Master Thesis

Assessment Of Deep Learning Models For Code Quality Evaluation

Marc Dillmann

Matr.No:

Academic Supervisors
Prof. Dr.-Ing. Peter Liggesmeyer
RPTU Kaiserslautern-Landau

Thesis Supervisor:
Dr. Julien Siebert
Fraunhofer IESE

Department of Computer Science
Erwin-Schroedinger-Stasse 1,
67663 Kaiserslautern

Summer Semester 2023

OBLIGATORY SIGNED DECLARATION

I hereby declare that the present master's thesis was composed by myself and that the work contained herein is my own. I also confirm that I have only used the specified resources. All formulations and concepts taken verbatim or in substance from printed or unprinted material or from the Internet have been cited according to the rules of good scientific practice and indicated by footnotes or other exact references to the original source. The present thesis has not been submitted to another university for the award of an academic degree in this form. This thesis has been submitted in printed and electronic form. I hereby confirm that the content of the digital version is the same as in the printed version. I understand that the provision of incorrect information may have legal consequences.

Kaiserslautern, Germany, October 4, 2023

.....

Marc Dillmann

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my academic supervisor, Prof. Dr.-Ing. Peter Liggesmeyer, for accepting and overseeing my thesis.

Special thanks are due to my thesis supervisor, Dr. Julien Siebert, whose consistent assistance, insights, and feedback have been instrumental in shaping this work. I am also deeply grateful to Dr. Adam Trendowicz and Thorsten Honroth for their invaluable assistance and contributions, which have enriched the quality and depth of this research.

I would also like to extend my appreciation to Anna Vollmer. It was her initiative that bridged the connection to the data science department of the Fraunhofer IESE and facilitated the initial subject search, setting the foundation for this thesis. Her efforts in laying the groundwork for this research are deeply acknowledged.

Lastly, I want to thank my family, my girlfriend, and my son, who always supported me in pursuing my master's degree.

To all mentioned, your support and guidance have been the pillars upon which this work stands, and I am truly grateful for your contributions.

Kaiserslautern, Germany, October 4, 2023

.....

Marc Dillmann

Abstract

The evaluation of software code quality, especially maintainability, is a cornerstone in the field of software engineering. As software systems grow in complexity and scale, traditional methods of code quality assessment often fall short, necessitating more advanced and scalable approaches. This thesis delves into the potential of harnessing deep learning models, specifically transformer-based large language models, for the task of maintainability prediction. Through a rigorous literature review, a notable gap was identified regarding the application of deep learning models in this domain, setting the stage for the research's primary objective. The proposed methodology eschews conventional preprocessing or feature extraction, instead leveraging the next-token probability of transformer-based models to compute the cross-entropy between the source code and the model's predictions. This cross-entropy is then utilized as a metric to predict maintainability and its associated sub-characteristics. Empirical results demonstrate the feasibility and competitive performance of this approach with accuracy and F1-Measure reaching up to 0.74 and 0.77 respectively, offering a streamlined alternative to existing methods. This research not only bridges a discernible gap in the literature but also builds upon a novel paradigm in code quality evaluation, potentially catalyzing further advancements in the intersection of deep learning and software engineering.

Contents

Chapter 1

Introduction

1.1 Context

In the realm of software engineering, code quality is paramount. High-quality code not only ensures the robustness and efficiency of software applications but also reduces maintenance costs and fosters a sustainable development environment. Traditionally, code quality evaluation has been a combination of manual reviews and automated tools that rely on predefined rules and heuristics based on static code metrics. However, with the exponential growth of software repositories and the increasing complexity of software systems, there is a pressing need for more sophisticated, scalable, and adaptive approaches to assess code quality.

Deep learning, a subset of machine learning, has shown remarkable success in various domains, from image and speech recognition to natural language processing. Its ability to learn complex patterns from vast amounts of data makes it a promising candidate for the task of code quality evaluation. Given the structured yet complex nature of code, there is potential for deep learning models to capture and understand the nuances that define high-quality code, going beyond the capabilities of traditional rule-based systems.

1.2 Goal and focus of the thesis

This thesis aims to explore the applicability of deep learning for assessing software quality. For this purpose, we execute two major steps. First, we deploy a systematic literature review in the realm of deep learning for software quality prediction. We identify bug detection and vulnerability prediction to be well-researched in this context, whereas maintainability prediction lacks almost any scientific attention. To fill this gap, we set our focus for the experimental phase to find a novel approach to predict maintainability using deep learning. Our second step is the experimental phase, where we investigate a recent hypothesis that states that the next token probability of the code generated by transformer models is correlated with maintainability and some of its sub-characteristics. We test 10 different fine-tuned transformer models based on GPT-2 and Llama 2 and our results show, that our hypothesis can be confirmed. When training a classifier on the cross-entropy values and the assessed maintainability values in an evaluation dataset to determine the relation between them, we reach an accuracy and F1-Measure of up to 0.74 and 0.77 respectively. A wide range in the results for the different models suggests the importance of a careful model selection for the specific task.

Through this investigation, we aim to contribute to the ongoing discourse in the software engineering community, providing a fresh perspective on how emerging technologies like deep learning can be harnessed to address long-standing challenges in the field.

1.3 Structure of the manuscript

This thesis is organized as follows:

- **Chapter 2: Background**
  Here, foundational concepts relevant to the research are discussed. This includes an overview of software quality, an introduction to code metrics, a primer on machine learning and deep learning, and an exploration of code representation and code embedding techniques.

- **Chapter 3: Related Work**
  This chapter delves into the existing literature and methodologies in the domain. It begins with a systematic literature review, detailing the research questions, search strategy, results, and conclusions. The subsequent snowballing technique and its results are also presented.
- **Chapter 4: Experiments**
  The core of the research is presented in this chapter. It starts with an initial procedure, discussing the applicability of existing models for code quality evaluation, the efficacy of transfer learning, and the challenges faced. This leads to the revised procedure, detailing the foundation and our unique approach. The experimental process, implementation details, based models, and model selection criteria are elaborated upon. The results section provides a comprehensive analysis of the deep learning model's performance across various metrics. The chapter concludes with a discussion of the findings.
- **Chapter 5: Conclusion**
  This final chapter wraps up the research, summarizing the key findings and contributions of the study. It also provides directions for future work in this domain.

Chapter 2

Background

As we venture into the domain of code quality evaluation using deep learning models, it is imperative to first establish a foundational understanding of the key concepts and methodologies that underpin this research. This chapter serves as a primer, explaining the fundamental principles of software quality, code metrics, and the overarching paradigms of machine learning and deep learning. Additionally, this chapter delves into the concepts of code representation and embedding.

2.1 Software quality

Software quality, as a multidimensional construct, encompasses a range of attributes that determine the value of a software product to its users. The International Organization for Standardization (ISO) has, over the years, developed standards to provide a structured framework for understanding an evaluating software quality. One of the most prominent among these is the ISO/IEC 25010 standard, which builds upon its predecessors to offer a comprehensive model for software product quality and its evaluation.

According to ISO/IEC 25010 [1], software quality characteristics are defined as follows:

- **Functional Suitability:** The degree to which a product or system provides functions that meet stated and implied

needs when used under specified conditions.

- **Performance Efficiency:** The performance relative to the amount of resources used under stated conditions.
- **Compatibility:** The degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment.
- **Usability:** The degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.
- **Reliability:** The degree to which a system, product or component performs specified functions under specified conditions for a specified period of time.
- **Security:** The degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.
- **Maintainability:** The degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers.
- **Portability:** The degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another.

In essence, the ISO/IEC 25010 standard provides a holistic view of software quality, emphasizing the inherent attributes of the software product. This comprehensive approach ensures that software quality is assessed from multiple perspectives, capturing the full spectrum of factors that contribute to a product's overall quality.

2.2 Code metrics

Code metrics provide a quantitative measure of software characteristics, enabling developers and researchers to assess various facets of software quality. Predominantly derived from static analysis of the source code, these metrics offer insights without necessitating the execution of the software. Since this thesis aims to find ways of predicting code quality without relying on those static code metrics, we only give a short summary of the most widely used static code metrics.

1. **Lines of Code (LOC) [2]:** One of the most basic metrics, it measures the number of lines in a software program. While simple, LOC can give a rough estimate of the program's size and complexity. However, it is worth noting that a higher LOC does not necessarily equate to lower quality or vice versa.
2. **Cyclomatic Complexity (CC) [3]:** Introduced by Thomas McCabe, this metric evaluates the complexity of a program by measuring the number of linearly independent paths through the source code. It provides insights into the testing effort required and the potential risks associated with the code.
3. **Halstead Complexity Measures [4]:** Proposed by Maurice Halstead, these measures evaluate various attributes of a program, including program length, vocabulary, volume, difficulty, and effort. They are derived from the number of operators and operands in the code.
4. **Depth of Inheritance [5]:** This metric measures the depth of a class in the inheritance hierarchy. A deeper inheritance might indicate greater complexity and reduced modularity, potentially making the code harder to maintain.
5. **Number of Children [5]:** It quantifies the number of immediate subclasses a particular class has. A higher number might indicate that a class has been heavily reused, but it might also suggest that the class is taking on too many responsibilities.
6. **Coupling Between Objects (CBO) [5]:** This metric evaluates the number of classes that a particular class is coupled to. High coupling

can reduce the modularity of the code and make it more prone to error during changes.

7. **Response For a Class (RFC) [5]:** It measures the number of unique methods that can be executed in response to a message received by an object of that class. A higher RFC indicates greater complexity.
8. **Lack of Cohesion in Methods (LCOM) [5]:** This metric assesses the cohesion of methods in a class. A higher LCOM value might suggest that a class is trying to perform too many operations, hinting at a potential need for refactoring.

2.3 Machine learning and Deep Learning

2.3.1 Machine Learning Overview

Machine Learning (ML) [6], a branch of Artificial Intelligence (AI), empowers systems to autonomously learn and enhance their performance from experience, without the need for explicit programming. This learning mechanism relies on identifying complex patterns within data and making informed decisions based on these insights.

There are three main types of ML:

- **Supervised Learning:** The algorithm is trained on a labeled dataset, which means that each training example is paired with an output label. The algorithm makes predictions or classifications based on the input data and is corrected when its predictions are incorrect.
- **Unsupervised Learning:** In this approach, the algorithm is presented with data without clear directives on how to process it. The system endeavors to discern patterns and structures within the data without the guidance of labeled responses.
- **Reinforcement Learning:** This learning method involves agents that operate within an environment to accumulate rewards. The agent's learning is driven by actions that optimize this cumulative reward, enabling the algorithm to be trained to make predictions that align with specific criteria or principles.

2.3.2 Deep Learning Overview

Deep Learning (DL) [7] is a specialized subset of ML. Deep learning algorithms are based on artificial neural networks, particularly deep neural networks (DNNs), visualized in figure 2.1. A DNN has more than one hidden layer between its input and output, allowing it to learn more complex patterns and representations.

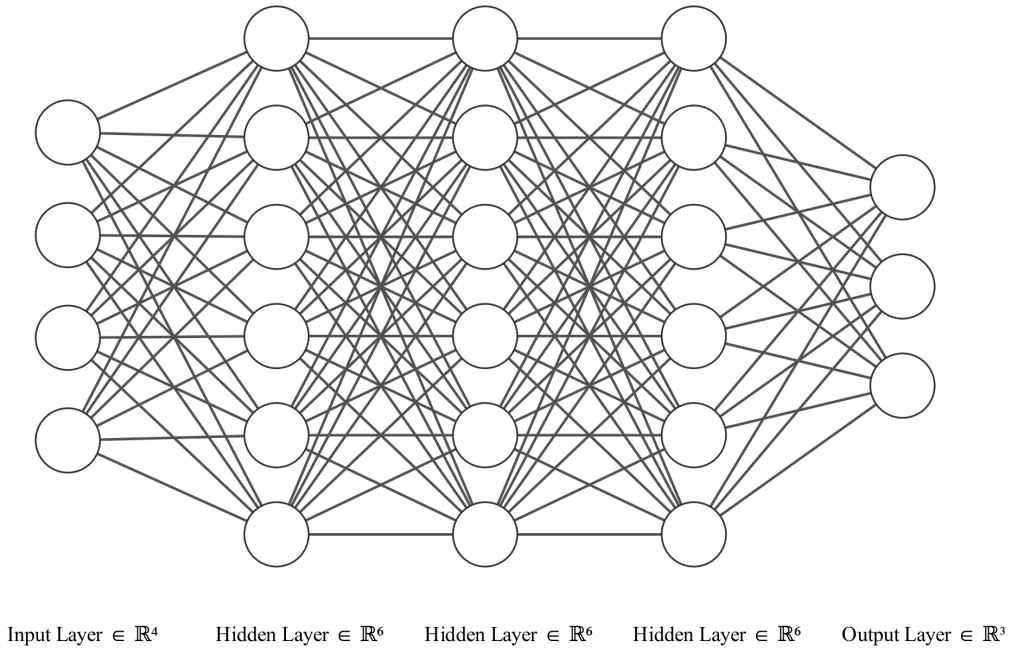

Figure 2.1: Visual representation of a deep neural network (DNN).

The main components of deep learning include:

- **Neurons:** The basic unit or nodes of a neural network, inspired by the neurons in the human brain.
- **Layers:** Neurons are organized in layers. There are typically three types of layers: input layer, hidden layers, and output layer.
- **Activation Function:** This function defines the output of a neuron given a set of inputs, introducing non-linear properties to the system.

- **Weights and Biases:** These are the parameters of the model, adjusted during the training process to minimize the error between the predicted and actual output.

2.3.3 Distinction Between Machine Learning and Deep Learning

While Deep Learning is a subset of Machine Learning, the distinction between the two lies in their capabilities and applications. ML algorithms are often designed for specific tasks and require feature engineering and data preprocessing to work effectively. In contrast, DL algorithms, particularly DNNs, are capable of automatically extracting features from raw data, making them more versatile and applicable to a broader range of problems, including image and speech recognition, natural language processing, and, as explored in this thesis, code quality evaluation.

In the context of this research, the focus is primarily on the application and evaluation of deep learning models due to their ability to handle large datasets and automatically learn features from raw data, providing a robust framework for the assessment of code quality. The subsequent section will delve deeper into the specifics of the most important deep learning models and architectures that are examined in the literature review and experiment parts of this thesis.

2.3.4 Deep Learning Models

Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNNs) [8] represent a class of deep learning models that have proven to be highly effective in tasks related to spatial hierarchies or topologies, most notably in image and video recognition. CNNs are specifically designed to recognize patterns from the spatial distribution of data. A visualization of AlexNet [9], a CNN defined in one of the most influential papers in computer vision with over 120000 citations, can be found in figure 2.2.

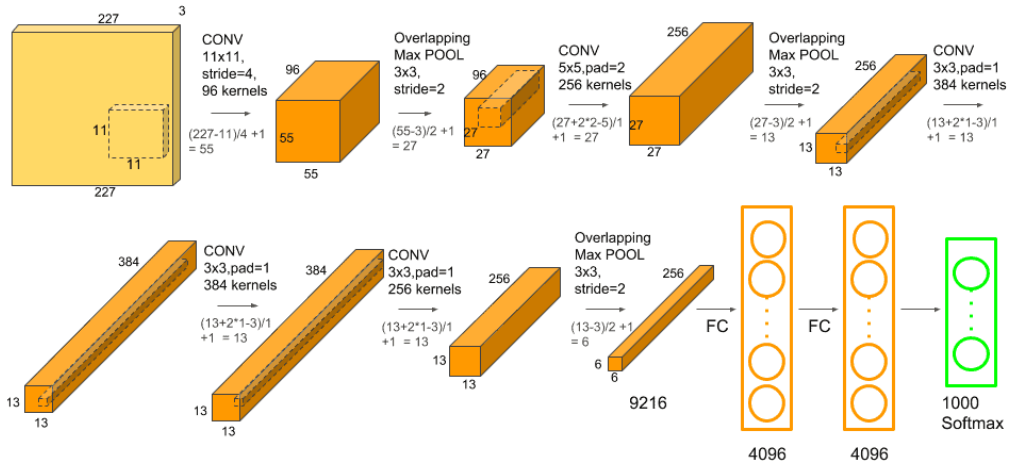

Figure 2.2: Visual representation of the AlexNet CNN.
Source: `https://learnopencv.com/understanding-alexnet/`

The architecture of a CNN is distinctively organized to automatically and adaptively learn spatial hierarchies of features from input images. The primary components of CNNS include:

- **Convolutional Layer:** At the heart of a CNN lies the convolutional layer. This layer is characterized by its learnable filters (or kernels) that possess a small receptive field, yet span the entire depth of the input. In the forward pass, these filters slide over the input's width and height, resulting in a 2D activation map. Essentially, the network is designed to learn filters that become active when they identify certain features at specific locations in the input.
- **Pooling Layer:** Often inserted between successive convolutional layers, pooling layers reduce the spatial dimensions of the representation, reducing the number of parameters and computation in the network. Max-pooling is the most common strategy, where each unit in the pooling layer outputs the maximum of a region in the input.

