mentioned in the previous section, represents 10% of the total data. During these steps, we will evaluate the performance of the model based on the classification of the given samples. Since we know the true labels of the samples, we can calculate the metrics of the model's performance.

Next, we will use the 'Plotter' component to create figures with the performance of the accuracy, precision, recall and f1-score metrics. These are placed over a vertical chart, having values in between 0.0 and 100.0. Lastly, in the same figure, we also include the confusion matrix, containing the predictions and their true values.

[15]Matplotlib: https://matplotlib.org.

In the end, all results are being stored in a 'Results Database', or locally in the project. This includes all the figures from training and evaluation, and also the model weights. The results are organised per model architecture, and contain timestamps and the name of the chosen code smell. Every figure contains the hyperparameters and their values, so that we can make sure that no piece of information is lost.

4 | Results

4.1 Long Parameter List

4.1.1 Training results

- **Model 1:**

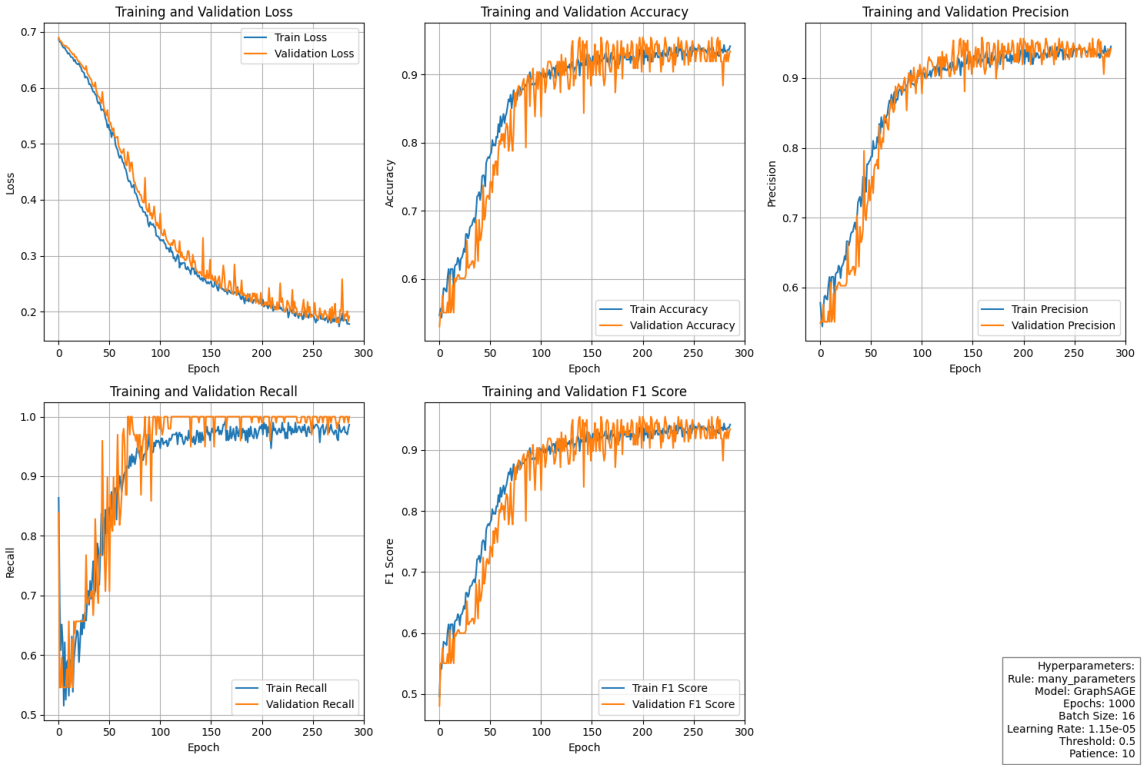

Figure 4.1: Training results for Model 1 on the 'Long Parameter List' code smell samples.

GraphSAGE Model

Process and Observations: The GraphSAGE model stopped after 280 epochs. We used a batch size of 16 and a learning rate of 1.15e-05. The learning rate was selected through experimentation. The training and validation loss curves show a steady decrease, indicating good

learning. Both training and validation accuracy gradually increase, approaching the 90% mark. The precision, recall, and F1-scores for both training and validation data present consistent improvement, maintaining high values above 85%. However, the model's performance shows fluctuation in validation metrics, particularly in recall, showing that the learning rate was too high.

- **Model 2:**

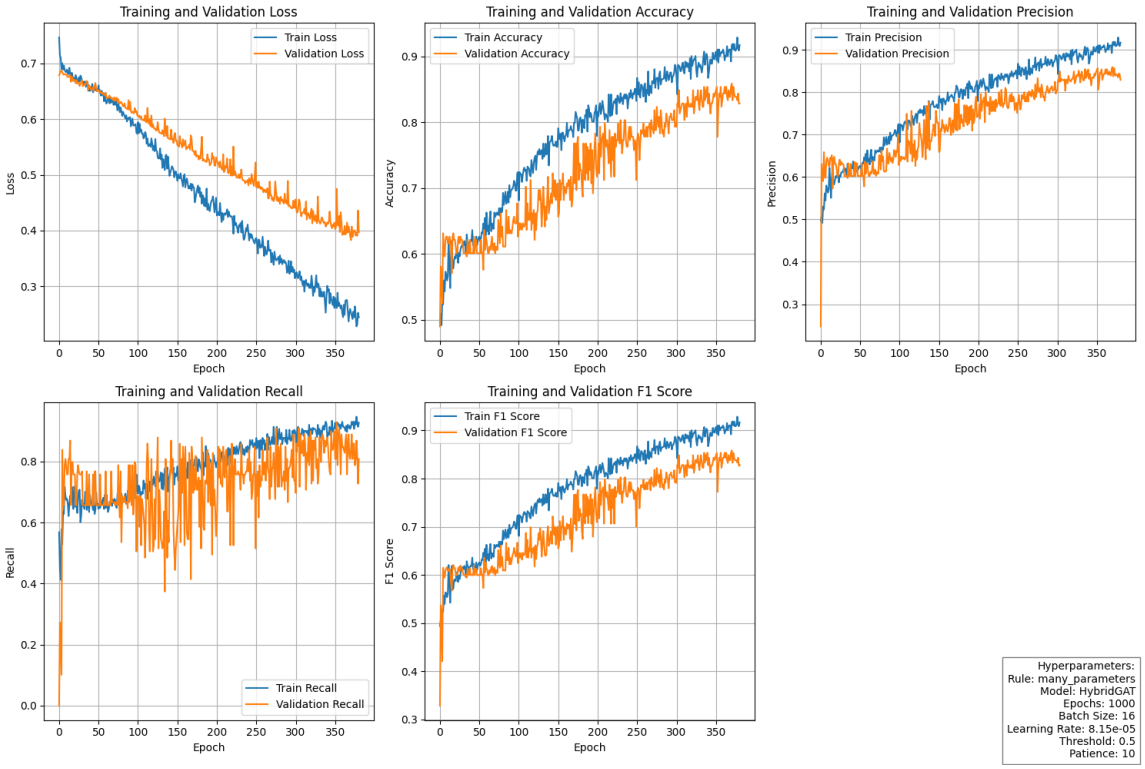

Figure 4.2: Training results for Model 2 on the 'Long Parameter List' code smell samples.

HybridGAT Model

Process and Observations: The HybridGAT models stopped after 375 epochs. We used a batch size of 16 and a learning rate of 8.15e-05. The learning rate was selected through experimentation. The loss curves show a decrease with spikes, while accuracy, precision, recall, and F1 scores for both training and validation data demonstrate fluctuation while improving. The metrics have not stabilised when early stopping was triggered. On the other hand, the validation metrics, particularly recall, show fluctuation, indicating that there might be potential sensitivity to high data variations. The complexity of this hybrid model may also lead to increased computational costs and longer training times.

- **Model 3:**

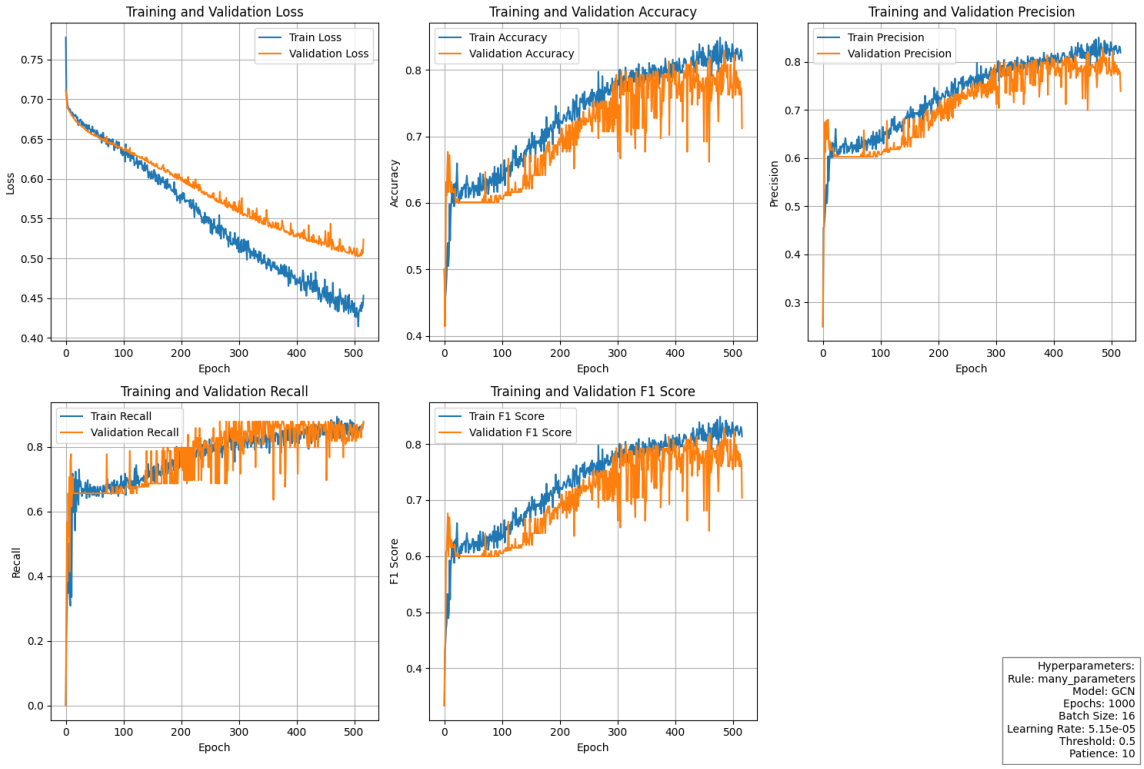

Figure 4.3: Training results for Model 3 on the 'Long Parameter List' code smell samples.