- **Fully Connected Layer:** Neurons in a fully connected layer have connections to all activations in the previous layer. These layers can be seen as traditional multi-layer perceptrons that use a softmax activation function for classification.
- **ReLU (Rectified Linear Unit) Activation Function:** After each convolution operation, the result is passed through an activation function. ReLU is the most commonly used activation function in CNNs, introducing non-linearity into the model.

CNNs are trained using backpropagation, similar to other neural networks. However, due to their hierarchical architecture, CNNs are particularly adept at learning spatial hierarchies of features. In the initial layers, the network might learn to recognize edges, colors, and textures. In deeper layers, the network can recognize more complex patterns, like shapes or specific objects.

While CNNs were initially designed for image classification tasks, their application has since expanded to a wide range of computer vision tasks, including:

- Image and video recognition
- Image segmentation
- Object detection
- Facial recognition

CNNs have revolutionized the field of computer vision, achieving state-of-the-art performance[1]. Their ability to automatically learn and hierarchically represent features in spatial data makes them a powerful tool for a wide range of applications, both within and outside the realm of image processing.

[1] `https://paperswithcode.com/sota/image-classification-on-imagenet`

LSTM

Long Short-Term Memory (LSTM) networks, a type of RNN, were introduced by Hochreiter and Schmidhuber [10] in 1997. Designed to address the limitations of traditional RNNs, particularly the vanishing and exploding gradient problems, LSTMs have since become a pivotal architecture in sequence modeling tasks, especially where long-term dependencies exist.

LSTM networks are characterized by their unique cell structures, which are designed to regulate the flow of information through the network. The primary of an LSTM cell include:

- **Forget Gate:** This component decides what information from the cell state should be thrown away or kept. It uses a sigmoid activation function to generate values between 0 (forget) and 1 (keep).
- **Input Gate:** It updates the cell state with new information. The input gate has two parts: a sigmoid layer that decides which values to update and a tanh layer that creates a vector of new candidate values.
- **Cell State:** Often denoted a C_t, it represents the "memory" of the LSTM unit. It can store long-term information and is regulated by the forget and input gates.
- **Output Gate:** Based on the cell state and the input, it decides what the next hidden state h_t should be. This hidden state can be used for predictions and is passed to the next LSTM cell.

One of the primary motivations behind the development of LSTMs was to combat the vanishing gradient problem encountered in traditional RNNs. This issue arises when gradients of the loss function become too small for effective training, leading to long training times and poor performance on long sequences. LSTMs, with their gated structure, can selectively remember or forget information, making them adept at learning and maintaining long-term dependencies in data.

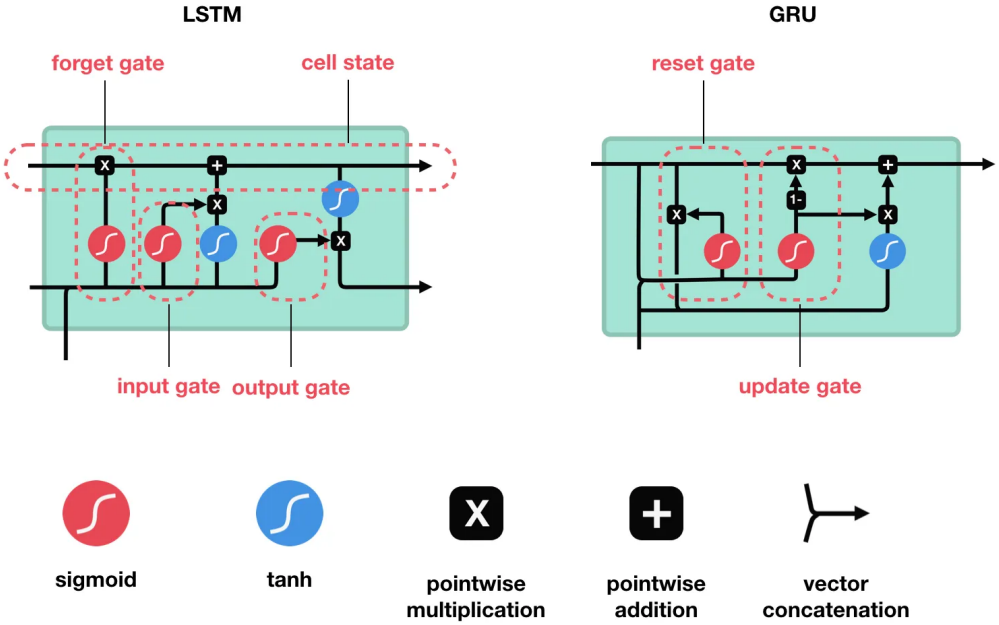

Figure 2.3: Visual representation of an LSTM (left) and GRU (right).
Source: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

LSTMs have found applications across a wide range of domains, including:

- Natural language processing tasks like machine translation, sentiment analysis, and text generation.
- Time series forecasting in finance, weather prediction, and energy consumption modeling.
- Speech recognition and music generation.
- Video activity recognition

Over the years, several variants and extensions of LSTMs have been proposed to enhance their capabilities or address specific challenges:

- **Bidirectional LSTMs [11]:** Process data from past to future and vice versa, making them effective for tasks where context from both directions is crucial.
- **Gated Recurrent Units (GRUs) [12]:** A simplified version of LSTMs with fewer gates but similar performance characteristics for certain tasks.

LSTM networks, with their ability to capture long-term dependencies and mitigate the challenges of traditional RNNs, have solidified their position as a go-to architecture for sequence modeling tasks. Their adaptability, coupled with their robust performance across diverse domains, underscores their significance in the ever-evolving landscape of deep learning.

Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNNs) [13] are a class of neural networks designed to recognize patterns in sequences of data, such as time series or natural language. Unlike traditional feedforward neural networks, RNNs possess connections that loop backward, allowing them to maintain a "memory" of previous inputs as their internal state. This characteristic makes them particularly suited for tasks where temporal dynamics and context from earlier inputs are essential.

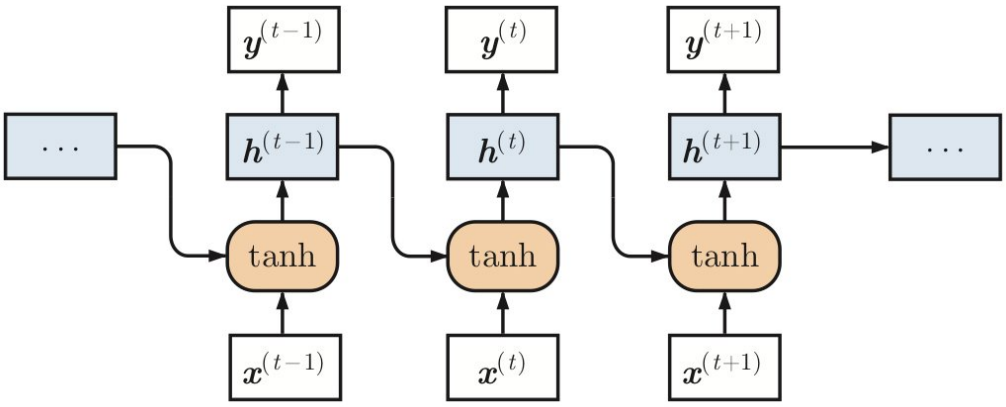

Figure 2.4: Visual representation of a RNN.
Source: https://blog.acolyer.org/2019/02/25/understanding-hidden-memories-of-recurrent-neural-networks/

The fundamental architecture of an RNN is characterized by the repetition of a single type of network module across different time steps of a sequence. The primary components of an RNN include:

- **Hidden State:** This represents the "memory" of the RNN. At each time step, the hidden state is updated based on the previous hidden state and the current input. This updated state can be used for predictions and is also passed to the next time step.
- **Weights:** Unlike traditional feedforward neural networks where weights might differ across layers, in an RNN, the weights are shared across time steps, reflecting the network's recurrent structure.
- **Activation Function:** Typically, a non-linear activation function like the hyperbolic tangent (tanh) is used to compute the hidden state.

While RNNs are theoretically capable of handling long-term dependencies, they often struggle in practice due to the following challenges:

- **Vanishing Gradient Problem:** As the network becomes deeper (i.e., processes longer sequences), gradients during backpropagation can become extremely small, causing the network to learn very slowly or not at all. This makes it difficult for the RNN to capture long-term dependencies in the data.
- **Exploding Gradient Problem:** Conversely, gradients can also become too large, causing the model's weights to update in extreme ways, leading to instability in learning.

RNNs have been employed in a myriad of applications, including:

- Natural language processing tasks such as text generation, machine translation, and sentiment analysis.
- Time series forecasting for stock prices, weather patterns, and energy consumption.
- Speech recognition and synthesis.
- Gesture recognition in videos.

Given the challenges associated with RNNs, several variants have been developed to enhance their capabilities:

- **Long Short-Term Memory (LSTM):** Designed to combat the vanishing gradient problem, LSTMs introduce a more complex recurrent unit with gating mechanisms to regulate the flow of information.

- **Gated Recurrent Units (GRUs):** A simplified version of LSTMs that combines the forget and input gates into a single "update gate".

Recurrent Neural Networks, with their inherent ability to process sequences and maintain contextual information, have paved the way for advancements in various domains dealing with sequential data. While they have intrinsic challenges, the development of variants like LSTMs and GRUs has expanded their applicability, making them a foundational model in the domain of deep learning for sequences.

Transformer

The Transformer architecture, introduced in the paper "Attention Is All You Need" by Vaswani et al. [14] in 2017, has since become a cornerstone in the realm of natural language processing (NLP). Distinct from traditional recurrent neural networks (RNNs) and convolutional neural networks (CNNs) used in sequence modeling tasks, the Transformer leverages attention mechanisms to draw global dependencies between input and output, offering significant improvements in terms of both efficiency and performance.

The Transformer architecture is primarily composed of an encoder and a decoder, each consisting of multiple identical layers, see figure 2.5. The salient components of the Transformer include:

- **Self-Attention Mechanism:** At its core, the Transformer utilizes a self-attention mechanism that weighs input elements differently, allowing the model to focus on different words in a sequence when producing an output. This mechanism enables the model to consider other words in the sequence, irrespective of their distance, making it highly effective for capturing long-range dependencies.
- **Multi-Head Attention:** Instead of having a single set of attention weights, the Transformer employs multiple sets, enabling the model to focus on different positions in the input simultaneously. This multi-faceted attention mechanism allows the model to capture a richer combination of features.

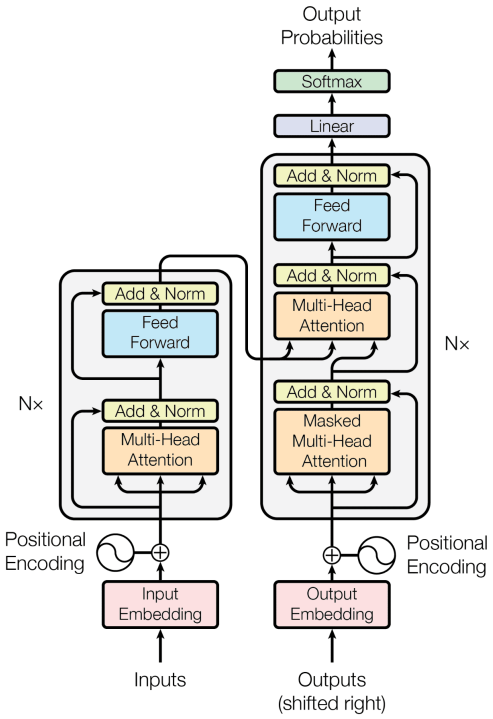

Figure 2.5: Visual representation of the transformer architecture [14].

- **Positional Encoding:** Since the Transformer lacks inherent sequential processing like RNNs, it uses positional encodings to consider the position of words in a sequence. These encodings are added to the embeddings at the base of the model, providing positional context to the attention mechanism.
- **Feed-foward Neural Networks:** Each Transformer layer contains a fully connected feed-forward network, which is applied independently to each position.
- **Normalization and Residual Connections:** To stabilize the activations in the network, layer normalization and residual connections are employed throughout the architecture.

The Transformer is trained using standard backpropagation techniques. However, due to its parallel processing capabilities (unlike the sequential nature of RNNs), it offers faster training times, especially on hard accelerators like GPUs.

While the Transformer was initially designed for machine translation tasks, its versatility has led to its adoption in a myriad of NLP tasks[2], including:

- Text summarization
- Question Answering
- Sentiment analysis
- Named entity recognition

Furthermore, the Transformer architecture has paved the way for several state-of-the-art models like BERT (Bidirectional Encoder Representations from Transformer) [15], GPT[3] (Generative Pre-trained Transformer), and T5 (Text-to-Text Transfer Transformer) [16], which have further pushed the boundaries of performance in various NLP benchmarks.

The Transformer architecture, with its innovative attention mechanisms and parallel processing capabilities, has redefined the landscape of sequence modeling and NLP. Its ability to capture intricate patterns and dependencies in data, combined with its scalability, makes it a foundational model for contemporary NLP research and applications.

2.4 Code Representation and Embedding

In the realm of software engineering and machine learning, representing source code in a manner that facilitates analysis, comparison, and prediction is paramount. Code representation is the process of converting raw source code into a structured format or embedding that can be processed by

[2] https://huggingface.co/models

[3] https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf

algorithms, especially machine learning models. This transformation is crucial for tasks like code quality evaluation, bug detection, and code completion.

Historically, several models have been employed to represent code:

- **Abstact Syntax Tree (AST) [17]:** An AST represents the hierarchical structure of source code. Each node in the tree corresponds to a construct in the source code, capturing the syntactic structure of the program without considering its specific syntax.
- **Source Code Metrics:** These are quantitative measures derived from the static analysis of the source code, such as Lines of Code (LOC), Cyclomatic Complexity, and Depth of Inheritance. While they provide a high-level overview, they might miss intricate patterns in the code.
- **Control Flow Graphs (CFG) [18]:** A CFG represents the flow of a program during its execution, with nodes being basic blocks of code and edges indicating the flow between these blocks.

With the advent of deep learning, embedding techniques have been developed to convert source code into dense vector representations:

- **Word Embeddings:** Techniques like Word2Vec [19] or FastText [20], initially designed for natural language processing, can be adapted to treat code tokens as "words" and generate embeddings that capture semantic relationships between code elements.
- **Graph Embeddings [21]:** Given that code can be represented as graphs (e.g. ASTs or CFGs), graph embedding techniques can be employed to convert these graphs into dense vector representations.
- **Sequence-to-Sequence Models [22]:** Encode-decoder architectures, especially those based on LSTMs or Transformers, can be used to encode source code into a fixed-size vector representation.

Unlike natural language, where semantics often rely on the arrangement of words, code semantics are heavily influenced by both local and global contexts. For instance, the meaning and impact of a variable can change based on its scope, usage, and the overall structure of the code. Hence, effective code representations must capture both the immediate context (e.g., neighboring tokens or statements) and the broader context (e.g., function or module structure).

Effective code representations find applications in various domains:

- **Code Quality Assessment:** By capturing the structural and semantic essence of the code, these representations can be used to predict code quality metrics or detect potential defects.
- **Code Search and Retrieval:** Dense vector representations allow for efficient similarity-based search, enabling developers to find relevant code snippets or modules.
- **Code Completion and Generation:** Embeddings can be used as inputs to models that assist developers by auto-completing code or generating code snippets based on specific requirements.

Code representation is a foundational step in the intersection of software engineering and machine learning. By converting raw source code into structured or dense formats, a plethora of applications is enabled that can assist developers, improve code quality, and streamline software development processes.

Chapter 3

Related Work

This chapter provides a comprehensive survey of the literature relevant to our study. The aim is to establish a robust understanding of the state-of-the-art deep learning models for code quality evaluation, and to identify gaps in the existing body of knowledge that our research seeks to fill.

To ensure a systematic and thorough review, we adopt a two-pronged approach: conducting a Systematic Literature Review (SLR) followed by a Snowballing method.

The Systematic Literature Review is a structured method designed to capture as much relevant academic literature on a particular topic as possible. It involves formulating clear research questions, defining explicit inclusion and exclusion criteria, systematically searching for literature that meets these criteria, and critically appraising the selected studies. Through this process, we aim to provide a comprehensive overview of the existing research on the application of deep learning models in code quality evaluation.

In the course of our systematic review, we identified a significant gap in the literature: the application of deep learning models in assessing code maintainability appears to be significantly less researched than the application to other aspects of code quality. To further investigate this gap, we apply the Snowballing method, a complementary approach to the systematic review, allowing for a more detailed examination of this specific domain.

Snowballing involves starting with a set of core papers and then "snow-

balling" outwards by looking at the references of these papers and the papers that cite them. This method helps us to uncover more literature relevant to our specific focus on code maintainability that might have been missed in the systematic review.

In the following sections, we will delve into the findings from both the Systematic Literature Review and the Snowballing process, setting the foundation for our subsequent research.

3.1 Systematic Literature Review

In this section, we will elaborate on our research questions, search strategy, selection criteria, and data extraction, guided by Kitchenham and Charters [23]. These elements collectively constitute the backbone of our SLR and will guide our exploration of the existing body of literature in this field.

3.1.1 Research Questions

Research questions serve as compasses, guiding the exploration and analysis of existing scholarly works. They encapsulate the specific areas of inquiry, allowing researchers to critically evaluate the literature, identify patterns, and synthesize knowledge to address gaps and contribute to the existing body of knowledge.

This SLR aims to provide an overview of deep learning models that evaluate the quality of code without relying on traditional code metrics as input. Consequently, the following research questions (RQ1 to RQ3) were identified:

- RQ1: Which code quality characteristics can be predicted utilizing deep learning models without using traditional code metrics as input?
- RQ2: Which deep learning models have been proposed to evaluate each of those code quality characteristics?
- RQ3: How well do the models perform in predicting the code quality characteristics?

3.1.2 Search strategy

This section will outline the specifics of our search strategy, including the selection of the database, the formulation of the search query using relevant keywords and boolean operators, and the inclusion and exclusion criteria we have established to filter the search results. This transparent and systematic approach aims to ensure the repeatability of the search, which is a key principle of a robust Systematic Literature Review.

Scopus[1] was chosen as the database to be searched for this literature review due to several reasons. Firstly, Scopus is a comprehensive and multidisciplinary database that covers a wide range of academic disciplines, making it suitable for a thorough examination of the literature. Secondly, as this review is conducted within the scope of a master thesis, focusing on a single database helps to manage the extent of this study, ensuring a manageable workload and allowing for a more in-depth analysis of the selected articles. By utilizing Scopus, we aim to maintain a balanced approach while ensuring the adequacy of the literature coverage within the constraints of the research project.

The following relevant keywords were identified to be combined in the search query:

Category	**Identified Keywords**
Deep Learning	Deep Learning, LSTM, CNN, RNN
Code Quality	Code, Quality, Compatibility, Usability, Reliability, Security, Maintainability, Portability, Bug, Fault
Task	Prediction, Assessment, Evaluation

Table 3.1: Identified Keywords ordered by Category.

[1] `https://www.scopus.com`

Using these keywords and boolean operators, the following search string was formulated:

TITLE-ABS (("Deep Learning" OR "LSTM" OR "CNN" OR "RNN") AND "Code" AND ("Quality" OR "Compatibility" OR "Usability" OR "Reliability" OR "Security" OR "Maintainability" OR "Portability" OR "Bug" OR "Fault") AND ("Prediction" OR "Assessment" OR "Evaluation"))

Based on this search string, the search was conducted on 11.05.2023 on Scopus, yielding an initial number of 739 papers.

The inclusion and exclusion criteria were established to ensure the selection of relevant studies that align with the research objectives, focusing on specific parameters such as publication date, language, and type of approach.

The initial collection of papers was filtered and narrowed down by applying the specific inclusion and exclusion criteria, reducing the 739 to 45 papers. The different filtering stages with the number of papers for each stage can be found in figure 3.1 and the titles and references of the publications composing the final collection can be found in table 3.3.

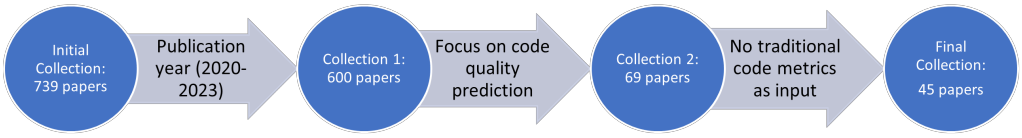

Figure 3.1: Overview of the different filtering stages: Filtering the initial collection by publication year, then by the focus on code quality prediction, and finally by the exclusion of code metrics as input. (Note: Three papers were additionally excluded in the second filtering stage because they were not obtainable.)

Inclusion	Exclusion
• Papers that are published in the last 4 years (2020-2023) • Papers that focus on deep learning models • Papers that focus on Code Quality Prediction • Approaches that directly use text or a direct representation of the code like AST • Papers that present an empirical evaluation of the presented deep learning model in the context of code quality evaluation/prediction • Papers that provide an implementation, either publicly available or defined by framework and configuration • Papers that are peer-reviewed • Papers that are written in English	• Papers that were published more than 4 years ago (2019 and earlier) • Papers that solely use traditional machine learning (non-deep learning) or other non-ML techniques • Approaches that use static code metrics as input • Papers that do not present an empirical evaluation • Papers that focus only on theoretical aspects or mathematical models without any practical application • Papers that are opinion pieces, editorials, or non-peer-reviewed • Papers that are not written in English

Table 3.2: Inclusion and Exclusion criteria

3.1.3 Results

Having executed our search strategy, this section presents the results of our Systematic Literature Review. The results represent the collection of studies that fulfill our predefined criteria, offering insight into the current state of research on the application of deep learning models for code quality evaluation.

The results are organized by the three central research questions that guided our review. We will first delve into which code quality characteristics can be predicted by deep learning without the inclusion of traditional code metrics in the input data (RQ1). We will then proceed to detail the various deep learning models that have been put forward to evaluate these code quality characteristics (RQ2). Lastly, we will evaluate the effectiveness of

Identifier	Title	Reference
P1	Multi-graph learning-based software defect location	[24]
P2	Defect Prediction via Tree-Based Encoding with Hybrid Granularity for Software Sustainability	[25]
P3	Combining AST Segmentation and Deep Semantic Extraction for Function Level Vulnerability Detection	[26]
P4	DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction	[27]
P5	Deep Just-In-Time Defect Localization	[28]
P6	A transformer-based IDE plugin for vulnerability detection	[29]
P7	Cross-project defect prediction based on G-LSTM model	[30]
P8	Path-sensitive code embedding via contrastive learning for software vulnerability detection	[31]
P9	Context-Aware Code Change Embedding for Better Patch Correctness Assessment	[32]
P10	VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python	[33]
P11	DreamLoc: A Deep Relevance Matching-Based Framework for bug Localization	[34]
P12	Visualization-Based Software Defect Prediction via Convolutional Neural Network with Global Self-Attention	[35]
P13	Convolutional Neural Network for Software Vulnerability Detection	[36]
P14	Early Experience with Transformer-Based Similarity Analysis for DataRaceBench	[37]
P15	Within-Project Defect Prediction Using Improved CNN Model via Extracting the Source Code Features	[38]
P16	A Software Aging-Related Bug Prediction Framework Based on Deep Learning and Weakly Supervised Oversampling	[39]
P17	Software Defect Prediction using Deep Learning by Correlation Clustering of Testing Metrics	[40]
P18	BERT-Based Vulnerability Type Identification with Effective Program Representation	[41]
P19	Feature Envy Detection with Deep Learning and Snapshot Ensemble	[42]
P20	VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection	[43]
P21	Multi-Label Code Smell Detection with Hybrid Model based on Deep Learning	[44]
P22	Correlation Feature Mining Model Based on Dual Attention for Feature Envy Detection	[45]
P23	Can Deep Learning Models Learn the Vulnerable Patterns for Vulnerability Detection?	[46]
P24	Investigating Graph Embedding Methods for Cross-Platform Binary Code Similarity Detection	[47]
P25	Fast Changeset-based Bug Localization with BERT	[48]
P26	Compiler IR-Based Program Encoding Method for Software Defect Prediction	[49]
P27	Web Service Anti-patterns Prediction Using LSTM with Varying Embedding Sizes	[50]
P28	CD-VulD: Cross-Domain Vulnerability Discovery Based on Deep Domain Adaptation	[51]
P29	A context-aware neural embedding for function-level vulnerability detection	[52]
P30	TokenCheck: Towards Deep Learning Based Security Vulnerability Detection in ERC-20 Tokens	[53]
P31	A novel software defect prediction method based on hierarchical neural network	[54]
P32	Deep Transfer Bug Localization	[55]
P33	AutoVAS: An automated vulnerability analysis system with a deep learning approach	[56]
P34	Asteria: Deep Learning-based AST-Encoding for Cross-platform Binary Code Similarity Detection	[57]
P35	DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network	[58]
P36	A Novel Tree-based Neural Network for Android Code Smells Detection	[59]
P37	DeepLaBB: A Deep Learning Framework for Blocking Bugs	[60]
P38	Vulmg: A Static Detection Solution for Source Code Vulnerabilities Based on Code Property Graph and Graph Attention Network	[61]
P39	Software defect prediction using hybrid model (CBIL) of convolutional neural network (CNN) and bidirectional long short-term memory (Bi-LSTM)	[62]
P40	Deep Just-In-Time Inconsistency Detection Between Comments and Source Code	[63]
P41	Combining Graph-Based Learning with Automated Data Collection for Code Vulnerability Detection	[64]
P42	Deep Semantic Feature Learning for Software Defect Prediction	[65]
P43	Software defect prediction via LSTM	[66]
P44	PathPair2Vec: An AST path pair-based code representation method for defect prediction	[67]
P45	Transfer Learning Code Vectorizer based Machine Learning Models for Software Defect Prediction	[68]

Table 3.3: Identified publications in the final collection.

these models, examining how well they perform in predicting code quality characteristics (RQ3).

This section will detail the number and nature of studies found, the data extracted from these studies, and a synthesis of their findings. It is important to note that in presenting these results, we aim to offer a detailed overview of the present body of literature and its implications for our study.

RQ1: Which code quality characteristics can be predicted utilizing deep learning models without using traditional code metrics as input?

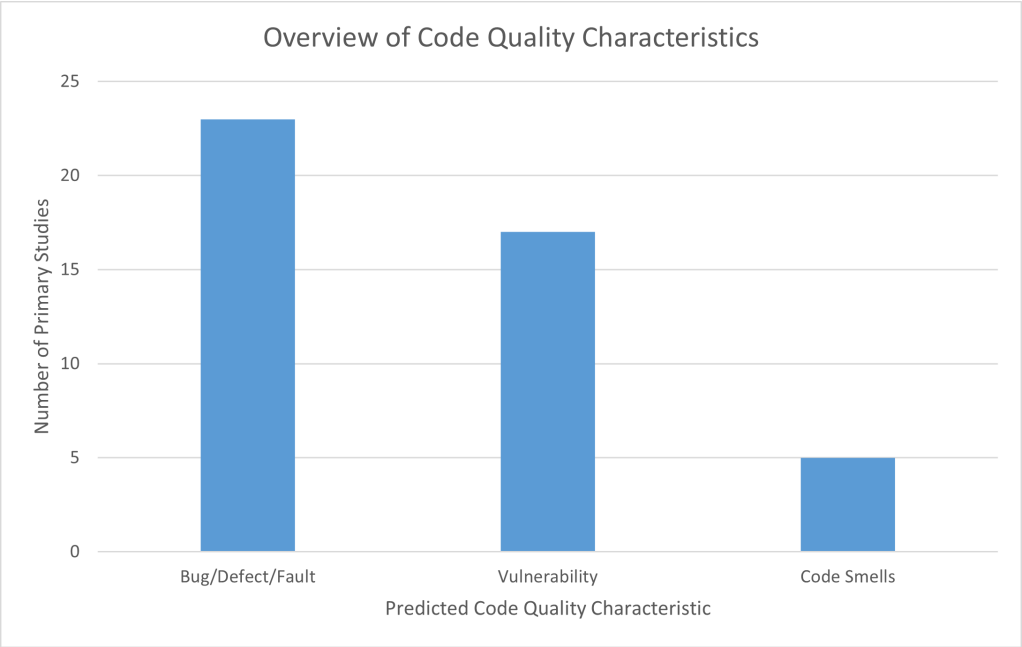

Figure 3.2: Overview of Code Quality Characteristics: Distribution of the Code Quality Characteristics that are predicted by the examined deep learning-based approaches.

In our review of the literature, we identified 45 papers that apply deep learning models to code quality evaluation. Notably, the majority of these approaches were developed for the specific purpose of bug prediction, with 23 out of 45 models ($\approx 51\%$) dedicated to this task. It is apparent that bug prediction is a focal point for research employing deep learning techniques, reflecting its critical importance in enhancing software reliability and user experience.

In terms of vulnerability prediction, this area represents the second most frequent application of deep learning models, with 17 out of the 45 approaches ($\approx 38\%$) being utilized for this purpose. As the cyber security landscape becomes increasingly complex and sophisticated, the importance of vulnerability prediction cannot be overstated. These deep learning models

are instrumental in preemptively identifying potential security breaches and ensuring robust software systems.

The application of deep learning models for code smell prediction is relatively less explored, with only 5 out of the 45 approaches ($\approx 11\%$) being dedicated to this task. This area, despite its lower representation, is crucial as it impacts code maintainability and readability, influencing the overall software development cycle.

As for maintainability and reliability prediction, these areas were addressed in the general scope of the literature review. However, it is important to note that none of these studies involved models that exclude traditional code metrics as input. Thus, the application of deep learning in predicting these particular code quality characteristics is an area that may warrant further exploration and research, given the potential benefits in terms of software longevity and robustness.

In summary, while deep learning models are increasingly used for predicting various aspects of code quality, the focus has predominantly been on bug and vulnerability prediction, with less emphasis on code smells, maintainability, and reliability prediction. Potential reasons for this focus on specific fields are explored in section 3.1.4.

RQ2: Which deep learning models have been proposed to evaluate those code quality characteristics?

A wide range of deep learning models are being employed to predict different code quality characteristics. The choice of models appears to vary depending on the specific quality attribute in focus. It is important to note that there might occur a discrepancy between the total number of papers and the total number of deep learning models in the following parts. The reason for that is the combined use of more than one deep learning model in some of the papers.

Bug Prediction Models

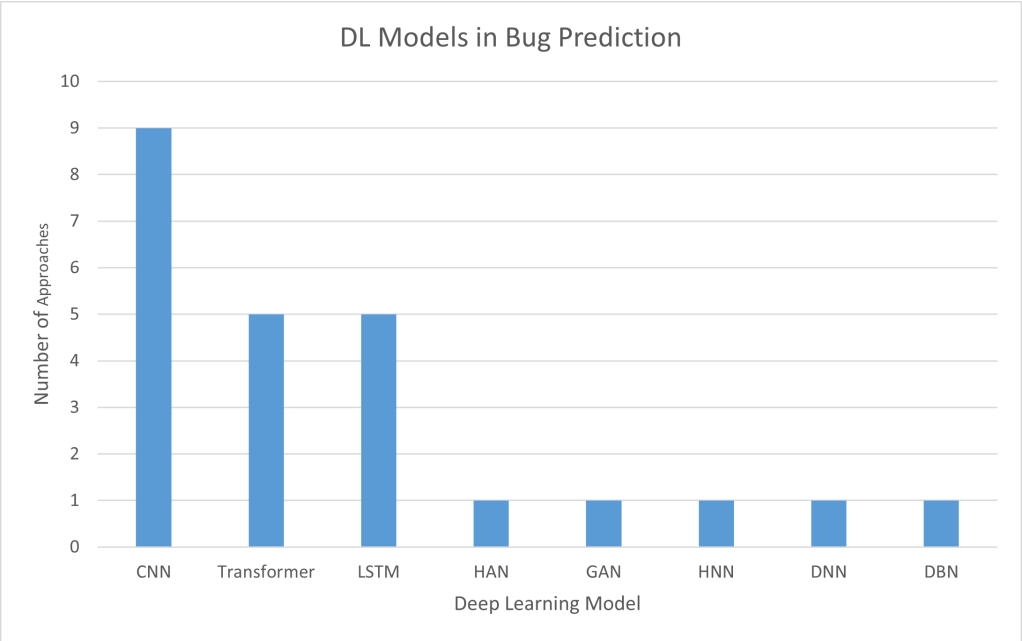

Figure 3.3: Bug Prediction DL Models: Distribution of deep learning models used in the space of bug prediction.