GCN Model

Process and Observations: Our GCN model stopped after 520 epochs. We used a batch size of 16 and a learning rate of 5.15e-05. The learning rate was selected through experimentation. The training and validation loss curves decline. Accuracy, precision, recall, and F1-scores for both training and validation data show significant improvement over time, with metrics stabilizing at high values. The GCN model demonstrates a fairly good performance, with some alignment between training and validation metrics, but indicating signs of overfitting. On the contrary, there are some fluctuations in the validation precision and recall, suggesting occasional difficulties in maintaining consistent performance across different epochs.

- **Model 4:**

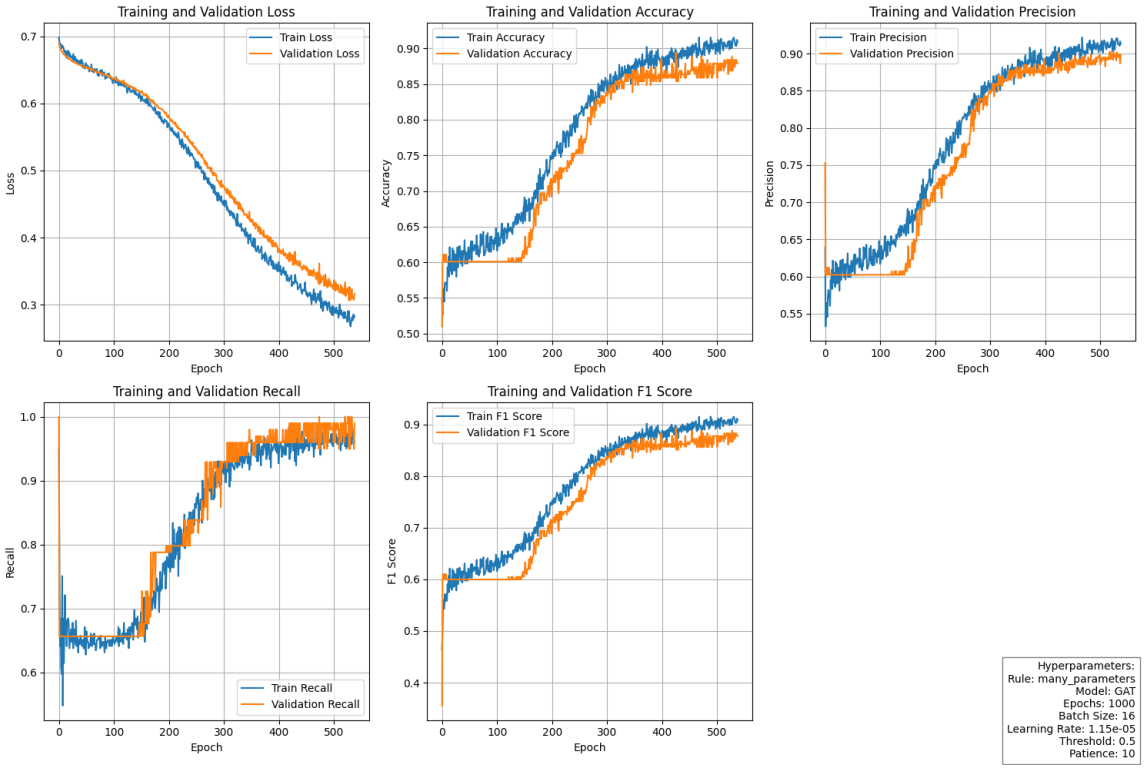

Figure 4.4: Training results for Model 4 on the 'Long Parameter List' code smell samples.

GAT Model

Process and Observations: The GAT (Graph Attention Network) model stoped after 530 epochs. We used a batch size of 16 and a learning rate of 1.15e-05. The learning rate was selected through experimentation. The loss curves show a decrease over the epochs, while accuracy, precision, recall, and F1-scores for both training and validation data present an upwards trend. The final metrics indicate strong performance, with high values across all metrics. The attention mechanism in GATs enables the model to capture more complex relationships within the graph, resulting in high accuracy and precision. On the other hand, the recall and F1-scores for validation data show some fluctuations, indicating potential instability in consistently detecting true positives.

4.1.2 Evaluation results

- **Model 1:**

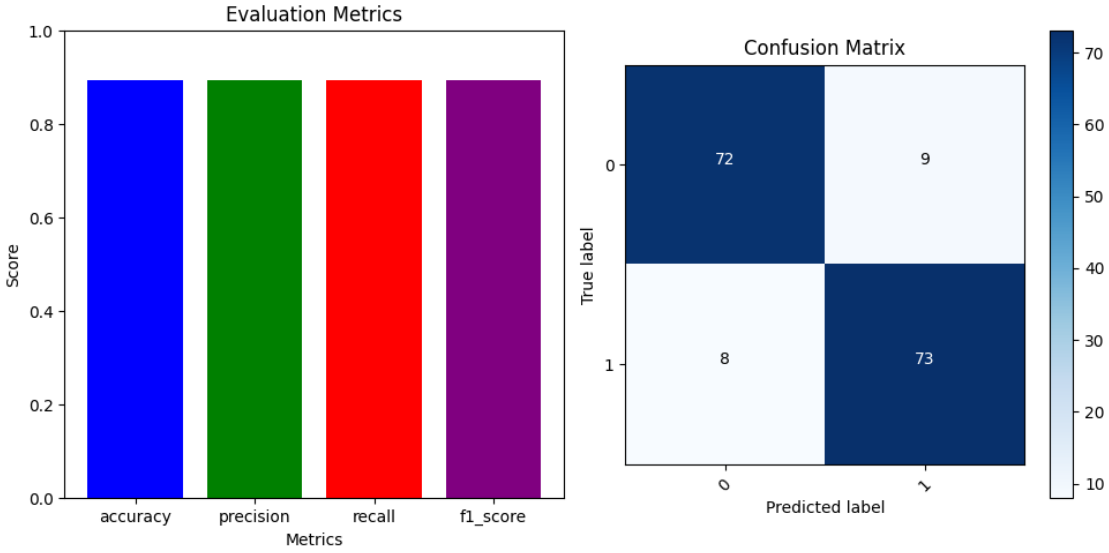

Figure 4.5: Evaluation results for Model 1 on the 'Long Parameter List' code smell samples.

The evaluation metrics for Model 1 (GraphSAGE), presents a strong performance, with high scores in accuracy, precision, recall, and F1 score, all close to 90%. From the confusion matrix we can see that the model accurately identifies both positive and negative instances, with 72 true negatives and 73 true positives, while having a few misclassifications too: 9 false positives and 8 false negatives.

Despite its overall high performance, the presence of 9 false positives and 8 false negatives suggests that the model occasionally misclassifies instances of the 'Long Parameter List' code smell.

- **Model 2:**

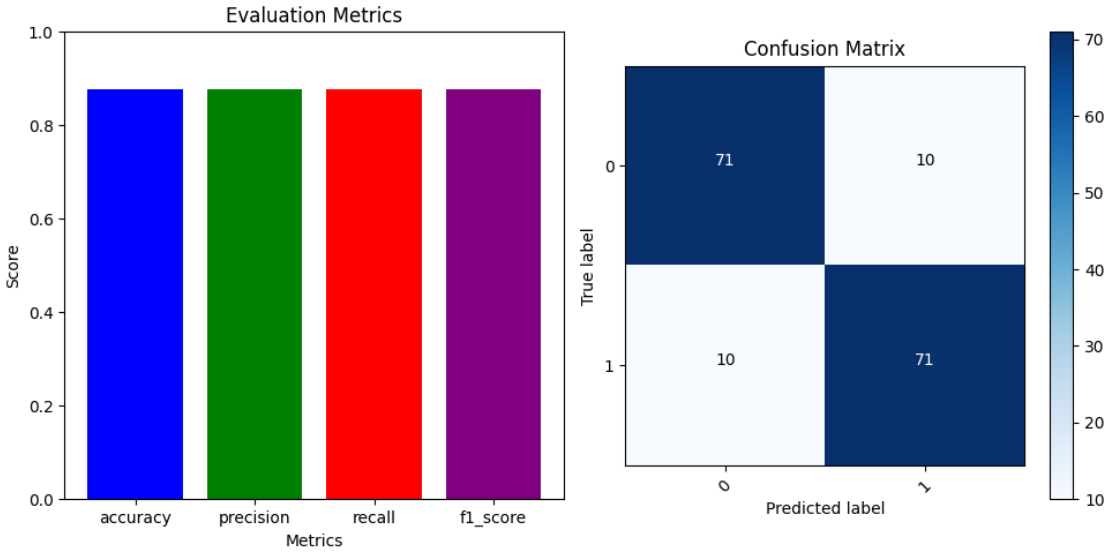

Figure 4.6: Evaluation results for Model 2 on the 'Long Parameter List' code smell samples.

The HybridGAT model (Model 2) also demonstrates strong performance, with slightly lower accuracy, precision, recall, and F1-score. The confusion matrix shows that the model has 71 true negatives and 71 true positives, but it also produced 10 false positive and 10 false negative results. This indicates that while the model seems to be effective, it still has room for future improvement in reducing errors.

While the HybridGAT model maintains high precision and recall, the equal number of false positives and false negatives suggests that the model could benefit from further refinement, especially in treating data variation. Reducing these errors could involve further fine-tuning the parameters, or modifying the architecture in order to improve its predictions for the 'Long Parameter List' code smell.

- **Model 3:**

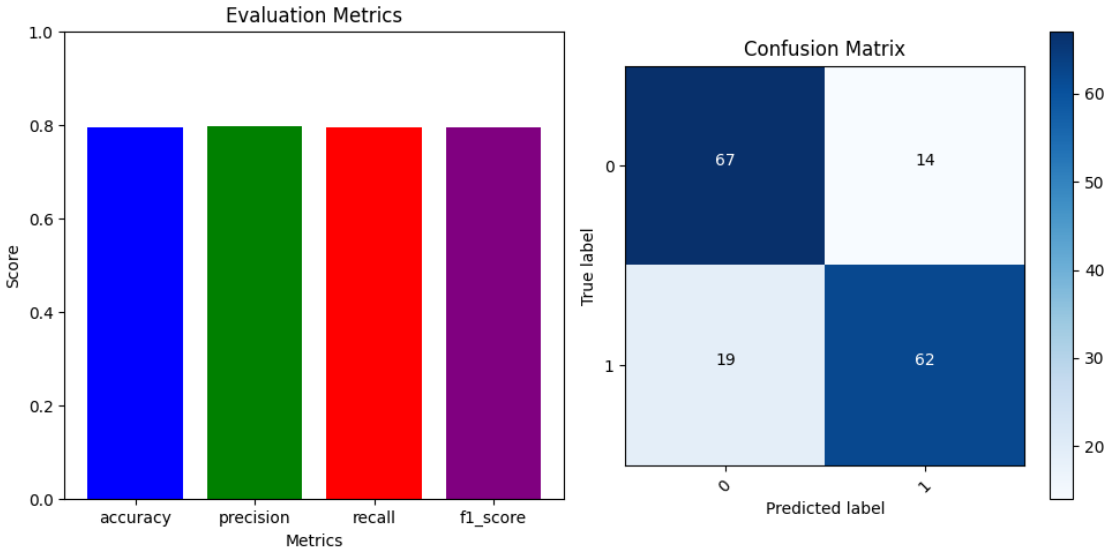

Figure 4.7: Evaluation results for Model 3 on the 'Long Parameter List' code smell samples.