For bug prediction, Convolutional Neural Networks (CNN) are the most widely used, appearing in 9 out of the 24 models ($\approx 38\%$). CNNs, originally designed for image processing, have proven effective in feature learning from complex software structures.

Transformers and Long Short-Term Memory (LSTM) models follow closely, each appearing in 5 out of the 24 models ($\approx 21\%$). Transformers are known for their attention mechanism, which allows the model to focus on different parts of the input sequence when producing an output. LSTMs, on the other hand, are a type of Recurrent Neural Network (RNN) designed to remember long-term dependencies in sequence data, which can be useful when analyzing code.

Hierarchical Attention Networks (HAN), Generative Adversarial Networks (GAN), Hierarchical Neural Networks (HNN), Deep Neural Networks (DNN) and Deep Belief Networks (DBN) each appear once, highlighting the diverse array of deep learning architectures that are being explored in bug prediction.

Vulnerability Prediction Models

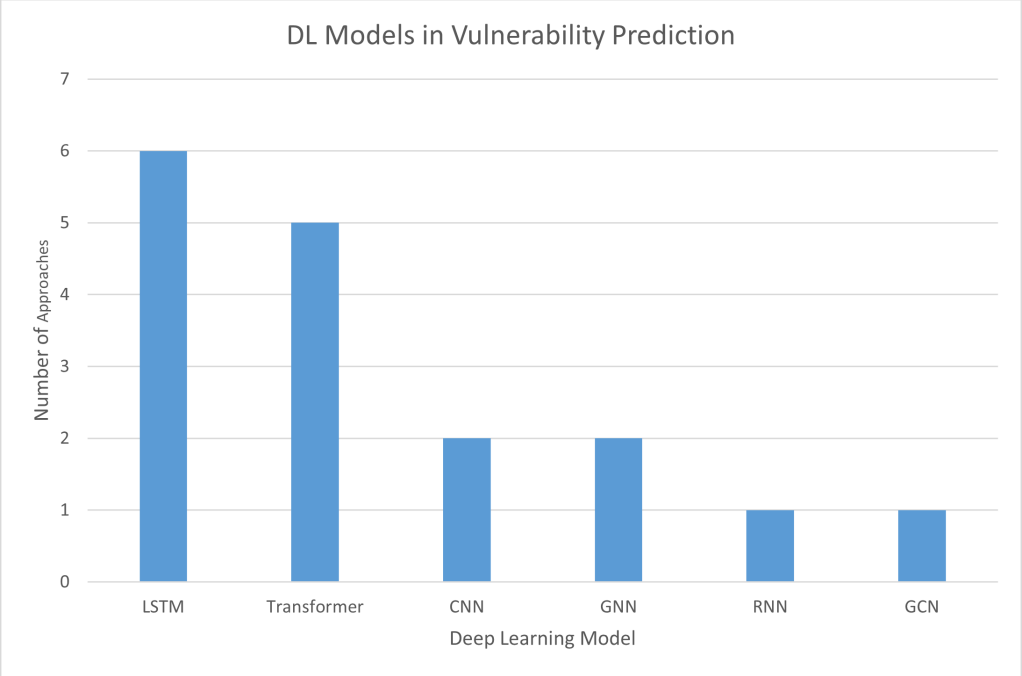

Figure 3.4: Vulnerability Prediction DL Models: Distribution of deep learning models used in the space of vulnerability prediction.

In terms of vulnerability prediction, both Transformers and LSTM models dominate, with LSTMs being used in 6 out of the 17 models ($\approx 35\%$) and Transformer being used in 5 out of the 17 models ($\approx 29\%$). The choice of these models suggests the relevance of sequence analysis and attention mechanisms in identifying software vulnerabilities.

CNNs are also utilized, but less frequently, appearing in 2 out of the 17 models ($\approx 12\%$). Graph Neural Networks (GNN) have the same representation, indicating their emerging role in analyzing code as a graph for vulnerability prediction.

Recurrent Neural Networks (RNN) and Graph Convolutional Networks (GCN) each have a single representation, adding to the variety of models being applied in this area.

Code Smell Prediction Models

For code smell prediction, CNNs take the lead again, accounting for 3 out of the 5 models. LSTMs are used in the remaining 2 models.

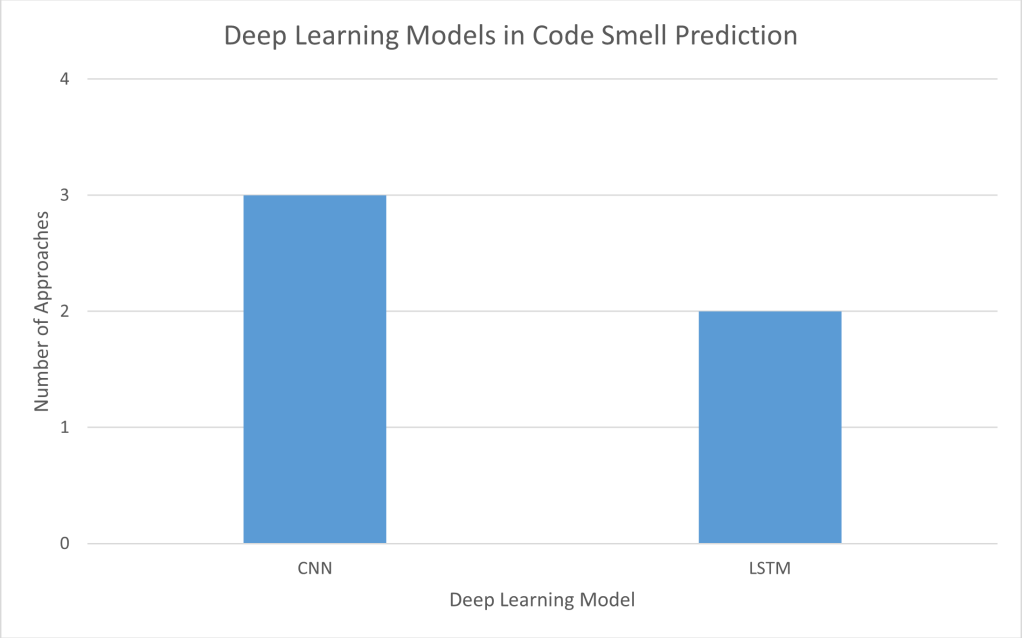

Figure 3.5: Code Smell Prediction DL Models: Distribution of deep learning models used in the space of code smell prediction.

In summary, while a variety of deep learning models are being applied in code quality evaluation, CNNs, Transformer, and LSTMs are the most popular. The choice of model appears to be influenced by the specific quality characteristic being predicted, suggesting that different models may have different strengths when it comes to analyzing code.

RQ3: How well do the models perform in predicting the code quality characteristics?

To understand the performance of deep learning models in predicting various code quality characteristics, we will now take a closer look at the empirical

results reported in the examined literature. Given the differing nature of the quality characteristics, the findings are subdivided accordingly, similar to the previous section.

For each of the three code quality characteristics - bug prediction, vulnerability prediction, and code smell prediction - we will present a detailed table that lists the respective performance for each deep learning approach. Unfortunately, the deployed performance metrics are not consistent across the publications, so we had to select a range of performance measures that are used most prominently to be able to create a concise overview of the performance of the different approaches and preserve the clarity of the presented information. The chosen performance metrics are F1-Measure, AUC (Area Under Curve), Accuracy, Recall, and Precision. However, for the sake of comparison and in alignment with standard practice in the field, we primarily use the F1-Measure as our comparison metric. The F1-Measure is a well-established and widely accepted performance metric in machine learning, being used as the most frequent performance metric in the identified papers. It combines Precision and Recall into a single value, effectively encapsulating the trade-off between those two metrics. By using the F1-Measure as our primary comparison metric, we aim to provide a balanced view of the models' performance. The chosen performance metrics are briefly defined and explained in the following section before going into a detailed overview of the different quality characteristics.

Performance Metrics

The following definitions are generalized versions of the definitions provided in section 4.4 of [32].

1. *Accuracy*: It is the measure of correct predictions out of the total predictions. If we have TP (True Positives), TN (True Negatives), FP (False Positives), and FN (False Negatives), then accuracy can be calculated as:

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

2. *Precision*: This is the measure of the relevancy of obtained results. Precision can be interpreted as the likelihood that a positive prediction by the model is correct and can be calculated as:

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

3. *Recall*: Also known as sensitivity, hit rate, or true positive rate, recall is the measure of how many of the true positives were recalled (found), out of all the actual positives. It can be calculated as:

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

4. *F1-Measure*: The F1-Measure is the harmonic mean of Precision and Recall. It provides a single score that balances both the concerns of precision and recall. It is particularly useful in the uneven class distribution where the negative class might vastly outnumber the positive class. It can be calculated as:

$$F1\text{-}Measure = 2 * \frac{Precision * Recall}{Precision + Recall}$$

5. *Area Under Curve (AUC)*: AUC stands for "Area under the ROC Curve" and can be interpreted as the probability that the model ranks a random positive example more highly than a random negative example. There does not exist a straightforward formula similar to the other metrics as it measures the entire two-dimensional area underneath the ROC (Receiver Operating Characteristic) Curve.

Bug Prediction Performance

A total of 23 approaches were identified that use deep learning for bug prediction. However, the F1-Measure is not available for some of these approaches. The model performances, as determined by their F1-Measure, are outlined below.

The approach in P37, which utilizes a Deep Neural Network (DNN), attains the highest F1-Measure with a score of 0.984, closely followed by the approach in P17 which employs a Convolutional Neural Network (CNN) with

Identifier	DL Model	F1	AUC	Accuracy	Recall	Precision	Remarks
P1	CNN	-	-	-	-	-	Only differences to other approaches given
P2	CNN	0.533	0.643	-	-	-	-
P4	HAN	-	0.81	0.63	0.25	-	Balanced Accuracy, Recall@Top20%LOC
P5	Transformer	-	-	0.3169 0.5849	-	-	Top-1 accuracy Top-5 accuracy
P7	GAN, LSTM	-	0.824	0.709	-	-	-
P9	CNN	0.75	0.732	0.701	0.819	0.691	-
P11	Transformer	-	-	0.528 0.788 0.871	-	-	Accuracy@1, Range Accuracy@5, Range Accuracy@10, Range
P12	CNN	0.625	-	-	-	-	-
P14	Transformer	0.949	-	-	0.9563	0.9418	-
P15	CNN	0.9367	-	-	-	-	-
P16	LSTM	-	0.942	-	-	-	Range
P17	CNN	0.97	-	-	-	-	Range
P25	Transformer	-	-	-	-	0.296 0.171 0.149	Precision@1, Range Precision@3, Range Precision@5, Range
P26	CNN	0.607 0.59	-	-	-	-	Within-project F1 Cross-project F1
P31	HNN	-	0.7576	-	-	-	-
P32	CNN	-	-	0.299 0.447 0.517	-	-	Top-1 accuracy Top-5 accuracy Top-10 accuracy
P37	DNN	0.984	0.998	-	-	-	Range
P39	CNN, LSTM	0.848	0.908	-	-	-	-
P40	Transformer	0.809	-	0.818	0.783	0.886	-
P42	DBN	0.641	-	-	0.707	0.63	-
P43	LSTM	0.521	-	-	-	-	-
P44	LSTM	0.7518	-	-	-	-	-
P45	LSTM	-	-	-	-	-	Only difference to metrics based features given

Table 3.4: Performance Overview of the Bug Prediction Approaches. If there is no entry (indicated by "-") for an approach and a specific performance measure, this publication did not provide the given measurement. The "Range" remark means, that a performance range was given and the upper limit of that range is displayed.

an F1-Measure of 0.97. These approaches demonstrate the highest efficacy in bug prediction among the examined models.

On the other hand, the approach P2, despite using a CNN, achieves a relatively lower F1-Measure of 0.533. Similarly, P43, which implements a Long Short-Term Memory (LSTM) model, also scores low with an F1-Measure of 0.521. These represent the least effective models in our comparison.

It is notable that approaches using CNNs, such as P9, P12, P15, P17, P26, and P39, generally demonstrate a strong performance with only a few outliers (P2). This emphasizes the potential of CNNs in bug prediction tasks.

The approach P14 that employs a Transformer model achieves an impressive F1-Measure of 0.949, demonstrating that Transformer models can also be very effective for this task. Other models such as Hierarchical Attention Networks (HAN - P4), Generative Adversarial Networks (GAN - P7), and

Hierarchical Neural Networks (HNN - P31) are also applied for bug prediction, but their F1-Measure is not available. Instead, they achieve a relatively high AUC value, indicating a good performance. Unfortunately, the comparability is not given by the differing nature of the performance metrics.

While this performance analysis provides valuable insights, it is crucial to keep in mind that factors such as the dataset used, the pre-processing steps, and the specific model configuration can greatly influence these results. Therefore, these results should be considered indicative, not definitive, of the models' predictive capabilities for bug prediction.

Vulnerability Prediction Performance

Identifier	DL Model	F1	AUC	Accuracy	Recall	Precision	Remarks
P3	LSTM	0.9847	-	0.9844	0.9848	0.9854	-
P6	Transformer	0.94	-	0.989	0.93	0.95	-
P8	Transformer	0.753 0.828	-	-	0.794 0.864	0.715 0.795	method-level slice-level
P10	LSTM	0.9	-	0.98	0.87	0.96	Range
P13	CNN	0.92	-	0.92	-	-	-
P18	Transformer	0.975	-	-	-	-	Range
P20	Transformer	0.4527 0.4529	-	-	-	0.9576 0.9526	MLP-based CNN-based
P23	GCN	0.9829	-	0.9818	0.9822	0.9837	-
P24	CNN	-	0.979	-	-	-	-
P28	LSTM	0.97	-	-	0.964	0.975	Range
P29	LSTM	-	-	-	-	-	-
P30	LSTM	0.93	0.93	0.9326	0.92	0.93	-
P33	RNN	0.9611	-	-	0.9638	0.9583	-
P34	LSTM	-	0.985	0.787	-	-	-
P35	GNN	0.9	-	0.93	-	-	-
P38	Transformer	0.963	-	-	-	0.965	Range
P41	GNN	0.94	-	0.92	0.94	0.92	-

Table 3.5: Performance Overview of the Vulnerability Prediction Approaches. If there is no entry for an approach and a specific performance measure, this publication did not provide the given measurement. The "Range" remark means, that a performance range was given and the upper limit of that range is displayed.

We identified a total of 17 different approaches that employ deep learning for vulnerability prediction. As with the previous characteristic, the F1-Measure is not available for all of these approaches. The approach in P3, which implements a Long Short-Term Memory (LSTM) model, achieves the highest F1-Measure, reaching a score of 0.9847. This is closely followed by P23, which uses a Graph Convolutional Network (GCN) and achieves an F1-Measure of 0.9829. These models emerged as the top performers in this space. The approach P20, utilizing a Transformer model, obtains the lowest

F1-Measure of 0.4527 among those that report that metric, signifying the least effective model. Approaches using LSTM models (P3, P10, P28, P30) generally show robust performance, with F1-Measures ranging from 0.9 to 0.9847. P34, which does provide the AUC instead of the F1-Measure, shows the highest AUC value (0.985) in this space, which highlights the overall strength of LSTM models in detecting code vulnerabilities. Likewise, Transformer models, as employed in P6, P8, P18, and P38, demonstrate a strong performance with F1-Measures between 0.828 and 0.975, with P20 being the only outlier. Other models such as Graph Neural Networks (GNN - P35, P41), Recurrent Neural Networks (RNN - P36), and Convolutional Neural Networks (CNN - P13) also show promising results, with their F1-Measures reaching up to 0.9611 (P33). In general, with only a few exceptions (P20), which in turn provided very high precision as a trade-off, all approaches show very good performance. Similar to the bug prediction performance, the presented performances can be influenced by different factors and are not definitive for the model's performance in vulnerability prediction tasks.

Code Smell Prediction Performance

Identifier	DL Model	F1	AUC	Accuracy	Recall	Precision	Remarks
P19	CNN	0.5177	0.8662	-	0.893	0.3645	-
P21	LSTM	0.99	-	-	0.99	0.98	Multi-label code smell, Range
		0.96			0.94	0.98	Single code smell, Range
P22	CNN	0.5579	-	-	0.7663	0.4518	-
P27	LSTM	0.85	0.96	0.9175	-	-	-
P36	CNN	0.91	-	-	0.9	0.93	Range

Table 3.6: Performance Overview of the Code Smell Prediction Approaches. If there is no entry for an approach and a specific performance measure, this publication did not provide the given measurement. The "Range" remark means, that a performance range was given and the upper limit of that range is displayed.

We found 5 different approaches that use deep learning for predicting code smells. Following is a summary of the performance of these models based on their F1-Measure. The approach identified as P21, which uses a Long Short-Term Memory (LSTM) model, achieved the highest F1-Measure with a score of 0.99. This indicates superior effectiveness in predicting code smells. However, it is important to note that the performance in P23 is given

as a range with 0.99 being the upper boundary. On average, the performance might be slightly lower, but still very high. In contrast, the approach P19, which utilizes a Convolutional Neural Network (CNN), achieves the lowest F1-Measure of 0.5179 among the studied models, signifying a lesser degree of effectiveness in the same task. Approaches that employed LSTM models (P21, P27) demonstrate strong performance, underscoring the potential of LSTM models in detecting code smells. The CNN model used by P39 yields a high F1-Measure of 0.91, again being the upper boundary of the performance range provided, contrasting with the lesser effectiveness of P19 and P22 that also use CNNs (F1-Measures of 0.5177 and 0.5579, respectively). This suggests that the performance of CNNs can vary significantly depending on the specifics of the implementation and dataset. Again, the performance may be influenced by the approach setup and should only be viewed as indicative of the models' performance

3.1.4 Conclusion

This systematic literature review has yielded findings on the use of deep learning models for code quality evaluation, specifically a research focus in the domains of bug prediction, vulnerability prediction, and code smell detection. Researchers are exploring a variety of models to tackle these different aspects of code quality, with certain models proving more popular in some areas than others.

Convolutional Neural Networks (CNN) were found to be the most used model for bug prediction and code smell detection. With top-performing models achieving an F1-Measure of 0.984 and 0.91 respectively. These models, originally designed for processing visual information, seem to lend themselves well to these tasks, likely due to their ability to extract local and global features from complex structures, analogous to analyzing code patterns.

Transformer and Long Short-Term Memory (LSTM) networks are equally prominent in bug and vulnerability prediction. The high performance of these models (achieving an F1-Measure of up to 0.949 and 0.9847 respectively) signals the relevance of sequence analysis and attention mechanisms

in both tasks. The ability of transformers to handle long sequences and focus on specific parts of the input when producing an output is likely a contributing factor to their prevalence. Similarly, LSTMs, with their ability to remember long-term dependencies, show promise in dealing with the inherently sequential nature of code.

In the realm of vulnerability prediction, it is interesting to see the emergence of graph-based approaches like Graph Neural Networks (GNN) and Graph Convolutional Networks (GCN). As code can often be represented as a graph (e.g. Abstract Syntax Trees), the application of graph-based models could offer new ways to detect software vulnerabilities, reflected by model performance of up to 0.9828 F1-Measure.

The variance in model application across the different quality characteristics suggests that the optimal model choice may depend on the specific task at hand. Different code quality characteristics may require different representations and analyses, leading to the use of different deep learning architectures.

An interesting observation is the pronounced focus on bug prediction in the literature, with more than half of the models being dedicated to this task. A possible explanation for this could be the objective nature of bug prediction. The task is binary - code is either buggy or not - and can be evaluated based on this clear distinction. Moreover, there are numerous publicly available datasets for bug prediction[2], making it a convenient and attractive area for research and model evaluation.

A similar argument can be made for vulnerability prediction being the second most focused quality characteristic. A variety of tools[3] and datasets[4] are available online for either creating their own dataset for training and evaluation purposes or using the existing ones to compare the performance of the approach to others in the same domain.

In contrast, the area of maintainability prediction seems to be underrepresented. Only one paper was found in the initial search ([69], Part of

[2] `https://github.com/dspinellis/awesome-msr`

[3] `https://owasp.org/www-community/Vulnerability_Scanning_Tools`

[4] `https://ieee-dataport.org/keywords/software-vulnerability-datasets`

Collection 2 in 3.2), but the inclusion of traditional code metrics as part of the model input caused it to be excluded from the final collection. Maintainability is a complex, but crucial aspect of code quality, as it influences the lifespan and evolution of software systems [70]. It encompasses aspects such as readability and understandability, which are highly subjective and thus more challenging to measure and predict accurately using deep learning. The lack of extensive and comparable datasets for training and evaluating maintainability prediction models has likely also contributed to the reduced focus on this area. However, with the recent emergence of new datasets, there is an opportunity for more research in this area. Given the success of deep learning models in predicting other code quality characteristics, it is worth considering whether these techniques could be successfully applied to maintainability prediction as well.

In summary, this review not only provides a comprehensive overview of the current state of deep learning for code quality evaluation but also highlights promising directions and identifies areas for further research. The findings underscore the potential of deep learning in this domain and pave the way for future research endeavors, particularly in the under-explored area of maintainability prediction, which will be the focus in the later parts of this master thesis. CNN, Transformer and LSTM-based approaches seem to be promising candidates to be applied to maintainability prediction as they show strong performance in the other code quality domains.

3.2 Snowballing

The Systematic Literature Review (SLR) has provided a broad overview of the current state of deep learning models for code quality evaluation. The review identified a prominent gap in the field of maintainability prediction. To delve deeper into this unexplored domain and expand upon the findings of the SLR, we performed a literature review based on the snowballing methodology.

In the following sections, we will detail the Snowballing process based on the guidelines by Wohlin [71] and share our findings, aiming to shed light on potential deep learning approaches for maintainability prediction which could form the foundation for further research in this area.

3.2.1 Process

Snowballing is an effective method for identifying relevant research papers by investigating the citation networks of already identified papers. In this context, we employ the Snowballing method to focus on maintainability prediction in the realm of deep learning models for code quality evaluation.

Starting Set Identification

The Snowballing process starts with three identified papers that discuss the prediction of maintainability or sub-characteristics of maintainability such as readability or complexity using deep learning models. Despite these papers not meeting all the inclusion criteria for our original systematic literature review, they provide a good starting point for the snowballing process, as they all have the combination of deep learning and maintainability prediction as their main objective. The papers that form the starting set are listed in table 3.7.

Identifier	**Title**	**Reference**
P46	Measuring code maintainability with deep neural networks	[69]
P47	Neural language models for code quality identification	[72]
P48	A Preliminary Study on Using Text- and Image-Based Machine Learning to Predict Software Maintainability	[73]

Table 3.7: Identified papers in the snowballing start set.

Forward and Backward Snowballing

This phase involves finding new relevant papers by examining the citation network of the starting set. This includes both forward snowballing (investigating the papers that cite our starting papers) and backward snowballing (examining the papers that are cited by our starting papers). For each new paper identified in this phase, a similar forward and backward snowballing process is carried out, creating a cascading effect of paper identification. This process stops when no new papers are identified.

Screening and Eligibility Check

As papers are identified via snowballing, they are subjected to a screening process to ensure their relevance to the topic of interest. Similar to the SLR, this involves checking the title and abstract for relevance, and if deemed potentially relevant, a more detailed full-text screening is performed. The inclusion and exclusion criteria are similar to the SLR, but with a pronounced focus on the prediction of maintainability and its sub-characteristics and can be found in table 3.8.

By following this iterative process, we aim to identify a collection of relevant papers that can deepen our understanding of the research landscape in maintainability prediction using deep learning models. The findings from this snowballing process will be presented in the following section.

3.2.2 Results

First Iteration of Backward Snowballing

The initial backward snowballing, which examined the sources cited by our three initial papers, identified a considerable number of 147 sources.

An initial screening was conducted based on the publication date of these papers. Since the technology and methodologies in the field of deep learning and software engineering evolve rapidly, the review was bound to include only those papers that fell in the same timeframe as the initial SLR. After

Inclusion	Exclusion
• Papers that are published in the last 4 years (2020-2023) • Papers that focus on deep learning models • Papers that focus on code maintainability prediction, including aspects of maintainability (readability, complexity, comprehensibility, understandability, modularity, reusability, analysability, modifiability, and testability) • Approaches that directly use text or a direct representation of the code like AST • Papers that present an empirical evaluation of the presented deep learning model in the context of code quality evaluation/prediction • Papers that provide an implementation, either publicly available or defined by framework and configuration • Papers that are peer-reviewed • Papers that are written in English	• Papers that were published more than 4 years ago (2019 and earlier) • Papers that solely use traditional machine learning (non-deep learning) or other non-ML techniques • Approaches that use static code metrics as input • Papers that do not present an empirical evaluation • Papers that focus only on theoretical aspects or mathematical models without any practical application • Papers that are opinion pieces, editorials, or non-peer-reviewed • Papers that are not written in English

Table 3.8: Inclusion and Exclusion criteria

this temporal exclusion (publication year 2019 and earlier) was applied, a total of 118 papers were excluded, leaving 29 papers for further evaluation.

Following this, the remaining papers were subjected to a title screening. This step aimed to quickly exclude sources that did not meet the relevance criteria. The screening resulted in the exclusion of 23 additional papers. The rationale for their exclusion was primarily based on the irrelevance of the topics to the research question, i.e., these papers did not pertain to maintainability prediction or the application of deep learning.

The remaining six papers were then thoroughly examined based on their abstracts and, if necessary, further reading of the full text. This in-depth examination led to the exclusion of all six papers. The main reasons for exclusion at this stage were the usage of traditional code metrics as input for the model or a lack of focus on deep learning techniques. Therefore, the first iteration of backward snowballing did not yield any further papers for inclusion.

First Iteration of Forward Snowballing

The forward snowballing process, which involved identifying papers that cited our three initial papers, was less exhaustive. Using Google Scholar[5] as one of the most comprehensive online search engines for research papers, only one source was identified during this stage. However, upon a brief review of the title, it was clear that this paper did not meet the inclusion criteria. The paper was consequently excluded from further consideration.

Conclusion

In conclusion, despite screening almost 150 sources, the first iteration of the snowballing process did not result in the identification of any additional relevant papers. Given that no new papers were discovered in this step, the snowballing process is brought to a halt, as per the standard snowballing methodology.

This outcome fortifies the conclusion from our initial systematic literature review - a conspicuous gap exists in the literature when it comes to the application of deep learning models for predicting software maintainability.

To further substantiate our findings, a supplementary systematic literature review was carried out with the same focus as the snowballing process. The reiteration of the lack of relevant literature in this specific domain, as showcased by the results of this second SLR, adds weight to our initial observations. In the interest of thoroughness and ensuring methodological

[5] `https://scholar.google.de/`

transparency, the subsequent section will provide a succinct outline of the research strategy and process undertaken in this targeted systematic literature review.

3.2.3 Systematic Literature Review Maintainability

This second SLR followed the same systematic, comprehensive, and repeatable process as the first one, but with a sharper focus on our identified research gap.

Search Strategy

The search strategy retained the stringent criteria of the first SLR but homed in on the subject of maintainability prediction. Revised to include terms directly related to 'Maintainability Prediction', the following search string was defined:

TITLE-ABS(("Deep Learning" OR "LSTM" OR "CNN" OR "RNN") AND "Code" AND ("Maintainability" OR "Readability" OR "Complexity" OR "Comprehensibility" OR "Understandability" OR "Modularity" OR "Reusability" OR "Analysability" OR "Modifiability" OR "Testability") AND ("Prediction" OR "Assessment" OR "Evaluation"))

Based on the same rationale as in the first SLR, Scopus was searched using this search string on 05.07.2023.

Study Selection

The selection process mirrored the snowballing process: a three-step procedure encompassing title review, abstract screening, and full-text reading was followed. For this purpose, the same inclusion and exclusion criteria (see table 3.8) were deployed. The studies were first filtered by publication date, then subjected to a title and abstract review, and the remaining papers were assessed using a full-text review.

Results

The results from this SLR echoed those from the initial review and the snowballing process. The search yielded an initial body of literature containing 201 papers. The first filtering step based on the publication year excluded 30 papers, leaving 171 papers for title screening. During the title screening, 166 papers were excluded, resulting in six papers for the final, more thorough screening. All of those six papers had to be excluded because they either were based on code metrics as input or did not focus on the prediction of maintainability or the utilization of deep learning models.

In conclusion, the second SLR solidifies the findings of the previous reviews. It validates the identified research gap and further emphasizes the need for exploration and research in this domain.

3.3 Datasets for Maintainability Prediction

One of the crucial elements for deep learning models to function effectively is the availability of extensive and high-quality datasets for training and evaluation. During the course of our literature reviews, we identified datasets that could be of interest for research in the domain of maintainability prediction.

Maintainability is a multifaceted concept, encompassing characteristics such as readability, understandability, and modifiability of code, among others. Due to the complex and somewhat subjective nature of these characteristics, the creation of a suitable dataset poses significant challenges.

Despite these challenges, one dataset for training and one dataset for evaluation have been identified and are available for use. It is important to note that, as of now, there is no standardized dataset for this purpose that is universally accepted or used by the research community.

In the following chapters, we will further investigate these datasets, and assess their suitability for our needs.

3.3.1 Training Dataset

The cornerstone of any deep learning application is an appropriate and extensive training dataset, which aids the model in learning patterns, inferring relationships, and generalizing from specific instances. During our literature reviews, one significant dataset was identified that has been specifically compiled for maintainability prediction in Java projects. The dataset was part of the paper "Measuring code maintainability with deep neural networks" by Hu et al. [69] and is available on Github[6].

This dataset employs a binary classification system, dividing projects into two categories: 'high-quality' and 'low-quality'. High-quality projects represent open-source applications that are actively maintained by seasoned programmers and uphold certain code quality standards. These projects reflect higher levels of maintainability and are thus considered 'good'. The high-quality category in this dataset encompasses 378 projects.

Conversely, the low-quality projects are those that do not meet specific thresholds on GitHub, such as having fewer than 5 contributors, less than 5 closed pull requests, no star ratings and no reviewed closed pull requests. These projects exhibit lower maintainability and are classified as 'bad'. This category contains 54882 projects.

Java classes extracted from these projects were assigned labels according to the quality of their parent project. To eliminate the effect of any confounding factors and to create a balanced dataset, the authors employed simple random sampling. They divided classes based on their Lines of Code (LOC) into a high-maintainability and a low-maintainability set. Classes from the larger set were randomly dropped until the size of the two sets was equal. This process resulted in a dataset of 697257 'good' and an equal number of 'bad' classes.

The existence of such a robust, balanced dataset is crucial for future research in this area. The size and diversity of the dataset allow for the development and testing of deep learning models capable of handling real-world complexity and variety.

[6] `https://github.com/DeepQuality/Deep_Maintainability`

3.3.2 Evaluation Dataset

Evaluation datasets are crucial for assessing the performance of a model, serving as a reliable means to assess its ability to generalize beyond the training data. A noteworthy dataset, tailor-made for predicting software maintainability, surfaced during our literature reviews. After identifying the absence of a publicly available and comprehensive dataset in this space, Schnappinger et al. [74] set out to create such a dataset [75] and make it available online[7]. The creation process is comprised of the following characteristics:

Selection of Code: In a concerted effort to eschew project-specific or domain-centric bias, code snippets were curated from a diverse range of nine different projects, both open and closed source.

Prioritization of Samples: The collected metrics were used as the basis to cluster the code snippets. A distinct order of priority was assigned to these snippets, based on iterative passes through the clusters. The highest priority was granted to the unlabeled code snippet from an accessible codebase that had not garnered sufficient valid ratings.

Labeling: A labeling platform was employed to showcase code snippets to 70 experts from a variety of companies for evaluation. The granularity of the snippets was determined at the Java class level, optimizing the feasibility of the labeling effort. This, however, meant that only intra-class characteristics could be scrutinized. The platform was also designed to accept free text comments from experts. The assessed code characteristics were overall maintainability, readability, understandability, complexity, and modularization and the assessment was done using a four-part Likert-scale[8] ranging from strongly disagree to strongly agree.

Validation and Aggregation: Each code snippet underwent evaluation by multiple experts. The gathered labels were subsequently validated. A final step of aggregation was applied where individual error probabilities for each participant were computed and a consensus score was derived.