The GCN model (Model 3) shows fairly good performance with the evaluation metrics, reflecting good accuracy, precision, recall, and F1-scores. The confusion matrix shows that the model has 67 true negatives and 62 true positives, with 14 false positives and 19 false negatives. This demonstrates the model's ability to accurately classify most instances, though there is still some room for reducing misclassification.

The GCN model's results are good, yet the presence of 14 false positives and 19 false negatives indicates potential areas for improvement. Enhancing the model's ability to correctly identify all instances of the 'Long Parameter List' code smell can involve further working for adjusting the sensitivity, or incorporating more diverse training data.

- **Model 4:**

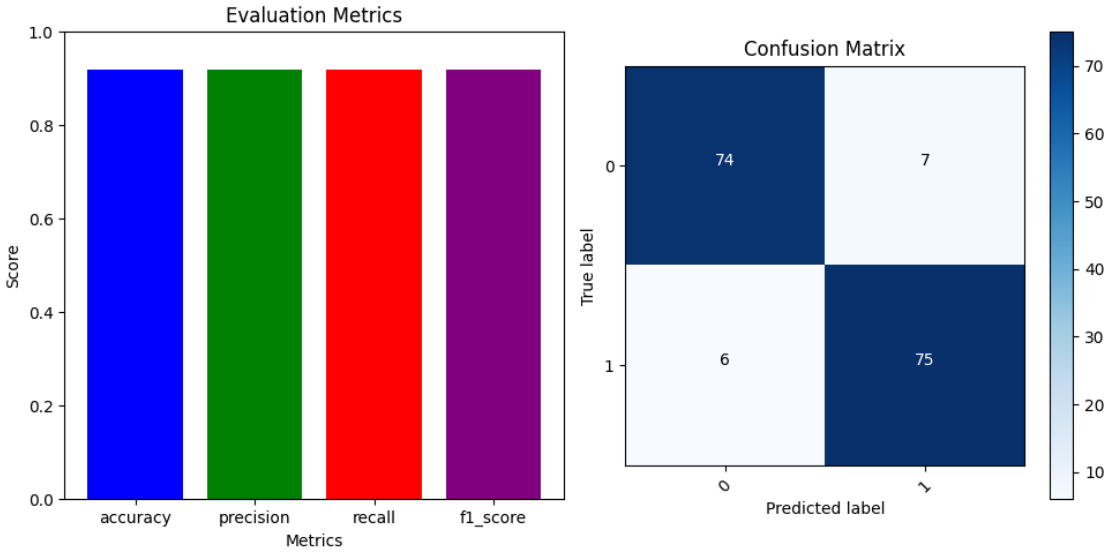

Figure 4.8: Evaluation results for Model 4 on the 'Long Parameter List' code smell samples.

The GAT model (Model 4) presents strong evaluation metrics with high scores in all accuracy, precision, recall, and F1-score. The confusion matrix shows that our model identified 74 true negatives and 75 true positives, while also misclassifying 7 false positives and 6 false negatives. This indicates that our model effectively identifies the majority of the instances, but has more difficulty in accurately classifying some samples.

Despite the high overall performance, the GAT model's 7 false positives and 6 false negatives suggest that there is room for improvement. Addressing these misclassifications could involve further refining our model's parameters, or incorporating additional variety in order to better distinguish between both the positive and negative instances.

4.2 Long Method

4.2.1 Training results

- **Model 1:**

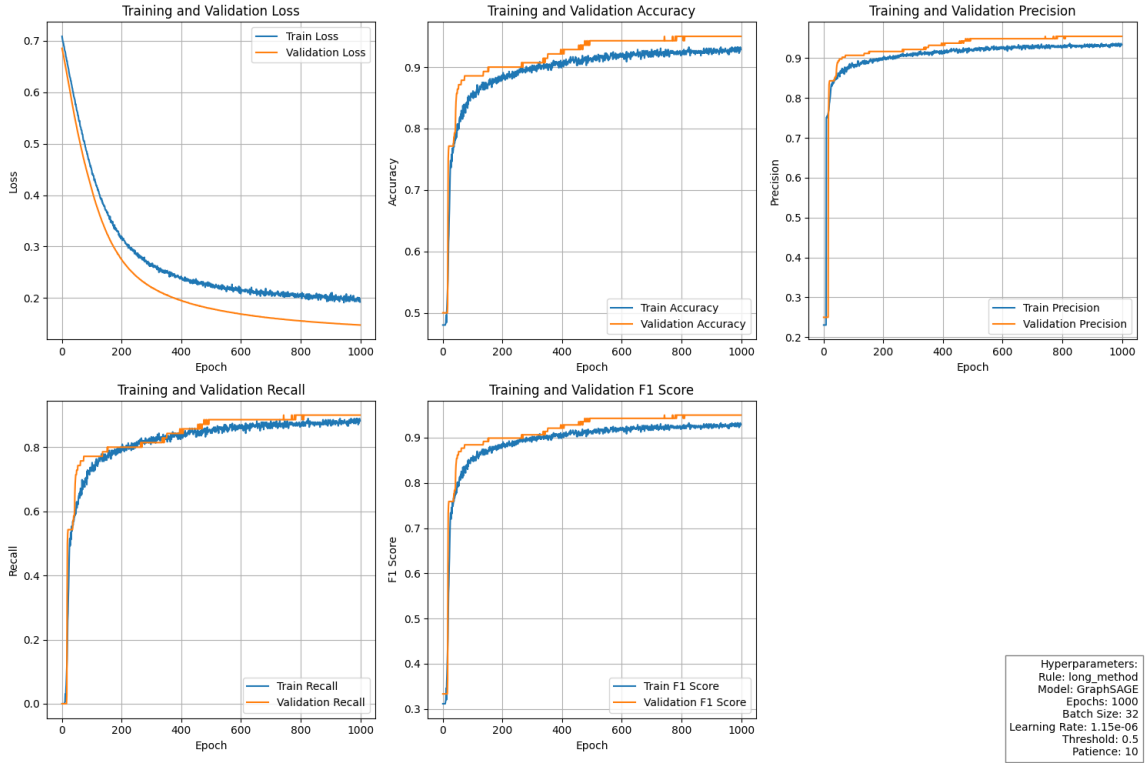

Figure 4.9: Training results for Model 1 on the 'Long Method' code smell samples.

GraphSAGE Model

Process and Observations: The GraphSAGE model stopped after 1000 epochs. We used a batch size of 32 and a learning rate of 1.15e-06. The learning rate was selected through experimentation. The training process saw a consistent reduction in loss and a corresponding increase in accuracy, precision, recall, and F1 score. The model's performance metrics for both training and validation data converge towards high values

- **Model 2:**

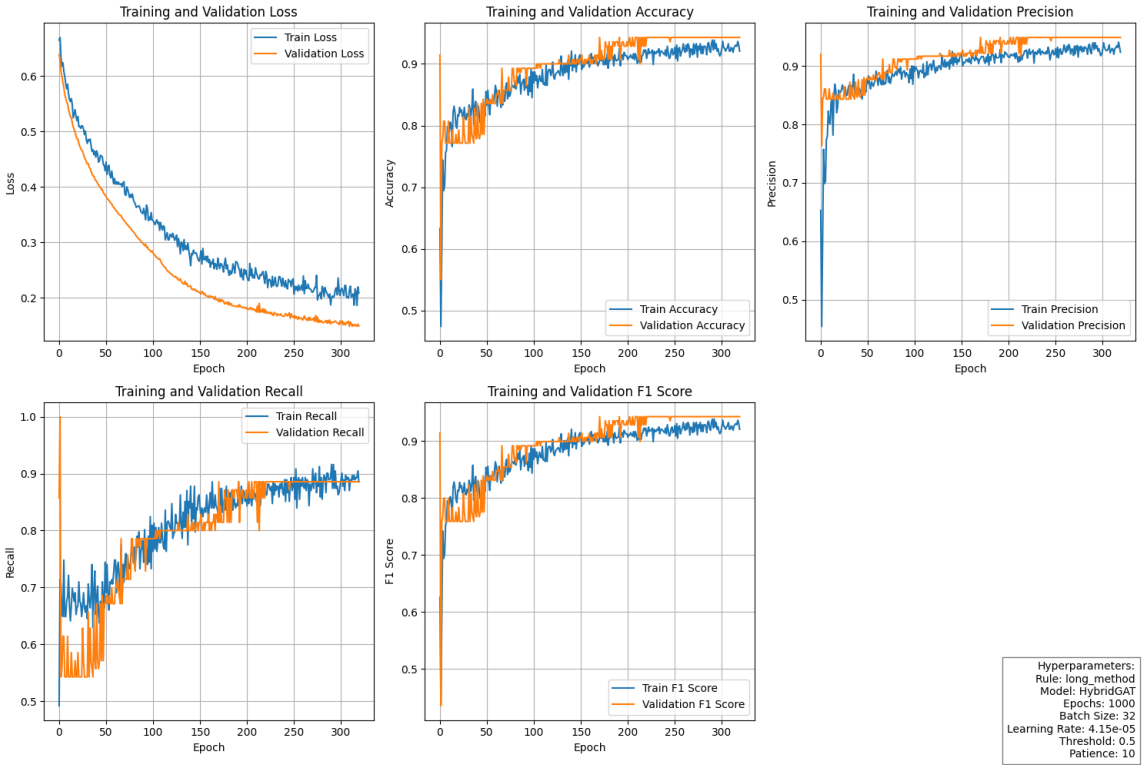

Figure 4.10: Training results for Model 2 on the 'Long Method' code smell samples.

HybridGAT Model

Process and Observations: The HybridGAT model stopped after 320 epochs. We used a batch size of 32 and a learning rate of 4.15e-05. The learning rate was selected through experimentation. The loss curves show a smooth decline, while accuracy, precision, recall, and F1 scores for both training and validation data show strong and consistent improvement. The high precision and recall indicate that the model makes reliable and accurate predictions, handling both positive and negative samples fairly well. However, the model's complexity lead to higher computational costs and longer training times compared to simpler models. Additionally, tuning such hybrid models proved to be more challenging, requiring careful consideration of the amount of hidden channels and the values of the hyperparameters.