Dataset Characteristics: This dataset exhibits an imbalance, with the

[7] `https://figshare.com/articles/dataset/A_Software_Maintainability_Dataset/12801215`

[8] `https://www.britannica.com/topic/Likert-Scale`

majority of the files evaluated as highly maintainable, while only a minority received negative evaluations. In this sample, merely 22% of the Java classes were deemed problematic.

Availability: The authors have made the results of this study publicly available. This includes the analyzed open-source code and the corresponding labels. However, the code from closed-source systems is only disseminated on a case-by-case basis.

This dataset's unique characteristics, coupled with its public availability, make it a valuable resource for future research. The imbalanced nature of the dataset presents a realistic scenario, reflecting the skew in maintainability that may be observed in real-world projects. This provides an opportunity to evaluate models that can perform well even under such challenging conditions.

Chapter 4

Experiments

This chapter is a systematic exploration into the practical facets of our research, designed to formulate, implement, and evaluate our approach with the aim of filling the research gap identified in the previous chapter, maintainability prediction using deep learning models. It is structured to provide a comprehensive account of the experimental procedures, methodologies, and results including a critical discussion of our findings, ensuring a scientific and transparent exposition of our investigative process.

4.1 Initial Procedure

The initial procedure for the experiments was divided into three distinct phases, each targeting a unique aspect of model applicability and performance in predicting software maintainability.

4.1.1 Phase 1: Applicability of Existing Models

- **Objective**: Investigate whether existing pretrained models from other domains such as bug or code smell prediction, could be repurposed for maintainability prediction.

- **Approach**:
  - Deploy already trained models directly to the evaluation dataset without further training.
  - Analyze their performance in terms of prediction accuracy.
- **Criteria for Model Selection**:
  - Availability of implementation.
  - Input compatibility: The models should be able to process source code at a single class level. Additionally, if any representation conversion is required, it should either be done internally by the model or have a specified conversion method and format (e.g., AST conversion or embedding).
  - Compatibility with Java code, as both the evaluation and training datasets are based on Java
  - Model should operate as a standalone application and not as a plugin or extension.

4.1.2 Phase 2: Efficacy of Transfer Learning

- **Objective**: Determine how transfer learning could enhance the performance of models in maintainability prediction.
- **Approach**:
  - Use the models that were assessed in Phase 1 and retrain only the final classification layer with the maintainability training set.
  - Evaluate and compare their performance with the outcomes from Phase 1.

4.1.3 Phase 3: Vanilla Model Training

- **Objective**: Investigate the performance of vanilla models when trained exclusively on the maintainability dataset.
- **Approach**:
  - Choose suitable models.
  - Train those models on the maintainability training dataset.
  - Evaluate using the evaluation dataset.
  - Compare their performance with the models from Phase 1 and Phase 2.

4.1.4 Challenges and Procedure Modification Decision

During the execution of Phase 1, a series of challenges emerged that profoundly impacted the feasibility of the initial procedure.

Limited Compatible Implementations: While 14 papers, listed in table 4.1 initially appeared promising due to their available implementations, a deeper dive revealed compatibility issues. Three models (P9, P11, P34) were found incompatible with our dataset's input format, and another (P6) was structured as a plugin rather than a standalone application.

Language Specificity: Another stumbling block was the language specificity of certain models. Five out of the ten remaining models (P10, P20, P29, P33, P35) were trained on programming languages other than Java, making them unsuitable for the task.

Ambiguous Input Formats: Among the final five models, challenges persisted. Ambiguities in their input formats raised significant hurdles. The process of transforming the dataset to meet each model's requirements turned out to be not only time-consuming but, in certain instances, impractical.

With these challenges effectively stopping the progression of Phase 1, it became evident that Phases 2 and 3, which were designed to build upon the foundation laid in Phase 1, were not viable to pursue in their planned form.

Given the circumstances, it was crucial to make an informed decision to overhaul the initial experimental procedure, ensuring meaningful and actionable results from the experiments.

Identifier	Title	Reference
P4	DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction	[27]
P6	A transformer-based IDE plugin for vulnerability detection	[29]
P9	Context-Aware Code Change Embedding for Better Patch Correctness Assessment	[32]
P10	VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python	[33]
P11	DreamLoc: A Deep Relevance Matching-Based Framework for bug Localization	[34]
P20	VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection	[43]
P21	Multi-Label Code Smell Detection with Hybrid Model based on Deep Learning	[44]
P23	Can Deep Learning Models Learn the Vulnerable Patterns for Vulnerability Detection?	[46]
P24	Investigating Graph Embedding Methods for Cross-Platform Binary Code Similarity Detection	[47]
P29	A context-aware neural embedding for function-level vulnerability detection	[52]
P33	AutoVAS: An automated vulnerability analysis system with a deep learning approach	[56]
P34	Asteria: Deep Learning-based AST-Encoding for Cross-platform Binary Code Similarity Detection	[57]
P35	DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network	[58]
P40	Deep Just-In-Time Inconsistency Detection Between Comments and Source Code	[63]

Table 4.1: Papers from the SLR that had their implementation publicly available at the time of writing.

4.2 Revised Procedure

The complications encountered in Phase 1 of the initially planned procedure urged us to seek a more robust and promising approach to assess maintainability in code. We found our foundation in the work of Sengamedu and Zhao titled "Neural Language Models for Code Quality Identification" [72]. Their work's novelty, coupled with its applicability, provided an ideal stepping stone to our revised procedure.

4.2.1 Foundation: Neural Language Models for Code Quality Identification

Sengamedu and Zhao, in their paper [72], introduce a novel approach to detecting code quality issues using neural language models. The core idea is that these models inherently possess vast information about code quality. By analyzing the token probability of each sub-token, along with cross-entropy and log probability at the method level, the approach can identify the following four distinct types of code quality issues:

- Token-Level Errors: The language model assigns probabilities to each code token. A low probability indicates that the token is "surprising" or less likely to appear in a particular position. The model can suggest a different token with a higher probability for that position, helping to identify potential token-level errors.
- Unnatural Code: The model identifies code that deviates from common conventions or patterns. For instance, starting a method with an identifier name without any modifiers might be flagged as unnatural.
- Repetitive Code: The model can detect repetitive patterns in code. For example, repeatedly calling a method with hard-coded indices or having a repeated code structure multiple times in a method. The model suggests refactoring such code for better readability and maintainability.
- Long and Complex Code: Traditional measures of code complexity, like Line of Code (LOC) and Cyclomatic Complexity (CC), have limitations. In this approach, the Log Probability (LP) of a code sequence is used to measure its complexity. A high LP indicates that the code snippet is long and unpredictable.

The core takeaway from their approach is that it leverages the implicit knowledge within language models to offer actionable insights into code quality, with the added advantage of minimizing false positives through project-specific models.

4.2.2 Our Approach

Building on Sengamedu and Zhao's foundation, our strategy pivots to test a range of transformer models sourced from HuggingFace[1]. We aim to discern if the underlying principle of using neural language models for code quality identification could be extrapolated to predict maintainability comprehensively. Furthermore, we also want to appraise its efficacy in capturing other characteristics assessed in the dataset, such as readability, understandability, complexity, and modularization. Based on Sengamedu and Zhao's findings, we formulate the following Research Question for our experiments:

- **RQ:** Can the cross-entropy between the next-token probability of a transformer model and the actual code be used as an indicator for code maintainability and its sub-characteristics?

4.3 Experiments Process

The experimentation phase is a critical component of our research, offering empirical insights to validate or challenge our hypothesis. In this section, we delineate the systematic methodology employed during our experiments. The process, selection criteria, challenges, and adjustments made are all discussed in detail to provide a comprehensive understanding and the rationale behind each step. This transparent documentation ensures both replicability and a foundational understanding of the experiment's results.

Our overall process, as depicted in figure 4.1, is comprised of the following key steps:

1. **Goal**: We assess whether there is a relation between the code generated (measured with cross-entropy) and maintainability aspects (given by the evaluation dataset).

[1] `https://huggingface.co`

2. **Cross-Entropy Computation:** We run 10 different models and let them generate code (based on the java class files available in the evaluation dataset) and compute the cross-entropy for each java class file based on the next-token probabilities.
3. **Evaluating the relation:** We evaluate the relation between cross-entropy and the values for each characteristic (overall maintainability, readability, understandability, complexity, modularization) assessed in the dataset and each deep learning model.
    - For categorial/binary values: We train a logistic regression classifier using the cross-entropy values and the binary assessment values of the evaluation dataset and then determine performance metrics like F1-Measure, Recall, and Precision.
    - For continuous values: We use a linear regression trained on the cross-entropy values and an average score based on the Likert distribution and then determine the mean squared error for the linear regression.

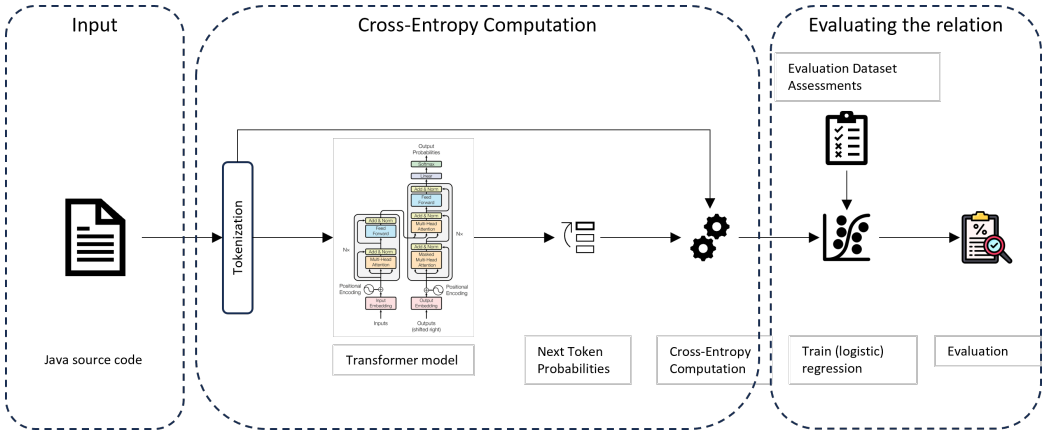

Figure 4.1: Steps of our experimental process.

Initially, the intent was to experiment with both text generation and fill mask models. However, as we proceeded further into implementation and initial testing, it became apparent that the fill mask models posed significant computational challenges. Due to the necessity to iterate over each token,

mask it, and then engage the model, the runtime surged drastically. To provide context, using the Bloomz-560m model for text generation (more details in section 4.3.3) resulted in a runtime of 50 seconds, whereas the CodeBERTa-small-v1[2] fill mask model consumed a staggering 57 minutes 46 seconds. This disproportionate increase in runtime, especially considering that Bloomz-560 has seven times the number of parameters compared to CodeBERTa-small-v1, led us to disqualify the fill mask models such as BERT-based models from our experiments, focusing our efforts primarily on text generation models based on GPT-2 and Llama-2.

4.3.1 Implementation Details

In this section, we delve into the details of our experiment's implementation, focusing on the tools, procedures, and rationale guiding our methodology.

1. **Libraries and Frameworks:**
    - We employ the **PyTorch**[3] library, an open-source machine learning framework known for its dynamic computational graph and efficient memory usage, particularly useful for deep learning.
    - The **Hugging Face**[4] **transformers** library, a renowned repository for pre-trained transformer-based models, is used to facilitate model inference.
2. **Code Preprocessing:**
    - Before processing the Java code, we purge it of comments and empty lines, ensuring a clean input for the subsequent steps.
3. **Model Input Challenges:**
    - Initially, our strategy entailed feeding the entire Java class into our models. However, we encountered several problems. The tokenized representation of these classes often surpassed the maximum input size allowed by many models. Additionally, the mem-

[2] `https://huggingface.co/huggingface/CodeBERTa-small-v1`
[3] `https://pytorch.org`
[4] `https://huggingface.co`

ory footprint of the tokenization quickly outgrew the memory capabilities of the employed hardware, especially for tokenizers with a larger vocabulary.

- Based on those findings, the idea emerged to divide a class into its constituent methods and subsequently compute cross-entropy as an average across these methods. This approach, however, presented two new challenges:
  - Accurately splitting the classes into methods while retaining their original formatting proved difficult. Regular expressions were ill-suited to cover every possible scenario in a complex Java class, and using a parser to then rebuild the code from the Abstract Syntax Tree (AST) did not guarantee retention of the original format.
  - Despite the segmentation, individual methods could still breach the model's maximum input length.
- Our final solution is to segment the entire class into chunks corresponding to the model's maximum permissible input size. This methodology, while pragmatic and effective, offers room for refinement in future endeavors.

4. **Cross Entropy Computation:**
   - The core metric under evaluation is the cross-entropy, which we define similar to Sengamedu and Zhao [72]: If we have a series of tokens represented as $t_0, t_1, t_2, ..., t_N$, a well-trained language model will determine the likelihood of this sequence as the product of the probabilities of each token given its predecessors. The cross-entropy for this sequence is essentially the average log probability for each token. It is calculated as $C(t_0, t_1, t_2, ..., t_N) = -\frac{1}{N} \sum_{m=1}^{N} log_2 P(t_m | t_{m-1}, ..., t_0)$ with $P(t_0, t_1, t_2, ..., t_N) = \prod_{m=1}^{N} P(t_m | t_{m-1}, ..., t_0)$. Importantly, cross-entropy does not change based on the length of the sequence, making it a popular measure for evaluating language models. A lower value for cross-entropy suggests that the model predicts with greater confidence.

- Our approach needs some minor adjustments to adapt to our use of chunks. Specifically, we deploy the CrossEntropyLoss() function from PyTorch to compute the cross-entropy for each chunk, adjusting for the number of tokens within each segment. This method guarantees that the impact, especially of the last, potentially shorter chunk, is accurately scaled.

5. **Evaluation Metrics Computation:**

   - To be able to analyze, evaluate, and interpret our results, we use different evaluation measures:
     - Accuracy, Precision, Recall, F1-Measure, and AUC (Area Under the ROC Curve), based on a logistic regression classifier that was trained on the computed cross-entropies and the corresponding binary assessments. Similar to [73], we split the data for the binary assessment into seemingly optimal code ("strongly agree" in the case of overall maintainability) and not optimal ("agree", "disagree", "strongly disagree" in the case of overall maintainability). This has to be done because of the imbalance in the dataset: most Java class files are labeled as maintainable ("strongly agree" and "agree"), while only a few are labeled the opposite (see Table 1 in [73] for more details). This holds true for the characteristics that benefit maintainability (readability and understandability) and is reversed for the ones that hurt maintainability (complexity and modularization). To be able to train the logistic regression classifiers reliably, such a strong imbalance is not beneficial, as underrepresented labels do not get enough attention during training, hurting the prediction performance of the classifier.
     - MSE (Mean-Squared-Error) of a linear regression that was trained using the cross-entropies and the average Likert-Scale of the specific characteristic.
     - Spearman Rank Correlation, including p-value, between the cross-entropies and average Likert-Scale.

6. **Infrastructure & Execution:**
    - Out initial experiments leveraged a Nvidia RTX 3080 GPU with 10GB VRAM. However, memory constraints, especially with larger models, necessitated a shift. Consequently, we transitioned to a GPU cluster housed on a cloud computing platform. Most models, barring the Code Llama variants, were evaluated on a server equipped with a Nvidia RTX A6000 (40GB VRAM). The resource-intensive Code Llama models demanded even more capacity, prompting us to use a Nvidia A100 (80GB VRAM).
7. **Additional Notes:**
    - Although one Fill Mask model is integrated within our implementation for potential reproducibility, it is omitted from the results and evaluations, consistent with our earlier discussions.
    - The codebase is publicly available at `https://github.com/marcdillmann/DeepCodeMaintainability`.

By offering this detailed account and publicly available codebase, we aim to increase transparency, assist reproducibility, and invite future work based on our efforts.

4.3.2 Base Models

Within the domain of natural language processing and machine learning, the choice of model architecture has a significant impact on the accuracy and robustness of any research experiment. This section provides a brief description of the two pivotal models that serve as the basis for all fine-tuned models examined in our experiments: GPT-2 and Llama-2.