- **Model 3:**

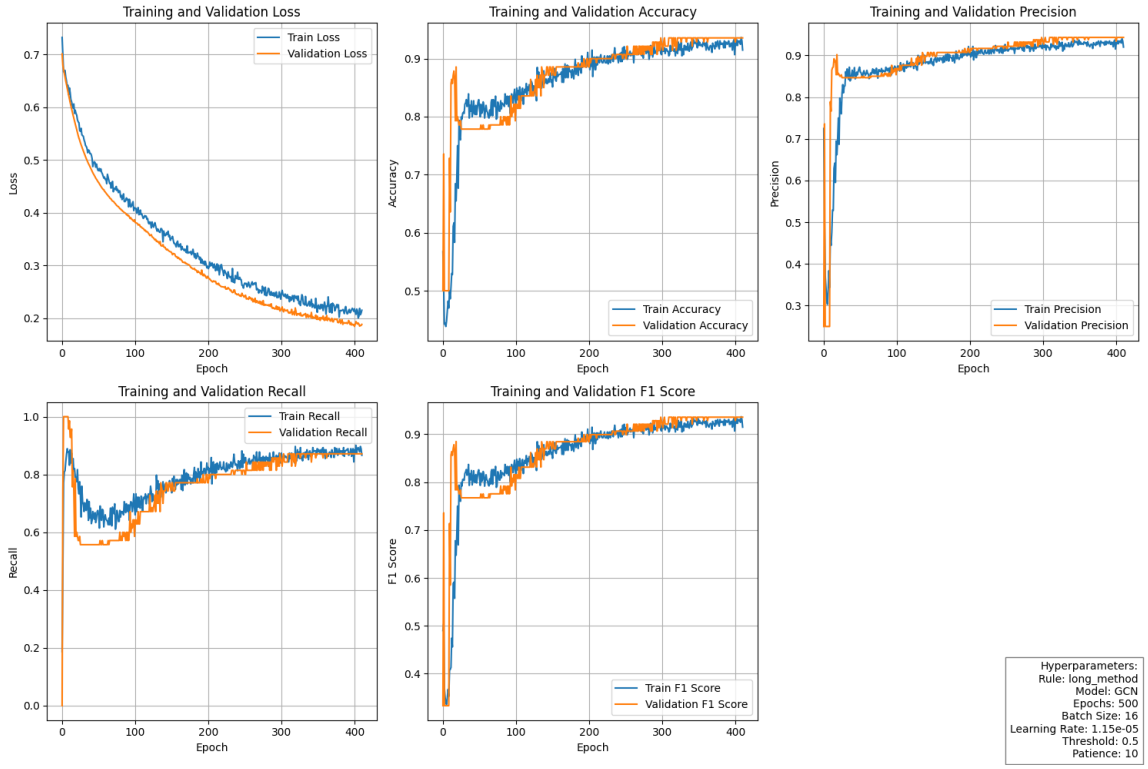

Figure 4.11: Training results for Model 3 on the 'Long Method' code smell samples.

GCN Model

Process and Observations: The GCN (Graph Convolutional Network) model stopped after for 410 epochs. We used a batch size of 16 and a learning rate of 1.15e-05. The learning rate was selected through experimentation. The training and validation loss curves demonstrate a consistent decrease, indicating that the model is learning well. Both training and validation accuracy show a steady increase, plateauing around the 90% mark. Precision, recall, and F1 scores for both the training and validation data also show substantial improvement, maintaining high values above 85%. There is a smooth decrease in the loss, and an increase in the accuracy without major fluctuations. This suggests that the learning rate and batch size were a good choice for this training instance. On the other hand, the training process is relatively slow, requiring a total of 410 epochs to be closer to converging. Additionally, the model might not capture very complex relationships within Code Property Graphs, because of the limitations of the basic graph convolution operation.

- **Model 4:**

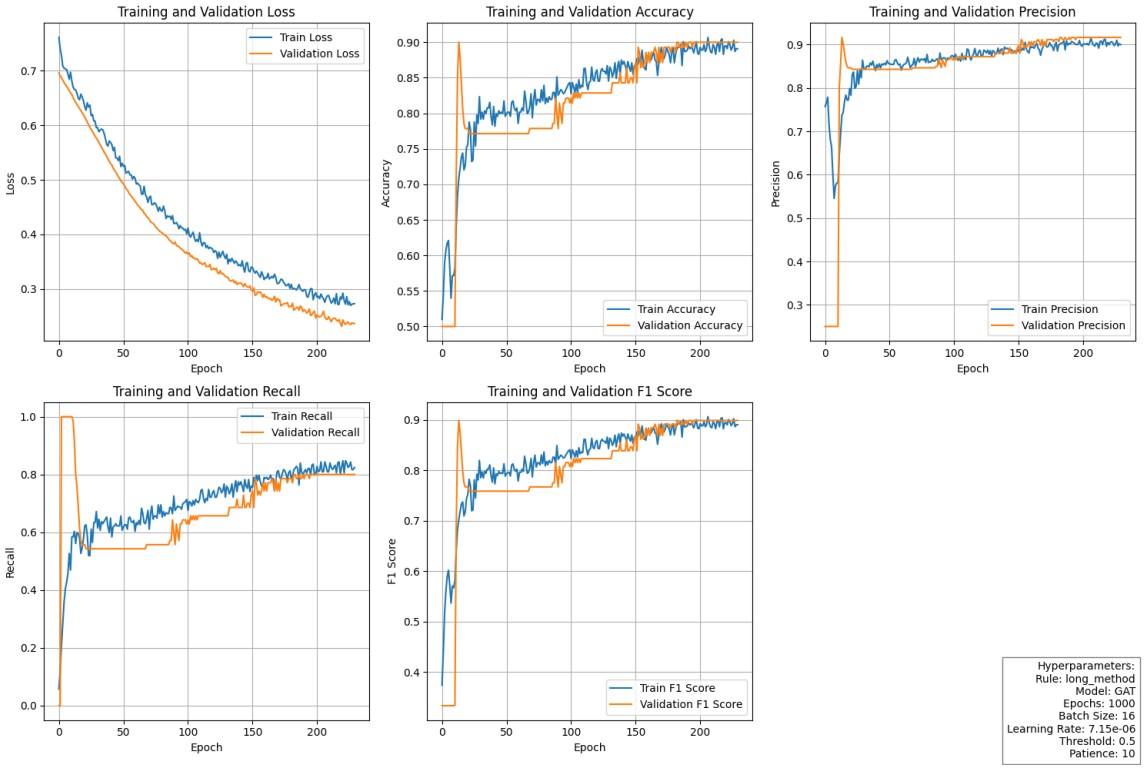

Figure 4.12: Training results for Model 4 on the 'Long Method' code smell samples.

GAT Model

Process and Observations: The GAT (Graph Attention Network) model stopped after 235 epochs. We used a batch size of 16 and a learning rate of 7.15e-06. The learning rate was selected through experimentation. The loss curves exhibit a steady decline, while the accuracy, precision, recall, and F1 scores for both training and validation data show significant improvement, stabilising at high values. The initial fluctuations in the precision and recall curves indicate the model's process of fine-tuning its parameters, in order to balance between different performance metrics. The attention mechanism in GATs allows the model to assign different weights to different nodes, capturing more complex relationships in the Code Property Graphs. High final values of precision, recall, and F1 score indicate in this case that the model is capable of making accurate and reliable predictions. However, the initial fluctuations in precision and recall suggest instability in the early stages of training, which proved to complicate the tuning process. Additionally, the model's training time is not that long, needing around 235 epochs to achieve high performance.

4.2.2 Evaluation results

- **Model 1:**

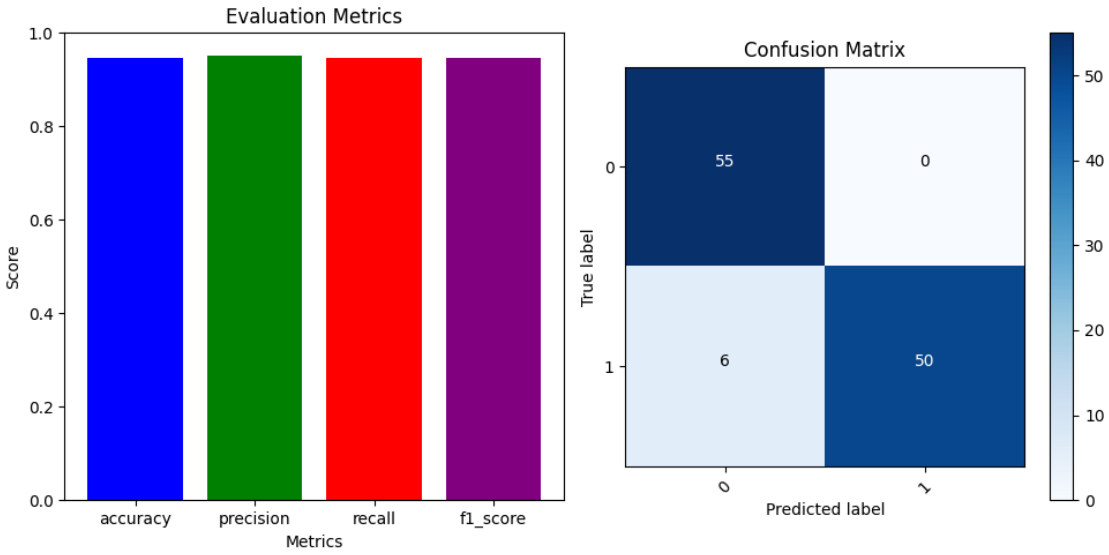

Figure 4.13: Evaluation results for Model 1 on the 'Long Method' code smell samples.

The evaluation metrics for the GraphSAGECustom (Model 1) display very good scores across all key performance indicators. The accuracy, precision, recall, and F1 score are all very high, hovering close to the 95% mark. This indicates that the model is highly effective at correctly identifying both positive and negative CPG samples of the 'Long Method' code smell. The confusion matrix further reinforces our observation, showing a minimal number of misclassifications: only 6 false negatives and 0 false positives out of a total of 111 evaluation samples.

Despite the overall strong performance, the presence of 6 false negatives suggests that the model occasionally misses instances of the 'Long Method' code smell. This could be an area for future improvement. Moreover, while the precision is great, a slightly higher recall could enhance the model's robustness in real-world analysis applications, where false negatives may be more important.

- **Model 2:**

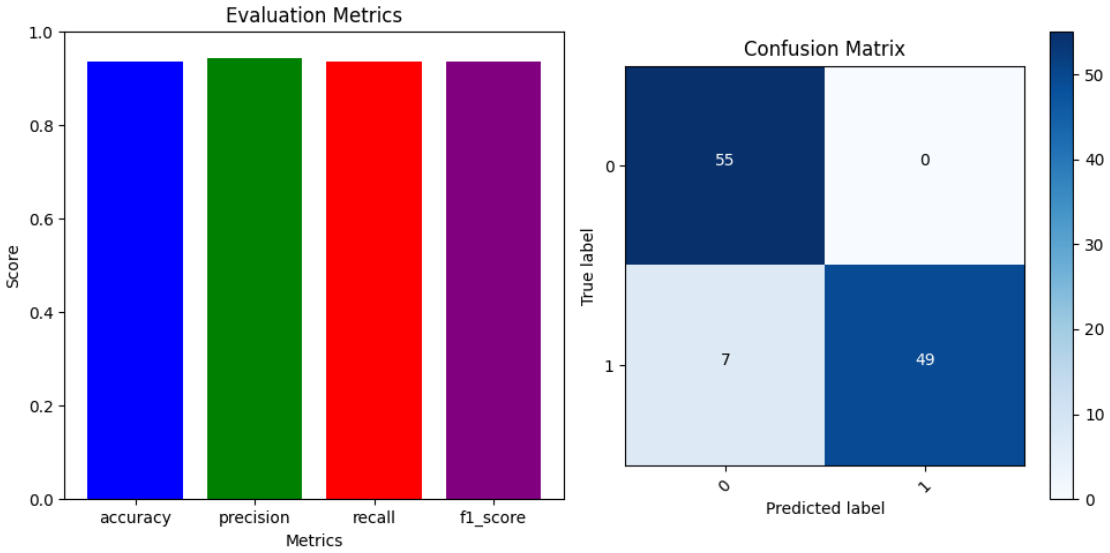

Figure 4.14: Evaluation results for Model 2 on the 'Long Method' code smell samples.

The HybridGAT model (Model 2) also shows a strong performance, with evaluation metrics showing high scores in accuracy, precision, recall, and F1-score. These metrics indicate that the model is well-calibrated and performs well in distinguishing between the presence and absence of the 'Long Method' code smell. The confusion matrix supports this case, with 55 true negatives and 48 true positives. On the other hand, it also reveals 8 false negatives.

Although the HybridGAT model maintains high precision and recall, the existence of 7 false negatives indicates that the model occasionally fails to detect some of the positives. This number suggests a slight need for improving its sensitivity towards positive cases.

- **Model 3:**

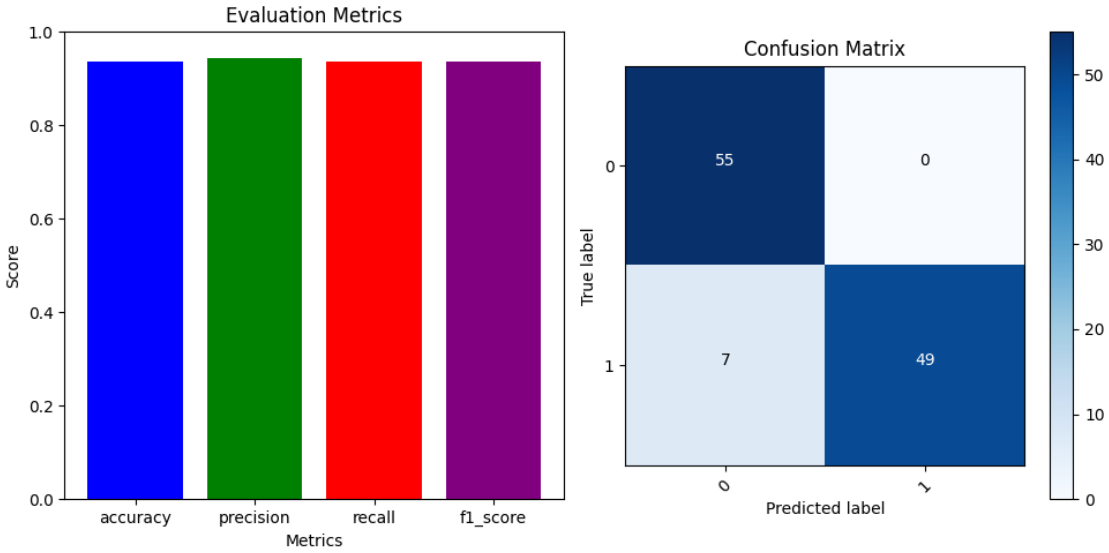

Figure 4.15: Evaluation results for Model 3 on the 'Long Method' code smell samples.

The GCN model (Model 3) demonstrates high performance with accuracy, precision, recall, and F1-score, all close to 95%. We can note from here that the model is fairly proficient in accurately classifying both positive and negative samples of the 'Long Method' code smell. The confusion matrix shows a total of 7 false negatives and no false positives, showing that, while the model is very good at detecting true negatives, there is still a small margin for missing some positive cases.

This might demonstrate a weakness in fully capturing all instances of this SonarQube code smell. In order to minimise these misses, a more diversification of the data would lead to a better generalisation.

- **Model 4:**

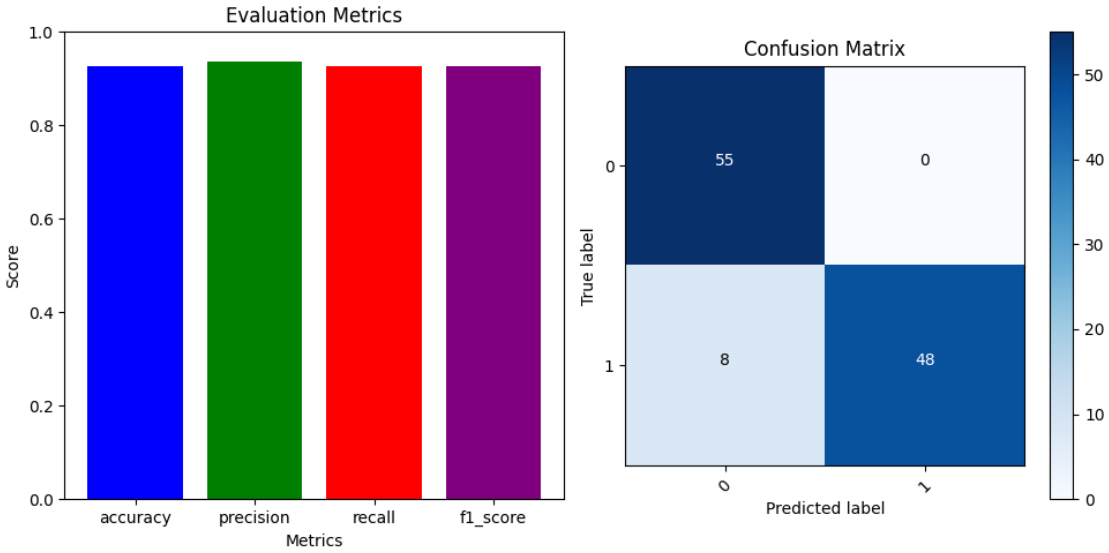

Figure 4.16: Evaluation results for Model 4 on the 'Long Method' code smell samples.

The GAT model (Model 4) shows robust evaluation metrics, with high scores across accuracy, precision, recall, and F1-score. These metrics indicate the model's good capability to accurately predict the 'Long Method' code smell. The confusion matrix illustrates 55 true negatives and 48 true positives, with 8 false negatives and no false positives, showing an emphasis on its precise classification of negative cases, but slight challenges in capturing all positive instances for this SonarQube rule.

Despite the high performance, the model's 8 false negatives represent an area for future improvement. In our view, this could be done through further tuning the hyperparameters, or adding more detailed features in order to capture a broader range of 'Long Method' code smell instances. This can thereby enhance the model's recall without compromising its current precision.

5 | Discussion

In this chapter, we will delve into an in-depth analysis of the results obtained in this research project, and presented in the previous chapter (Chapter 4). We applied four different graph-based models on the Code Property Graphs obtained from two SonarQube code smells types of samples: 'Long parameter list' (Section 3.5.2) and 'Long method' (Section 3.5.2). Our aim is now to interpret our findings, understand their implications to practitioners and researchers, and identify some potential threats to the validity of this study.

The different model architectures that we used bring unique strengths and characteristics that help us formulate an answer for the first research question. By evaluating the models' performances we can gain insights into which model architectures are the most effective for these two different types of code smells, and under what conditions they perform best.

We will begin by interpreting our results and discussing the obtained performance metrics in the contexts of 'Long parameter list' and 'Long Method' code smell detection (Section 5.1). Next, we will cover the implications of our findings for software engineers and researchers, providing a series of leads and suggestions for directions for future research (Section 5.3). Lastly, we will examine the threats to the validity of our research study, addressing potential limitations that could impact the ability to generalise and rely on these conclusions (Section 5.4). By doing so and acknowledging these current factors, our aim is to provide a balanced and comprehensive perspective that can provide a solid foundation for future advancements in this research area.

5.1 Interpretation of Results

In order to fully understand the performance of the models that we built, we will first take a look how they performed in the detection of the two SonarQube code smells overall. If we analyse the training metrics for 'Long method', we can see that generally all models (Figures 4.9, 4.10, 4.11 and 4.12) showed a decrease in loss with minimal gaps between the training and validation loss, thus indicated a good fit. The metrics for accuracy, precision, recall and F1-score were consistently high, and there was a close alignment between the training and validation overall.

On the other hand, the training metrics for 'Long parameter list' for all models (Figures 4.1, 4.2, 4.3 and 4.4) showed a tendency for overfitting, particularly noticeable in the case of Model 2 (HybridGAT) and Model 3 (GCN) (Figures 4.2, Figures 4.3). Those two models had more significant gaps between the training and validation metrics. Overall, we could notice that compared to the 'Long method' training metrics, for this code smell the accuracy, precision, recall and F1-scores were fairly high across most models, but the validation metrics would be slightly lower.

Now, if we also analyse the evaluation metrics, we can clearly notice that in a 'Long method' context, the models performed generally better than in a 'Long parameter list' context. Figures 4.13, 4.14, 4.15 and 4.16 all indicate accuracy, precision, recall and F1-score metrics in the interval 90%-95%, with no false positives being classified and Model 1 (GraphSAGE) being on top. However, if we take a look at figures 4.5, 4.6, 4.7 and 4.8, we can notice that the same metrics are situated in the interval 80%-90%, with Model 4 (GAT) performing the best.

We will now summarise the key takeaways for each model based on their training and evaluation performances for both code smells:

- **Model 1: GraphSAGE**
  - **Long Parameter List**:
    - **Accuracy**: Very High, close to 90%.
    - **Precision, Recall, F1 Score**: High, almost identical to accuracy.
    - **Confusion Matrix**: Some misclassifications, indicating a good performance.
  - **Long Method**:
    - **Accuracy**: Very high, close to 95%.
    - **Precision, Recall, F1 Score**: Very high, around the 95% mark.
    - **Confusion Matrix**: Very few misclassifications (no false positives), indicating a great performance.

- **Model 2: HybridGAT**
  - **Long Parameter List**:
    - **Accuracy**: Slightly lower than GraphSAGE, below 90%.
    - **Precision, Recall, F1 Score**: Similar to accuracy.
    - **Confusion Matrix**: More misclassifications than GraphSAGE
  - **Long Method**:
    - **Accuracy**: High, slightly below Model 1.
    - **Precision, Recall, F1 Score**: Close to accuracy.
    - **Confusion Matrix**: Fair amount of misclassifications (only one more false negative compared to Model 1), indicating great performance.
- **Model 3: GCN**
  - **Long Parameter List**:
    - **Accuracy**: Lowest overall, close to 80%.
    - **Precision, Recall, F1 Score**: Also around the 80% mark.
    - **Confusion Matrix**: Significantly increased number of misclassifications, indicating overfitting. Highest amount of misclassifications
  - **Long Method**:
    - **Accuracy**: Similar to Model 2.
    - **Precision, Recall, F1 Score**: High, but slightly lower than Model 1.
    - **Confusion Matrix**: Same amount of false negatives as Model 2, fair amount of misclassifications.
- **Model 4: GAT**
  - **Long Parameter List**:
    - **Accuracy**: Very similar to Model 1.
    - **Precision, Recall, F1 Score**: Close to the value of accuracy.
    - **Confusion Matrix**: Lowest amount of misclassifications.
  - **Long Method**:
    - **Accuracy**: High, similar to Models 2 and 3.