GPT-2

GPT-2 was introduced by Radford et al. [76] from OpenAI as a successor to GPT[5]. It stands for "Generative Pre-Trained Transformer 2" and rep-

[5] `https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf`

resents an important advancement in transformer architecture, having been pre-trained on an extensive corpus of English textual data in a self-supervised manner. This model's pre-training operates on raw texts without any human-supervised labels, harnessing vast amounts of publicly accessible data. The training methodology employed is automated, generating both inputs and corresponding labels from the textual data.

The model's training paradigm involves processing sequences of continuous text of specified lengths. Utilizing an internal masking mechanism, GPT-2 ensures that the prediction for a given token only leverages preceding tokens, excluding any subsequent ones. This sophisticated training approach empowers GPT-2 to develop a nuanced internal representation of the English language, which can be harnessed to extract features for a variety of downstream tasks. While GPT-2 is inherently versatile, it exhibits peak efficacy in its foundational objective: synthesizing coherent text based on provided prompts.

GPT-2 has been architected in a spectrum of sizes to cater to diverse computational requirements and application domains. At the most compact end, GPT-2 features a model with 12 layers and approximately 117 million parameters, matching the complexity of its predecessor, GPT. This scales up to the most expansive variant, boasting a substantial 48 layers with roughly 1.5 billion parameters. These configurations allow for a graded trade-off between computational efficiency and model expressiveness. The entire GPT-2 family was trained on the WebText dataset, a vast compilation of web-based content. This dataset forms the backbone of the model's extensive knowledge base, capturing a broad array of topics, writing styles, and information densities present on the internet.

Llama-2

Llama-2 (Large Language Model Meta AI 2), delineated by Touvron et al. [77], represents an advanced sequence of pre-trained and subsequently fine-tuned large language models built upon the transformer architecture. Acting as an evolution of the original Llama model, this series, often referred to as

Llama 2 Chat, was crafted to enhance text generation within conversational contexts. Comparative benchmarks highlight their superior performance over numerous open-source conversational models. When subjected to human-led assessments focused on utility and safety, Llama-2 showcased promise as a viable competitor to proprietary models.

Mirroring the versatility of GPT-2, the Llama-2 models are available in a spectrum of sizes, with configurations spanning from 7 billion to an extensive 70 billion parameters. This diversity ensures compatibility with varied application requirements and computational setups.

For the development of the Llama 2 series, the foundational pre-training methodology described in Touvron et al. [78], served as a starting point. While retaining the essence of the auto-regressive transformer, modifications were introduced to enhance performance metrics. These modifications encompassed rigorous data cleansing procedures, refreshed data assortment strategies, an expanded training phase encompassing 40% additional tokens, an augmented context window, and the integration of grouped-query attention (GQA) to augment the inference scalability, particularly for the more substantial models.

The data corpus utilized for training Llama-2 is based on diverse publicly available sources, explicitly excluding any content from Meta's own offerings. A concerted effort was undertaken to omit sources notorious for hosting extensive personal information about individuals. With a training foundation of 2 trillion tokens, the emphasis was on striking a balance between performance and computational costs, accentuating factually correct sources to improve accuracy and minimize potential inaccuracies.

4.3.3 Model Selection

In this section, we give an overview of the specific models selected for our experiments with the rationale behind the selection. The overview delineates different characteristics of the models and can be found in table 4.2.

In the process of model selection for our experiments, multiple rationales anchored our decisions:

1. **Popularity and Reputability:** We decided to use models that have garnered significant attention in the academic and industry landscapes and were developed by reputable entities. Specifically, both Bloomz (developed by BigScience[6]) and Code Llama (developed by Meta AI[7]) emerged as our top picks. This was determined through metrics such as download counts under the category of text generation with the language specified as "code". Crucially, models demanding additional preprocessing of the input, such as the requirement of adding specific tokens, were excluded from our consideration.
2. **Diverse Base Models:** Diversity in foundational architectures was an important factor. We aimed to represent at least two distinct base models to ensure a meaningful evaluation. Consequently, Bloomz, based on GPT-2, and Code Llama, stemming from Llama-2, were incorporated. Furthermore, to fine-tune our scope towards the context of our research, we sought models that underwent specialized training specifically on Java code.
3. **Varying Complexity Levels:** The complexity of a model often has a significant impact on the performance of a model. In light of this, where feasible, we selected multiple versions of a single model, differentiated by their parameter sizes. This strategy was selected to empirically discern if heightened model complexity would be advantageous for our specific task.

4.4 Results

In this section, we provide an examination of the experimental outcomes. Initially, we detail the evaluation metrics employed, encompassing metrics such as F1-Measure, Accuracy, and Precision, to ensure an unambiguous interpretation of the data. Subsequent to the metric descriptions, we systematically dissect the results as they relate to the different code characteristics assessed

[6] `https://bigscience.huggingface.co/`
[7] `https://ai.meta.com/`

ID	Model Name	Number of Parameters	Base Model	Max Input	Reference
M1	bigscience/bloomz-560m	560M	GPT-2	1024	[79]
M2	bigscience/bloomz-1b1	1.1B	GPT-2	1536	[79]
M3	bigscience/bloomz-1b7	1.7B	GPT-2	2048	[79]
M4	bigscience/bloomz-3b	3B	GPT-2	2560	[79]
M5	bigscience/bloomz-7b1	7.1B	GPT-2	4096	[79]
M6	microsoft/ CodeGPT-small-java-adaptedGPT2	124M	GPT-2	1024	-
M7	thmk/codegpt-java-10.2	124M	GPT-2	1024	-
M8	ammarnasr/ codegen-350M-mono-java	124M	GPT-2	1024	[80]
M9	codellama/ CodeLlama-7b-hf	7B	Llama 2	16384	[81]
M10	codellama/ CodeLlama-13b-hf	13B	Llama 2	16384	[81]

Table 4.2: Overview of the selected models. The column "Model Name" refers to the identification string in Hugging Face. "Max Input" describes the maximum number of tokens that the model can handle as input.

in the dataset by Schnappinger et al. [75], namely: Overall Maintainability, Readability, Understandability, Complexity, and Modularization. Through this structured presentation, we aim to offer a detailed overview of the models' performance within the designated quality parameters. As stated in section 4.3, when assessing the performance in the following sections, the performance metrics are not directly computed based on the output of the transformer models. We rather use the next token probability output of the transformer models when re-generating the code, compute the cross-entropy,

and then train a logistic regression classifier and a linear regression using the cross-entropy and different representations of the assessed characteristic (binary representation for logistic regression and average Likert score for the linear regression). We then determine the performance of those classifiers in predicting the characteristic under examination based on the cross-entropy and use that assessment to verify our hypothesis/research question.

4.4.1 Evaluation Metrics

Evaluation metrics are a crucial part of an experimental setup as they enable the objective evaluation of a model's performance and the comparability with other approaches. The following sections give a short definition and explanation of the metrics used in our experiments. Accuracy, Precision, Recall, F1-Measure, and Area Under Curve (AUC) are defined according to the definitions in chapter 3.1.3.

Accuracy

It is the measure of correct predictions out of the total predictions. If we have TP (True Positives), TN (True Negatives), FP (False Positives), and FN (False Negatives), then accuracy can be calculated as:

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

Precision

This is the measure of the relevancy of the obtained results. Precision can be interpreted as the likelihood that a positive prediction by the model is correct and can be calculated as:

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

Recall

Also known as sensitivity, hit rate, or true positive rate, recall is the measure of how many of the true positives were recalled (found), out of all the actual positives. It can be calculated as:

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

F1-Measure

The F1-Measure is the harmonic mean of Precision and Recall. It provides a single score that balances both the concerns of precision and recall. It is particularly useful in the uneven class distribution where the negative class might vastly outnumber the positive class. It can be calculated as:

$$F1\text{-}Measure = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Area Under Curve (AUC)

AUC stands for "Area under the ROC Curve" and can be interpreted as the probability that the model ranks a random positive example more highly than a random negative example. There does not exist a straightforward formula similar to the other metrics as it measures the entire two-dimensional area underneath the ROC (Receiver Operating Characteristic) Curve.

Mean Squared Error (MSE)

Mean Squared Error [82] is the average of the squared differences between the predicted and actual values. It serves as an indicator of an estimator's accuracy - it is always non-negative, and values closer to zero are better. MSE is commonly used in regression analysis, with a lower MSE indicating that a model has a better fit to the data. If y_i is the actual value, $\hat{y}_i$ the predicted value, and n the number of observations, MSE can be calculated as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Spearman Rank Correlation

It is a non-parametric measure of rank correlation and assesses how well a monotonic function can describe the relationship between two variables. With d_i being the difference between the ranks of corresponding variables and n the number of observations, the Spearman rank correlation [83] can be calculated as:

$$\rho = 1 - \frac{6 \sum {d_i}^2}{n(n^2 - 1)}$$

P-value

In hypothesis testing, the p-value [84] represents the likelihood of witnessing a test statistic as extreme as the one calculated under the assumption that the null hypothesis holds. It provides evidence against a null hypothesis. A small p-value (typically $p < 0.05$) indicates strong evidence against the null hypothesis.

4.4.2 Overall Maintainability

The fundamental cornerstone of the study, "Overall Maintainability", represents the holistic assessment of the code's manageability and potential long-term robustness and was evaluated by the experts in the evaluation dataset based on the statement "Overall, this code is maintainable" [74]. The results, as outlined in table 4.3, span a diverse range of Large Language Models and performance metrics.

Accuracy, a metric with high importance, suggests an intriguing spectrum of results. Most notably, the 'CodeGPT-java-10.2' model, denoted as M7, achieved the highest accuracy score at 0.74. This stands in sharp contrast to the 'CodeLlama-13b' model (M10) which had the lowest score of 0.44.

Precision (PR) for predicting high maintainability ranges from 0.32 (M10) to 0.67 (M7). Interestingly, when predicting low maintainability, the models exhibit an overall higher precision with a range from 0.53 (M10) to 0.79 (M7).

ID	Model	ACC	PR(P)	PR(N)	REC(P)	REC(N)	F1(P)	F1(N)	AUC	MSE	SRC	P-value
M1	Bloomz-560m	0.54	0.45	0.63	0.56	0.53	0.5	0.58	0.6	0.7932	-0.371	0
M2	Bloomz-1b1	0.56	0.47	0.67	0.64	0.5	0.54	0.57	0.58	0.7979	-0.3753	0
M3	Bloomz-1b7	0.57	0.49	0.69	0.68	0.5	0.57	0.58	0.58	0.7908	-0.4099	0
M4	Bloomz-3b	0.56	0.47	0.68	0.68	0.47	0.56	0.56	0.58	0.791	-0.3897	0
M5	Bloomz-7b1	0.54	0.46	0.65	0.64	0.47	0.53	0.55	0.5	0.8372	-0.1588	0.0055
M6	CodeGPT-small-java	0.59	0.5	0.69	0.64	0.56	0.56	0.62	0.65	0.7115	-0.4912	0
M7	CodeGPT-java-10.2	0.74	0.67	0.79	0.72	0.75	0.69	0.77	0.84	0.6106	0.6607	0
M8	codegen-350m	0.49	0.4	0.58	0.48	0.5	0.44	0.54	0.59	0.7877	-0.3479	0
M9	CodeLlama-7b	0.48	0.39	0.57	0.48	0.47	0.43	0.52	0.48	0.7986	-0.2023	0.0004
M10	CodeLlama-13b	0.44	0.32	0.53	0.32	0.53	0.32	0.53	0.43	0.8023	-0.1322	0.0211

Table 4.3: Results for Overall Maintainability. Abbreviations: ACC = Accuracy, PR = Precision, REC = Recall, F1 = F1-Measure, AUC = Area under Curve, MSE = Mean Squared Error, SRC = Spearman Rank Correlation. For Precision, Recall, and F1-Measure, the (P) indicates that the value is computed for the prediction of positive (high maintainability) samples and (N) indicates that the value is computed for the prediction of negative (low maintainability) samples.

The Recall (REC) and F1-Measure generally followed a similar trajectory but with Recall for high maintainability being generally higher than for low maintainability.

The Area Under Curve (AUC) values predominantly hovered between 0.5 and 0.84, with M7 showcasing the highest. A notable metric is the Spearman Rank Correlation (SRC), which provides insights into the rank order relationship between cross-entropy values of the LLMs and expert evaluations. While many models exhibited a negative correlation, indicating that higher cross-entropy might be associated with lower maintainability, M7 revealed a positive correlation (0.6607).

Intriguingly, while most models recorded a p-value of 0, indicating statistical significance in their correlations, models such as 'Bloomz-7b1' (M5) and 'CodeLlama-13b' (M10) presented p-values of 0.0055 and 0.0211 respectively. This necessitates a cautious interpretation of their Spearman rank correlations.

In summary, while the models offer a wide variance in performance, it becomes evident that specific configurations, like 'CodeGPT-java-10.2', are

better suited for predicting the overall maintainability of Java code based on cross-entropy than others. Possible implications and explanations for some of the more notable findings can be found in the discussion part (section 4.5).

4.4.3 Readability

Readability is paramount in software development, as it dictates how seamlessly developers can understand and modify the code. It directly impacts the overall maintainability of code, facilitating more effortless navigation, debugging, and modification processes. The results for readability, as showcased in table 4.4, allow us to gauge the proficiency of the models in discerning this important aspect.

ID	Model	ACC	PR(P)	PR(N)	REC(P)	REC(N)	F1(P)	F1(N)	AUC	MSE	SRC	P-value
M1	Bloomz-560m	0.59	0.43	0.73	0.57	0.6	0.49	0.66	0.67	0.3676	-0.2203	0.0001
M2	Bloomz-1b1	0.59	0.43	0.74	0.62	0.57	0.51	0.65	0.64	0.3696	-0.2154	0.0002
M3	Bloomz-1b7	0.59	0.43	0.73	0.57	0.6	0.49	0.66	0.66	0.3646	-0.2582	0
M4	Bloomz-3b	0.57	0.41	0.72	0.57	0.57	0.48	0.64	0.65	0.3634	-0.2513	0
M5	Bloomz-7b1	0.52	0.3	0.63	0.29	0.65	0.29	0.64	0.55	0.3805	-0.0645	0.2626
M6	CodeGPT-small-java	0.72	0.58	0.81	0.67	0.75	0.62	0.78	0.75	0.3505	-0.4047	0
M7	CodeGPT-java-10.2	0.66	0.5	0.76	0.57	0.7	0.53	0.73	0.75	0.2907	0.5733	0
M8	codegen-350m	0.61	0.44	0.74	0.57	0.62	0.5	0.68	0.67	0.3663	-0.1996	0.0005
M9	CodeLlama-7b	0.49	0.19	0.6	0.14	0.68	0.16	0.64	0.55	0.3729	-0.1226	0.0326
M10	CodeLlama-13b	0.54	0.18	0.62	0.1	0.78	0.12	0.69	0.51	0.379	-0.0855	0.1371

Table 4.4: Results for Readability. Abbreviations: ACC = Accuracy, PR = Precision, REC = Recall, F1 = F1-Measure, AUC = Area under Curve, MSE = Mean Squared Error, SRC = Spearman Rank Correlation. For Precision, Recall, and F1-Measure, the (P) indicates that the value is computed for the prediction of positive (high readability) samples and (N) indicates that the value is computed for the prediction of negative (low readability) samples.

From an accuracy standpoint, the 'CodeGPT-small-java' model, denoted as M6, surfaced as the leading performer with an accuracy of 0.72, while 'CodeLlama-7b' (M9) lagged behind the other models, marking the lowest accuracy of 0.49.

In terms of precision for high readability predictions, the range spans from 0.18 (M10) to 0.58 (M6). For low readability predictions, the span

is more compact, ranging from 0.6 (M9) to 0.81 (M6). The Recall and F1-Measure exhibited diverse outcomes across the models, the 'CodeGPT-small-java' (M6) maintaining its upper hand, particularly when predicting high readability.

The Area Under Curve exhibited values largely hovering between the range of 0.51 to 0.75. The 'CodeGPT-java-10.2' model (M7) notably possessed the maximum AUC value of 0.75, tying with 'CodeGPT-small-java' (M6).

Regarding the Spearman Rank Correlation, it is worth noting that the findings match with the results for "Overall Maintainability". Most models displayed a negative correlation, suggesting that increased cross-entropy could correspond with decreased readability, with 'CodeGPT-java-10.2' (M7) being an outlier again, with a positive correlation of 0.5733. P-values of models, with a few exceptions 'Bloomz-7b1' (M5) and both CodeLlama models (M9, M10), were predominantly close to 0, emphasizing the statistical significance of their correlations.