* **Precision, Recall, F1 Score**: Around the same mark as accuracy.
* **Confusion Matrix**: Still a fair amount of misclassifications, only one more false negative compared to Models 2 and 3.

From the key takeaways and from the training and evaluation metrics from Chapter 4, we can clearly notice that Model 1 (GraphSAGE) and Model 4 (GAT) performed better for the code smell 'Long parameter list', and that the performance of all four models was very similar for 'Long method', with Model 1 coming on top again. Overall, Model 1 is the only one to have the better fit in both contexts, and it also maintained the lowest fluctuation across the training and validation metrics.

Another observation that can be made from the results of the training and evaluation is that Model 2 (GCN) and Model 3 (HybridGAT) are more prone to overfitting in the context of 'Long parameter list', or are too complex. This might happen due to the fact that both models are using GCN layers, which in this case can lead to high fluctuations of the training and validation metrics. HybridGAT on the other hand, has a smaller gap between the training and validation metrics, since it also encompases a GAT layer.

One important aspect that might influence the models to perform better in the 'Long method' context might be the difference in size in CPGs created from the negative and positive code samples. If we are looking for 'Long parameter list' in a sample, we analyse the number of parameter nodes in the Code Property Graph of a method. If the number of parameter nodes is above a given threshold (more than four parameters), then a sample is considered positive, or otherwise negative. In this context, we can encounter CPGs of samples that have similar sizes, but a different number of parameters (one being negative, the other positive).

On the other hand, in the context of 'Long method', we are interested in analysing the number of statements (and their nodes) in the Code Property Graph of a method. If the number of statements is greater than a given threshold, than we encountered a long method, otherwise it is a negative sample of code. Generally, in this case, positive samples have CPGs with larger sizes compared to negative samples.

Lastly, as it can be seen from the evaluation metrics, the task might have been slightly too easy for the models. The metrics are very high, supporting this claim. In our view, this might be a consequence of the simple nature of the selected code smells. Both 'Long parameter list' and 'Long method' are structural SonarQube code smells, referring to the number of statements or the number of parameters in a method.

5.2 Addressing the Research Question

Research Question: How effective are GCN, GAT and GraphSAGE graph-based models in detecting technical debt in existing technical debt datasets?

Our proposed methodology involved applying four different graph-based model architectures that we built on the 'Technical Debt Dataset' [15], and transforming code samples for the selected code smells into Code Property Graphs (CPGs). The results that we achieved demonstrated that models like GraphSAGE and GAT are highly effective in detecting both selected code smells. These model architectures achieved high accuracy, precision, recall, and F1-scores for both 'Long Parameter List' and 'Long Method', and, in this context, they were proven to be effective tools for technical debt detection. Of course, this question still remains open, since there are a lot of other different types of SonarQube code smells and model architectures that we haven't tried, but this provides solid grounds for future research in this area. Through our experimentation, we managed to confirm the potential that graph learning has in technical debt detection.

- **Models' Performances - Key Takeaways**:
  - **GraphSAGE (Model 1)** and **GAT (Model 4)** were the top performing models, showing strong generalisation abilities and an increased accuracy in detecting both types of code smells. These models demonstrated robustness against overfitting and provided reliable detection metrics. Excellent performance
  - **HybridGAT (Model 2)** and **GCN (Model 3)** showed a lower effectiveness, but were more prone to overfitting, especially for the 'Long Parameter List' SonarQube code smell.
  - **Overall Performance**: Despite the variation in individual model performance, the overall effectiveness of the four graph-based models in detecting technical debt was affirmed.
  - **Difficulty of the Task:** Especially for rules like 'Long method' or 'Long parameter list', the task is not that difficult for the model architectures employed. Both SonarQube code smells are structural in nature, and refer to the number of statements and the number of parameter nodes.
  - **Possible Problem #1:** There might still be some data leakage for the SonarQube code smell 'Long parameter list', which might have impacted the training and performance of the models.
  - **Possible Problem #2:** The (big) fluctuations in training metrics might have been caused by the increased complexity of the models employed, as opposed to a learning rate issue.

5.3 Implications to Practitioners and Researchers

The aim of this research project was to investigate the effectiveness of four graph-based models in detecting technical debt, and how Code Property Graphs (CPGs) impact their effectiveness. Based on the results that we achieved, we can clearly see that there is a lot of potential in this area. This is further supported by the significant results obtained by applying graph learning for vulnerability detection, as stated in the background chapter (Chapter 2), since technical debt and vulnerabilities are similar in nature (in graph learning).

For practitioners, being able to detect technical debt effectively results in reductions in additional costs, effort and time for software maintenance and evolution. Through our approach using Code Property Graphs, we aimed at finding a singular graph representation that could be used in multiple analysis tasks. This could potentially reduce the complexity of tools that extract multiple different graph representations from the given code samples.

Also, the 'Debtective' tool that we created (see Section 3.5), can be a good starting point for other tools that aim to centralise the preprocessing, training and evaluation processes in a single place. Through our methodology, we tried to simplify the process of using multiple model architectures with a graph representation that encapsulates all information needed for various analysis tasks.

For researchers, the implications of our proposed methodology and the results that we achieved are that there is a lot of potential in this research area. We only managed to cover a limited amount of code samples and model architectures, but there are many possibilities and questions open. This research project offers a perspective that the four selected graph-based models are effective in identifying the complex relations within code samples, and this might provide a starting point for creating a multi-smell detection model. The CPGs played a significant role in our study, but if those could be 'paired' with a model that is successful in identifying more code smells simultaneously (both semantic and structural), this would lead to great advancements in the area of technical debt detection. Researchers can explore multi-task learning approaches and other advanced techniques, in order to create models that are capable of handling the complexity of real-world codebases and their large sizes.

5.4 Threats to Validity

This research project covers only a limited number of code smells, which are currently defined by SonarQube. The scope of our analysis and model training is limited to only a small amount of selected code smells (two in the context of this thesis), which may not be representative for the full spectrum of code smells that could help identify technical debt. This limitation means that the models' applicability and effectiveness might be constrained when encountering other types of code smells not included in our study. As a consequence, the findings and model performance might be specific only to the chosen smells, and additional research is needed in order to explore and fully validate the selected models' capabilities across a broader range of code smells.

Furthermore, the analysis was performed exclusively on Java code samples. Java has well-developed coding standards, patterns, and practices, which might not be as mature or well-documented in other programming languages. As a result, the findings and performance of the models trained for our study might not generalize well to other languages with different coding conventions and practices (SonarQube rules might also differ). As an example, languages like Python, JavaScript, or Haskell have their own unique idioms and style guides, which could affect how code smells are detected and addressed. The specificity of Java's coding context could mean that the models may not capture the nuances present in other programming languages, thereby limiting our ability to generalise our results.

Another important point is that the models in this study were trained to detect individual code smells. Developing a model capable of identifying multiple code smells in a single analysis is a more complex task that requires more complicated approach. The complexity of such models is significantly higher, and the development process was beyond the scope of this project. A multi-code smell detection model, for example, would need to account for the interactions and dependencies between more code smells (which can be different in nature: semantic or structural). This could definitely lead to a more complicated training and evaluation process.

Lastly, the training data used in this study was limited to the files of projects from the dataset. A larger and more diverse approach might improve the model's performance and ability to generalise. A smaller or project-based approach on code smells can introduce biases or fail to capture the full variability of code smells in real-world software projects, thus affecting the models' abilities to generalise to unseen examples. Including samples with higher varieties for each code smell in the future might lead to better performances and generalisation.

5.4.1 Reflections on Dataset

The dataset that we used for this study was the 'Technical Debt Dataset' [15], which included metrics data from 33 Java projects within the Apache Software Foundation. This dataset was chosen for its comprehensive analysis of commits, SonarQube to gather technical debt information. The structured format of the dataset, accessible through an SQLite database, greatly facilitated data queries and scripting necessary for our analysis.

A very important aspect of the dataset was its detailed information on fault-inducing and fault-fixing commits, which was instrumental in distinguishing positive and negative samples for training our models. The ability to join data from multiple tables using the projectID key allowed us to create detailed project-specific information tables. Despite these advantages, the preprocessing phase was challenging due to the dataset's size and complexity. Another important point is that even some code samples are negative samples for a code smell, they might be positive samples for another code smell that is not investigated by the model. There was no guarantee that the negative samples were negative entirely, in the context of our dataset.

The variation of the samples for the SonarQube code smells that we analysed was good. The number of contributors in each one of the projects analyised in the 'Technical Debt Dataset' was high, and the timeframe was spanning almost a decade. This ensured that we had multiple coding styles in this project, which was a positive aspect for the variety of the data.

On the other hand, there were a few instances where pairs of fault-inducing and fault-fixing commit files were not useful. For example, in the fault-inducing commit there was a file containing a code smell, but the problem was fixed in the fault-fixing commit by removing that specific file, instead of fixing the code. This lead to a difference in numbers between the positive and negative samples, requiring us to repair the balance after filtering, later in the preprocessing.

One important consideration for our study was the collection of negative the samples for our analysis tasks. The negative samples collected for training, validation and testing were taken mainly from the fault-fixing commits. Since the code smells that we investigated are structural code smells, referring to the number of parameters and the number of statements in methods, this shouldn't produce any bias in the training of the models that we used. Additionally, the only bias that might exist is in the fact that for training, validation and testing we used positive and negative samples from the same project for each analysis task.

Training our models on this dataset, which is based on SonarQube code smells, brings both advantages and limitations. While this provides a structured manner of identifying technical debt, it trained the models to detect what basically SonarQube could do already. This means that the effectiveness of our model architectures that we used is tightly coupled with the ef-

fectiveness of SonarQube's detection capabilities. Consequently, our models might encounter some of the limitations of SonarQube, potentially missing other types of code smells or technical debt that SonarQube currently does not cover. In the end, using SonarQube as a basis allowed us to ensure that our findings are aligned with a widely-accepted standard in the industry, and follow the detection of a well-established tool, both in academia and in industry.

Overall, the 'Technical Debt Dataset' was a very useful collection of data. Despite this series of negative points, it proved important for obtaining the findings of our project. It also impacted our 'Debtective' tool's architecture, and provided us with a good starting point for this research project.

6 | Conclusions

The aim of our research project was to investigate the effectiveness of graph-based models in detecting technical debt. We focused specifically on a limited set of SonarQube code smells, out of which two were presented in this thesis: 'Long Parameter List' (Section 3.5.2) and 'Long Method' (Section 3.5.2). Through building the necessary instrumentation (Section 3.5) and running our analysis tasks (Chapter 4), we have made a series of findings.

6.1 Summary of our Findings

The study demonstrated that graph learning models, particularly GraphSAGE and GAT, are highly effective in detecting some specific types of technical debt, such as 'Long Parameter List' and 'Long Method' code smells. The models showed very high performance metrics, indicating their robustness in identifying these issues within software projects. On the other hand, these very high metrics might be a consequence of tasks that were too easy for the models that we employed.

The results indicated that GraphSAGE and GAT outperformed other model architectures generally, achieving higher precision, recall, and F1-scores. This highlights the suitability of these models. for technical debt detection tasks, and their capability to provide valuable insights into the quality of the code in structural contexts. The consistent performance across different test scenarios shows the reliability and adaptability of the models for other SonarQube code smells.

In addition to the performance of the models, the study also highlighted the importance of preprocessing and data handling, which influences the performance of the selected graph-learning models. The use of Code Property Graphs (CPGs) was crucial in capturing the necessary code attributes for effective model training. By using CPGs, we managed to transform the samples for universal analysis tasks. The CPGs of the given code samples contain information that can be used not only in analysing structural code smells (as we do in this study), but also in the analysis of more complex, semantic code smells.

6.2 Instrumentation: 'Debtective'

To streamline the process of data gathering, preprocessing, and model training and evaluation, we developed the tool called 'Debtective'. With our tool we aimed at automatising the conversion of code samples into graph representations, applying the necessary preprocessing tasks, and training the various graph-based models that we built on these samples. 'Debtective' proved to be an effective asset in this research, significantly reducing our manual effort, and ensuring that we have a consistent and systematic approach for our models' evaluations. By integrating multiple model architectures within a single analysis pipeline, 'Debtective' provides a scalable starting point which can be used for future, more complex technical debt detection studies and tools.

6.3 Future Work

Building on the findings of our research project, there are a few elements which can provide initial directions for future research in this area:

- **Multi-Smell Detection**: Developing models capable of detecting multiple code smells simultaneously would address a larger range of technical debt issues, providing more support for software maintenance. On the other hand, this is a process that has an increased complexity.
- **Cross-Language Applicability**: Extending the application of these models to other programming languages could lead to the development of more versatile and widely applicable technical debt detection tools. Similar instrumentation can be also developed from scaling the 'Debtective' tool that we created.
- **Integration with existing CI/CD Pipelines**: Incorporating these models into continuous integration/continuous deployment (CI/CD) pipelines that could enable the detection of technical debt in real-time. This is also a complex process, and such tools provide challenges especially when we are discussing about the necessary resources.

This research project provided results which emphasise the potential of the four selected graph-based models in advancing the detection and management of technical debt. By addressing the limitations that we identified, and exploring the some of the future directions, both the academic community and the software industry can benefit from more effective and efficient technical debt management practices and tools. The findings from this project can provide the groundwork for ongoing improvements in this area, creating opportunities for more sustainable and maintainable software practices.

Bibliography

[1] Abien Fred Agarap. "Deep Learning using Rectified Linear Units (ReLU)". In: *CoRR* abs/1803.08375 (2018). arXiv: 1803.08375. URL: http://arxiv.org/abs/1803.08375.

[2] Frances E. Allen. "Control flow analysis". In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: Association for Computing Machinery, 1970, 1–19. ISBN: 9781450373869. DOI: 10.1145/800028.808479. URL: https://doi.org/10.1145/800028.808479.

[3] P. Avgeriou et al. "Technical Debt Management: The Road Ahead for Successful Software Delivery". In: *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023, pp. 15–30. DOI: 10.1109/ICSE-FoSE59343.2023.00007. URL: https://doi.ieeecomputersociety.org/10.1109/ICSE-FoSE59343.2023.00007.

[4] Robert Baggen et al. "Standardized code quality benchmarking for improving software maintainability". In: *Software Quality Journal* 20.2 (June 2012), pp. 287–307. ISSN: 1573-1367. DOI: 10.1007/s11219-011-9144-9. URL: https://doi.org/10.1007/s11219-011-9144-9.

[5] Sicong Cao et al. "BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection". In: *Information and Software Technology* 136 (2021), p. 106576. ISSN: 0950-5849. DOI: https://dx.doi.org/10.1016/j.infsof.2021.106576. URL: https://www.sciencedirect.com/science/article/pii/S0950584921000586.

[6] José M. Conejero et al. "Early evaluation of technical debt impact on maintainability". In: *Journal of Systems and Software* 142 (2018), pp. 92–114. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.04.035. URL: https://www.sciencedirect.com/science/article/pii/S0164121218300736.

[7] Zhangyin Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Ed. by Trevor Cohn, Yulan He, and Yang Liu. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139.

URL: https://aclanthology.org/2020.findings-emnlp.139.

[8] Jeanne Ferrante, Karl Ottenstein, and Joe Warren. "The Program Dependence Graph and Its Use in Optimization." In: *ACM Transactions on Programming Languages and Systems* 9 (July 1987), pp. 319–349. DOI: 10.1145/24039.24041.

[9] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf.

[10] Van Thuy Hoang et al. "Graph Representation Learning and Its Applications: A Survey". In: *Sensors* 23.8 (2023). ISSN: 1424-8220. URL: https://www.mdpi.com/1424-8220/23/8/4168.

[11] Kavi, Buckles, and Bhat. "A Formal Definition of Data Flow Graph Models". In: *IEEE Transactions on Computers* C-35.11 (1986), pp. 940–948. DOI: 10.1109/TC.1986.1676696.

[12] Shi Ke et al. "MPT-embedding: An unsupervised representation learning of code for software defect prediction". In: *Journal of Software: Evolution and Process* 33 (Dec. 2020). DOI: 10.1002/smr.2330.

[13] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). URL: https://api.semanticscholar.org/CorpusID:6628106.

[14] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. arXiv: 1609.02907 [cs.LG]. URL: https://arxiv.org/abs/1609.02907.

[15] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. "The Technical Debt Dataset". In: *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE'19. Association for Computing Machinery. New York, NY, USA, 2019, pp. 2–11. DOI: http://dx.doi.org/10.1145/3345629.3345630.

[16] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: 1907.11692 [cs.CL]. URL: https://arxiv.org/abs/1907.11692.

[17] Jamell Samuels. *One-Hot Encoding and Two-Hot Encoding: An Introduction*. Jan. 2024. DOI: 10.13140/RG.2.2.21459.76327.

[18] 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)*. 2021, pp. 590–594. DOI: http://dx.doi.org/10.1109/MSR52588.2021.00080.

[19] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *J. Mach. Learn. Res.* 15.1 (2014), 1929–1958. ISSN: 1532-4435.

[20] Weisong Sun et al. *Abstract Syntax Tree for Programming Language Understanding and Representation: How Far Are We?* 2023. arXiv: 2312.00413 [cs.SE]. URL: https://arxiv.org/abs/2312.00413.

[21] A. Moschou A. Chatzigeorgiou T. Amanatidis N. Mittas, A. Ampatzoglou, and L. Angelis. "Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities". In: *Empirical Software Engineering* 25.5 (2020), pp. 4161–4205. DOI: http://dx.doi.org/10.1007/s10664-020-09869-w.

[22] Petar Veličković et al. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML]. URL: https://arxiv.org/abs/1710.10903.

[23] Fabian Yamaguchi et al. "Modeling and Discovering Vulnerabilities with Code Property Graphs". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 590–604. DOI: 10.1109/SP.2014.44.

[24] Yaqin Zhou et al. "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/49265d2447bc3bbfe9e76306ce40Paper.pdf.

[25] Luka Šikić et al. "Graph Neural Network for Source Code Defect Prediction". In: *IEEE Access* 10 (2022), pp. 10402–10415. DOI: http://dx.doi.org/10.1109/ACCESS.2022.3144598.

A | Additional Data

A.1 Query results in data preprocessing

Table A.1: Top 100 most frequently encountered SonarQube rules from the Technical Debt dataset

RULE	TYPE	RULE_COUNT
squid:S134	CODE_SMELL	62796
squid:S1192	CODE_SMELL	58247
xml:NewlineCheck	CODE_SMELL	41470
code_smells:long_method	CODE_SMELL	41062
squid:S00117	CODE_SMELL	38715
common-java:DuplicatedBlocks	CODE_SMELL	37219
squid:UselessImportCheck	CODE_SMELL	35855
squid:S1166	CODE_SMELL	35064
squid:S00112	CODE_SMELL	31526
squid:S1213	CODE_SMELL	30548
squid:S1135	CODE_SMELL	29190
squid:MethodCyclomaticComplexity	CODE_SMELL	27565
xml:IndentCheck	CODE_SMELL	26106
squid:S00122	CODE_SMELL	25796
squid:CommentedOutCodeLine	CODE_SMELL	22781
code_smells:complex_class	CODE_SMELL	20959
squid:S00116	CODE_SMELL	20927
squid:S1066	CODE_SMELL	17852
squid:S1244	BUG	17608
squid:S1132	CODE_SMELL	17336
squid:ModifiersOrderCheck	CODE_SMELL	17003
squid:S106	CODE_SMELL	14979
squid:RedundantThrowsDeclarationCheck	CODE_SMELL	13944
squid:S1151	CODE_SMELL	13500
code_smells:long_parameter_list	CODE_SMELL	13368
squid:S1149	CODE_SMELL	11357
squid:S1186	CODE_SMELL	11336
squid:S1226	CODE_SMELL	11213

RULE	TYPE	RULE_COUNT
squid:ClassVariableVisibilityCheck	VULNERABILITY	10733
squid:S00105	CODE_SMELL	10144
squid:S1199	CODE_SMELL	9873
squid:S1125	CODE_SMELL	8935
squid:S1481	CODE_SMELL	8605
squid:S1197	CODE_SMELL	8230
squid:S00100	CODE_SMELL	6606
squid:S1068	CODE_SMELL	6359
squid:S1172	CODE_SMELL	6154
code_smells:lazy_class	CODE_SMELL	5781
squid:S00115	CODE_SMELL	5324
squid:S1312	CODE_SMELL	5253
Web:S1827	CODE_SMELL	5228
squid:HiddenFieldCheck	CODE_SMELL	5120
css:one-declaration-per-line	CODE_SMELL	5107
squid:S1161	CODE_SMELL	5104
squid:S1133	CODE_SMELL	5097
squid:S00108	CODE_SMELL	4930
squid:S1181	CODE_SMELL	4908
squid:S1067	CODE_SMELL	4861
squid:S00101	CODE_SMELL	4549
squid:S1313	VULNERABILITY	4328
squid:S1148	VULNERABILITY	4260
squid:S1488	CODE_SMELL	4249
squid:MissingDeprecatedCheck	CODE_SMELL	4233
squid:S1193	CODE_SMELL	4179
squid:S1168	CODE_SMELL	4162
squid:S1126	CODE_SMELL	4012
squid:S1301	CODE_SMELL	3958
squid:S1118	CODE_SMELL	3911
squid:S1155	CODE_SMELL	3848
squid:S1444	VULNERABILITY	3777
squid:S1145	BUG	3407
javascript:Semicolon	CODE_SMELL	3192
squid:S1319	CODE_SMELL	2795
code_smells:class_data_private	CODE_SMELL	2495
squid:EmptyStatementUsageCheck	CODE_SMELL	2473
code_smells:antisingleton	CODE_SMELL	2371
squid:S1160	CODE_SMELL	2327
squid:SwitchLastCaseIsDefaultCheck	CODE_SMELL	2271
squid:S1141	CODE_SMELL	2195
squid:UnusedPrivateMethod	CODE_SMELL	2195
javascript:TrailingWhitespace	CODE_SMELL	2133
squid:S00107	CODE_SMELL	2115
squid:S2250	CODE_SMELL	2094

RULE	TYPE	RULE_COUNT
squid:S1905	CODE_SMELL	2081
squid:S135	CODE_SMELL	2016
css:experimental-property-usage	CODE_SMELL	1888
squid:S1134	CODE_SMELL	1796
css:selector-naming-convention	CODE_SMELL	1738
Web:UnsupportedTagsInHtml5Check	BUG	1716
squid:S1452	CODE_SMELL	1691
squid:S2131	CODE_SMELL	1662
squid:S1188	CODE_SMELL	1633
squid:S1596	CODE_SMELL	1630
squid:S2130	CODE_SMELL	1611
squid:S1170	CODE_SMELL	1599
squid:AssignmentInSubExpressionCheck	CODE_SMELL	1458
css:leading-zeros	CODE_SMELL	1353
squid:S2386	VULNERABILITY	1321
xml:IllegalTabCheck	CODE_SMELL	1279
squid:S1948	BUG	1250
squid:S1598	CODE_SMELL	1238
javascript:EqEqEq	CODE_SMELL	1212
squid:S1214	CODE_SMELL	1191
squid:S128	CODE_SMELL	1184
squid:ForLoopCounterChangedCheck	CODE_SMELL	1164
squid:S2157	BUG	1044
squid:S1185	CODE_SMELL	991
squid:LabelsShouldNotBeUsedCheck	CODE_SMELL	954
css:semicolon-declaration	CODE_SMELL	934
Web:BoldAndItalicTagsCheck	BUG	929

```
PROJECT_ID,GIT_LINK,FAULT_INDUCING_COMMIT_HASH,
   FAULT_FIXING_COMMIT_HASH,RULE,COMPONENT
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/AbstractSVGFilterPrimitiveElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/AbstractSVGGradientElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/CSSUtilities.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/ConcreteGVTBuilder.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/PaintServer.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/RepaintManager.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGAbstractFilterPrimitiveElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGAltGlyphElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGCircleElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
```

```
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGEllipseElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGFeColorMatrixElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGFeComponentTransferElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGFeCompositeElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGFeConvolveMatrixElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGFeDiffuseLightingElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGFeDisplacementMapElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGFeFloodElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGFeGaussianBlurElementBridge.java
org.apache:batik,https://github.com/apache/batik,
   badaead1c2b9f35970146994a936a0920f3ab2d6,231
   c435ba7354d99ab41a02e9a421a6dcb41a409,code_smells:
   long_parameter_list,org.apache:batik:sources/org/apache/
   batik/bridge/SVGFeImageElementBridge.java
```

Listing A.1: SQL Results for Query 3.3

A.2 Lists of SonarQube Rules

Table A.2: Selected Rules from SonarQube Analysis

Rule Name	**Description**
Cyclomatic Complexity (MethodCyclomaticComplexity)	This metric measures the complexity of functions by counting the number of linearly independent paths through a program's source code. In graph representations like Control Flow Graphs (CFGs), a high cyclomatic complexity indicates multiple paths and decision points, suggesting areas that may be error-prone or difficult to maintain. It helps in identifying functions that need simplification or refactoring to reduce technical debt.
Class Data Should Be Private (class_data_private)	Enforcing data encapsulation is crucial in object-oriented programming to prevent external classes from depending on internal implementation details. In an Abstract Syntax Tree (AST), private class data ensures that class interfaces are used correctly, reducing the risk of bugs and making the codebase easier to refactor, thus reducing technical debt.
Duplicated Blocks (DuplicatedBlocks)	This rule identifies blocks of code that are identical or very similar across the codebase. Detectable through graph representations as subgraphs that are isomorphic, indicating redundant code that increases maintenance effort and potential for bugs, contributing significantly to technical debt.
Long Method (long_method)	Long methods often try to do too much, making them difficult to understand, test, and maintain. In an AST, these methods manifest as nodes with high complexity and depth, indicating potential refactoring candidates to break down into simpler, more manageable functions, thus reducing technical debt.
Complex Class (complex_class)	A class deemed too complex may have multiple responsibilities or excessive functionality, which complicates the maintenance and scalability of software. Graph analyses like Class Dependency Graphs can highlight complex classes by showing dense connections and interactions with other classes, pinpointing areas where simplification can reduce technical debt.

Rule Name	Description
Unused Private Method (UnusedPrivateMethod)	Private methods that are not called reflect dead code that unnecessarily complicates the codebase. In Call Graphs, these methods appear as isolated nodes, which can be candidates for removal to simplify the code structure and decrease technical debt.
Base Class Should Not Use Derived Class Functions (S2157)	This rule highlights a reversal in the intended direction of dependency in inheritance hierarchies. Such design flaws can be visualised in a Class Inheritance Graph, where base classes should not have direct dependencies on derived classes. Correcting this reduces complexity and potential errors, thereby cutting down on technical debt.
Method With Boolean Parameter (S1126)	Boolean parameters in methods often suggest that the method performs different functions based on the boolean value, which can lead to complex and confusing code. Such methods, when represented in a CFG, show branching based on boolean values, indicating a need for method separation to improve modularity and reduce technical debt.
Redundant Throws Declaration (Redundant-ThrowsDeclarationCheck)	This rule identifies exceptions in the method signature that are never thrown, which can mislead developers and contribute to unnecessary code complexity. In CFGs, this can be identified by tracing exception handling paths, simplifying these paths can help in reducing technical debt.
Variable Declaration Distance (S1481)	Variables should be declared close to where they are used to improve code readability and maintainability. In ASTs or CFGs, long distances between declaration and usage points can indicate scattered logic, suggesting a restructuring to reduce technical debt.
Modifiers Order (ModifiersOrderCheck)	Standardizing the order of modifiers enhances code readability and reduces the chance of errors. This can be enforced in ASTs, where nodes representing declarations will follow a consistent pattern, aiding in quicker understanding and maintenance of the code, thus reducing technical debt.
Switch Cases Without Break (S128)	Omitting breaks in switch cases can lead to unintentional fall-through, which might introduce bugs. In CFGs, this rule helps ensure each case is properly separated, preventing such errors and reducing technical debt.

Rule Name	Description
Method Should Not Have Too Many Parameters (long_parameter_list)	Methods with a long list of parameters are challenging to understand and use, suggesting poor method design. Such methods can be visualised in ASTs or Call Graphs with complex nodes, indicating the need for refactoring into simpler, more coherent methods, thus reducing technical debt.
Class Variable Visibility Check (ClassVariableVisibilityCheck)	Proper visibility of class variables ensures that encapsulation is maintained, which is crucial for the robustness and flexibility of OOP code. ASTs can reveal inappropriate access levels, guiding refactoring efforts to encapsulate class data properly, reducing technical debt.
Lazy Class (lazy_class)	Classes that are overly large tend to have multiple responsibilities, making them hard to maintain and understand. Graph representations can show these as large nodes with excessive edges, indicating a need for decomposition into smaller, more focused classes, reducing technical debt.

A.3 SQL Queries used in data preprocessing

```
CREATE TABLE manyParameters AS
SELECT PROJECTS.PROJECT_ID, PROJECTS.GIT_LINK,
       SZZ_FAULT_INDUCING_COMMITS.FAULT_INDUCING_COMMIT_HASH,
           SZZ_FAULT_INDUCING_COMMITS.
          FAULT_FIXING_COMMIT_HASH,
       selected_rules.RULE, selected_rules.COMPONENT
FROM PROJECTS
INNER JOIN selected_rules ON PROJECTS.PROJECT_ID =
   selected_rules.PROJECT_ID
INNER JOIN SZZ_FAULT_INDUCING_COMMITS ON PROJECTS.PROJECT_ID
   = SZZ_FAULT_INDUCING_COMMITS.PROJECT_ID
WHERE selected_rules.RULE = 'code_smells:long_parameter_list'
LIMIT 5000;

-- Total Rules
CREATE TABLE manyParametersTotal AS
WITH DistinctFaultInducingCommits AS (
    SELECT *,
           ROW_NUMBER() OVER (PARTITION BY
              FAULT_INDUCING_COMMIT_HASH ORDER BY (SELECT
              NULL)) AS rn
    FROM manyParameters
)
SELECT *
FROM DistinctFaultInducingCommits
WHERE rn = 1;
```

```
select count(*) from manyParametersTotal; -- 1423

-- Training Rules 80%
-- Create `manyParametersTraining` table with the first 1140
   entries from `manyParametersTotal`
CREATE TABLE manyParametersTraining AS
SELECT *
FROM manyParametersTotal
LIMIT 1140;

-- Optionally, count the number of rows in `
   manyParametersTraining`
SELECT COUNT(*) FROM manyParametersTraining; -- Should be
   1140

-- Validation Rules 10%
-- Create `manyParametersValidation` table with 141 elements
   not in `manyParametersTraining`
CREATE TABLE manyParametersValidation AS
SELECT *
FROM manyParametersTotal
EXCEPT
SELECT *
FROM manyParametersTraining
LIMIT 141;

-- Optionally, count the number of rows in `
   manyParametersValidation`
SELECT COUNT(*) FROM manyParametersValidation; -- Should be
   141

-- Evaluation Rules 10%
-- Create `manyParametersEvaluation` table with entries not
   in `manyParametersTraining` or `manyParametersValidation`
CREATE TABLE IF NOT EXISTS manyParametersEvaluation AS
SELECT t.*
FROM manyParametersTotal t
LEFT JOIN manyParametersTraining tr ON t.
   FAULT_INDUCING_COMMIT_HASH = tr.FAULT_INDUCING_COMMIT_HASH
LEFT JOIN manyParametersValidation v ON t.
   FAULT_INDUCING_COMMIT_HASH = v.FAULT_INDUCING_COMMIT_HASH
WHERE tr.FAULT_INDUCING_COMMIT_HASH IS NULL
  AND v.FAULT_INDUCING_COMMIT_HASH IS NULL;

-- count the number of rows in `manyParametersEvaluation`
SELECT COUNT(*) FROM manyParametersEvaluation; --  142
```

Listing A.2: Full query for obtaining a training, validation and evaluation set, with comments included