In essence, the results reveal that while various models exhibit a broad spectrum of performance metrics, models such as 'CodeGPT-small-java' appear particularly adept at predicting readability based on cross-entropy.

4.4.4 Understandability

Transitioning to 'Understandability', our analysis is anchored around the assertion: "The semantic meaning of this code is clear." ([74], Chapter 3). We aim to ascertain the viability of cross-entropy as a predictor for the clear conveyance of code semantics, further underscoring its positive bearing on maintainability. Table 4.5 provides insights into the ability of various models to discern code's understandability.

Beginning with accuracy, 'CodeGPT-java-10.2' (M7) stands out with a value of 0.74, leading the table. On the opposite end of the spectrum, both CodeLlama models (M9, M10) secure the lower ranks with accuracies of 0.46 and 0.48 respectively.

Considering precision for high understandability predictions, the spread

ID	Model	ACC	PR(P)	PR(N)	REC(P)	REC(N)	F1(P)	F1(N)	AUC	MSE	SRC	P-value
M1	Bloomz-560m	0.61	0.55	0.71	0.79	0.45	0.65	0.56	0.55	0.7949	-0.3351	0
M2	Bloomz-1b1	0.62	0.56	0.78	0.86	0.42	0.68	0.55	0.51	0.807	-0.3335	0
M3	Bloomz-1b7	0.64	0.57	0.79	0.86	0.45	0.69	0.58	0.54	0.7937	-0.3696	0
M4	Bloomz-3b	0.64	0.57	0.79	0.86	0.45	0.69	0.58	0.53	0.7945	-0.3558	0
M5	Bloomz-7b1	0.57	0.52	0.71	0.82	0.36	0.64	0.48	0.5	0.8405	-0.1603	0.0051
M6	CodeGPT-small-java	0.57	0.53	0.65	0.71	0.45	0.61	0.54	0.63	0.7439	-0.4893	0
M7	CodeGPT-java-10.2	0.74	0.7	0.77	0.75	0.73	0.72	0.75	0.8	0.6657	0.6115	0
M8	codegen-350m	0.61	0.55	0.76	0.86	0.39	0.67	0.52	0.56	0.7928	-0.3222	0
M9	CodeLlama-7b	0.48	0.45	0.52	0.64	0.33	0.53	0.41	0.43	0.8275	-0.1956	0.0006
M10	CodeLlama-13b	0.46	0.44	0.5	0.61	0.33	0.51	0.4	0.39	0.8367	-0.1262	0.0278

Table 4.5: Results for Understandability. Abbreviations: ACC = Accuracy, PR = Precision, REC = Recall, F1 = F1-Measure, AUC = Area under Curve, MSE = Mean Squared Error, SRC = Spearman Rank Correlation. For Precision, Recall, and F1-Measure, the (P) indicates that the value is computed for the prediction of positive (high understandability) samples and (N) indicates that the value is computed for the prediction of negative (low understandability) samples.

is from 0.44 (M10) to 0.7 (M7). Meanwhile, for low understandability, values are predominantly distributed between 0.5 and 0.79, showcasing the general capability of models in predicting lower understandability.

The recall metrics further highlight some intriguing patterns. For high understandability predictions, three of the Bloomz models (M2, M3, M4) share a similar top score of 0.86. Conversely, 'CodeLlama-7b' (M9) and 'CodeLlama-13b' (M10) again exhibit lower values.

Delving into the F1-Measure, 'CodeGPT-java-10.2' (M7) displays a balanced performance for both high and low understandability, with scores of 0.72 and 0.75, respectively.

The AUC metric underscores the models' performance concerning the true positive rate versus false positive rate. 'CodeGPT-java-10.2' (M7) takes the lead again with an AUC of 0.8, while 'CodeLlama-13b' (M10) trails with 0.39.

Matching the findings from the previous sections, almost all models exhibit a negative Spearman Rank Correlation, suggesting that higher cross-entropy could indicate lower understandability. Again, the one exception to

this is 'CodeGPT-java-10.2' (M7) with a positive correlation.

Lastly, regarding p-values, the majority of models feature values close to 0, indicating statistically significant correlations. However, exceptions such as 'Bloomz-7b1' (M5) and the two CodeLlama models (M9, M10) possess marginally higher p-values, yet still fall within the general threshold for significance.

To encapsulate, the presented results unveil 'CodeGPT-java-10.2' as a potential frontrunner in predicting understandability, though other models also exhibit noteworthy competencies.

4.4.5 Complexity

Complexity in code is a metric that gauges the intricacy and potential understandability challenges a piece of software might present. It is pivotal for developers to have a handle on the complexity as it can influence the potential for defects, which can have a negative impact on maintainability. In table 4.6, we turn our attention to the results indicative of various models' efficacy in determining code complexity.

ID	Model	ACC	PR(P)	PR(N)	REC(P)	REC(N)	F1(P)	F1(N)	AUC	MSE	SRC	P-value
M1	Bloomz-560m	0.54	0.7	0.39	0.53	0.57	0.6	0.46	0.6	0.7156	0.3302	0
M2	Bloomz-1b1	0.52	0.69	0.38	0.5	0.57	0.58	0.45	0.57	0.7211	0.3302	0
M3	Bloomz-1b7	0.57	0.75	0.42	0.53	0.67	0.62	0.52	0.59	0.7164	0.3686	0
M4	Bloomz-3b	0.57	0.77	0.43	0.5	0.71	0.61	0.54	0.59	0.7145	0.3615	0
M5	Bloomz-7b1	0.52	0.7	0.38	0.47	0.62	0.57	0.47	0.55	0.7497	0.1837	0.0013
M6	CodeGPT-small-java	0.62	0.77	0.47	0.6	0.67	0.68	0.55	0.63	0.6656	0.4396	0
M7	CodeGPT-java-10.2	0.7	0.81	0.56	0.72	0.67	0.76	0.61	0.77	0.6487	-0.5704	0
M8	codegen-350m	0.54	0.69	0.38	0.55	0.52	0.61	0.44	0.61	0.7051	0.3492	0
M9	CodeLlama-7b	0.57	0.73	0.42	0.55	0.62	0.63	0.5	0.54	0.716	0.2447	0
M10	CodeLlama-13b	0.52	0.67	0.36	0.55	0.48	0.6	0.41	0.49	0.7261	0.1763	0.002

Table 4.6: Results for Complexity. Abbreviations: ACC = Accuracy, PR = Precision, REC = Recall, F1 = F1-Measure, AUC = Area under Curve, MSE = Mean Squared Error, SRC = Spearman Rank Correlation. For Precision, Recall, and F1-Measure, the (P) indicates that the value is computed for the prediction of positive (high complexity) samples and (N) indicates that the value is computed for the prediction of negative (low complexity) samples.

In terms of accuracy, 'CodeGPT-java-10.2' (M7) takes the lead with a score of 0.7. This is closely followed by 'CodeGPT-small-java' (M6) at 0.62. Conversely, 'Bloomz-1b1' (M2) and 'Bloomz-7b1' (M5) are at the lower end with scores of 0.52.

For precision concerning high complexity predictions, 'CodeGPT-java-10.2' (M7) shines with a value of 0.81. In the context of low complexity, there is a tighter spread among the models ranging from 0.36 (M10) and 0.56 (M7).

When analyzing recall for high complexity, 'CodeGPT-java-10.2' (M7) maintains its superior performance with 0.72. As for low complexity, 'Bloomz-3b' (M4) emerges as a significant outlier with a commendable score of 0.71.

In the domain of the F1-Measure, 'CodeGPT-java-10.2' (M7) continues to set the benchmark with 0.76, indicating its balanced efficiency in complexity prediction.

Concerning AUC, 'CodeGPT-java-10.2' (M7) yet again leads with 0.77, implying its capability to distinguish between high and low complexity samples effectively. On the contrary, 'CodeLlama-13b' (M10) logs the lowest AUC at 0.49.

A majority of models show a positive Spearman Rank Correlation, suggesting a tendency for higher cross-entropy values with increased complexity. This time, 'CodeGPT-java-10.2' (M7) displays a negative correlation, continuing its deviation from the pattern.

Lastly, p-values across the board mostly hover around 0. Exceptions like 'Bloomz-7b1' (M5) and 'CodeLlama-13b' (M10) have marginally higher p-values but still maintain significance.

In conclusion, 'CodeGPT-java-10.2' consistently exhibits superior performance across several metrics, establishing its robustness in predicting code complexity. While many models show adeptness in this domain, the results also shed light on areas of potential refinement. Understanding these nuances aids in streamlining model selection based on specific complexity-related objectives.

4.4.6 Modularization

Concluding with 'Modularization', the evaluation statement pivots to "This code should be split up into smaller pieces." ([74], Chapter 3). Traditionally, modularity's overarching principles might not seamlessly align with isolated class-level code snippets. However, in the context of the assessment, it embodies the essence of appropriate sizing on a class level as well as a method level. Here, in table 4.7, we delve into the performance of different models in predicting the need for modularization.

ID	Model	ACC	PR(P)	PR(N)	REC(P)	REC(N)	F1(P)	F1(N)	AUC	MSE	SRC	P-value
M1	Bloomz-560m	0.56	0.71	0.35	0.6	0.47	0.65	0.4	0.64	0.7161	0.3143	0
M2	Bloomz-1b1	0.54	0.72	0.34	0.55	0.53	0.62	0.42	0.64	0.7135	0.3262	0
M3	Bloomz-1b7	0.54	0.71	0.33	0.57	0.47	0.63	0.39	0.65	0.7101	0.3575	0
M4	Bloomz-3b	0.54	0.72	0.34	0.55	0.53	0.62	0.42	0.65	0.7154	0.3424	0
M5	Bloomz-7b1	0.57	0.69	0.32	0.69	0.32	0.69	0.32	0.55	0.7556	0.1102	0.0548
M6	CodeGPT-small-java	0.67	0.76	0.47	0.76	0.47	0.76	0.47	0.75	0.6325	0.4286	0
M7	CodeGPT-java-10.2	0.77	0.87	0.61	0.79	0.74	0.82	0.67	0.87	0.647	-0.6373	0
M8	codegen-350m	0.61	0.74	0.39	0.67	0.47	0.7	0.43	0.64	0.707	0.3068	0
M9	CodeLlama-7b	0.54	0.65	0.15	0.74	0.11	0.69	0.12	0.57	0.7109	0.1918	0.0008
M10	CodeLlama-13b	0.57	0.66	0.18	0.79	0.11	0.72	0.13	0.53	0.7191	0.1275	0.0262

Table 4.7: Results for Modularization. Abbreviations: ACC = Accuracy, PR = Precision, REC = Recall, F1 = F1-Measure, AUC = Area under Curve, MSE = Mean Squared Error, SRC = Spearman Rank Correlation. For Precision, Recall, and F1-Measure, the (P) indicates that the value is computed for the prediction of positive (high modularization) samples and (N) indicates that the value is computed for the prediction of negative (low modularization) samples.

At the outset, 'CodeGPT-java-10.2' (M7) leads the pack in accuracy, registering a strong 0.77. This model also showcases impressive precision for high modularization with a score of 0.87. Meanwhile, for predictions of low modularization, the same model (M7) achieves 0.61, far surpassing others, with 'CodeLlama-7b' (M9) at the other end of the spectrum with a mere 0.15.

In terms of recall for high modularization, 'CodeLlama-13b' (M10) shows prowess with a score of 0.79, matching the performance of 'CodeGPT-java-10.2' (M7).

Evaluating the F1-Measure, we again find 'CodeGPT-java-10.2' (M7) at the forefront for high modularization predictions with a score of 0.82. On the other side, both CodeLlama models (M9, M10) demonstrate the lowest F1-Measures for low modularization, indicating some challenges in this specific aspect.

'CodeGPT-java-10.2' (M7) outperforms others with an AUC of 0.87, suggesting superior capability in discerning modularization levels.

The trend for the Spearman Rank Correlation continues, with most models showcasing a positive correlation, while 'CodeGPT-java-10.2' (M7) sharply deviates with a negative correlation.

The p-values largely indicate statistical significance with values close to 0. However, 'Bloomz-7b1' (M5) at 0.0548 and 'CodeLlama-13b' (M10) at 0.0262, while significant, hover on the margin, hinting a reduced confidence in those particular results.

In summation, 'CodeGPT-java-10.2' (M7) emerges as the frontrunner in predicting modularization, demonstrating consistent superiority across multiple metrics. However, its consistent deviation in the Spearman Rank Correlation value across all characteristics indicates the need for further investigation, which we will aim to do in the discussion part. As always, understanding specific model strengths is pivotal when selecting tools for assessing modularization in various contexts.

4.4.7 Additional Details

Expanding upon our primary evaluations, this section provides additional insights into the experiment results, including the runtime of the various models and details about the decision boundaries for the logistic classifiers concerning each model and characteristic. As a foundational concept, cross-entropy measures the difference between two probability distributions, serving as a pivotal metric for classifiers' decision boundaries. Here, the boundary values represent the threshold at which the model switches its prediction from positive to negative or vice-versa.

Table 4.8 lays out the runtimes for each model, alongside the decision

boundary cross-entropy values for overall maintainability as well as the other characteristics.

ID	Model	Runtime (mm:ss)	Overall Boundary	Readability Boundary	Understandability Boundary	Complexity Boundary	Modularization Boundary
M1	Bloomz-560m	00:50	1.48	1.38	1.65	1.45	1.32
M2	Bloomz-1b1	01:21	1.21	1.12	1.37	1.18	1.07
M3	Bloomz-1b7	02:05	1.20	1.12	1.35	1.17	1.07
M4	Bloomz-3b	03:52	1.13	1.04	1.29	1.11	1
M5	Bloomz-7b1	08:00	0.97	0.72	1.19	0.94	0.7
M6	CodeGPT-small-java	00:16	3.93	3.79	4.17	3.86	3.65
M7	CodeGPT-java-10.2	00:14	3.31	3.37	3.13	3.34	3.47
M8	codegen-350m	00:49	1.20	1.09	1.35	1.17	1.05
M9	CodeLlama-7b	07:43	0.53	0.43	0.66	0.53	0.41
M10	CodeLlama-13b	14:21	0.44	0.33	0.58	0.46	0.31

Table 4.8: Additional Results. The boundaries correspond to the cross-entropy value of the decision boundaries of the classifier, so the point at which the classification switches from positive to negative prediction or vice versa.

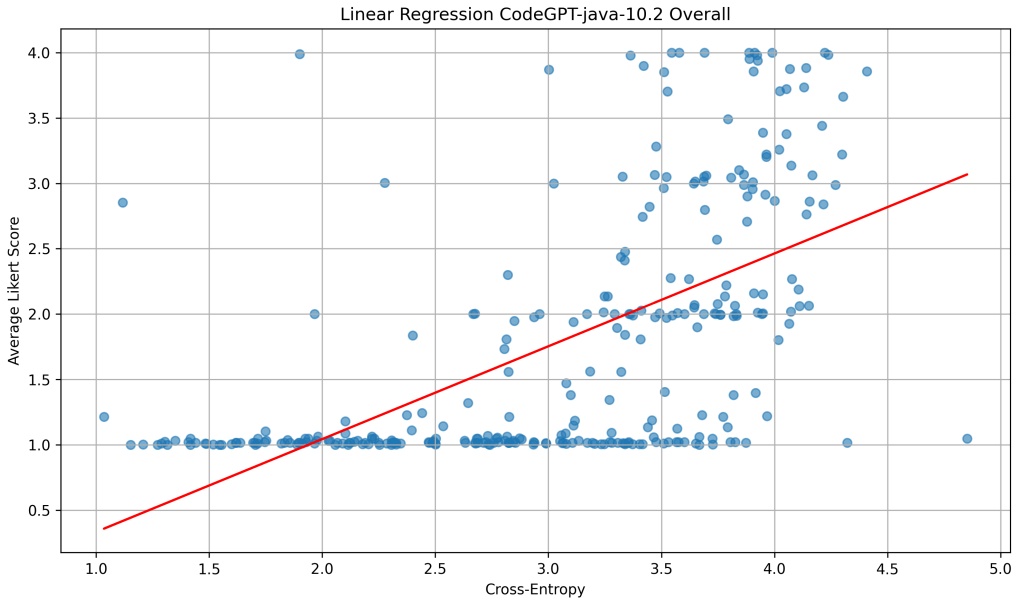

Figure 4.2: Visualization of the Linear Regression for the model 'CodeGPT-java-10.2' (M7) and characteristic 'Overall Maintainability'.

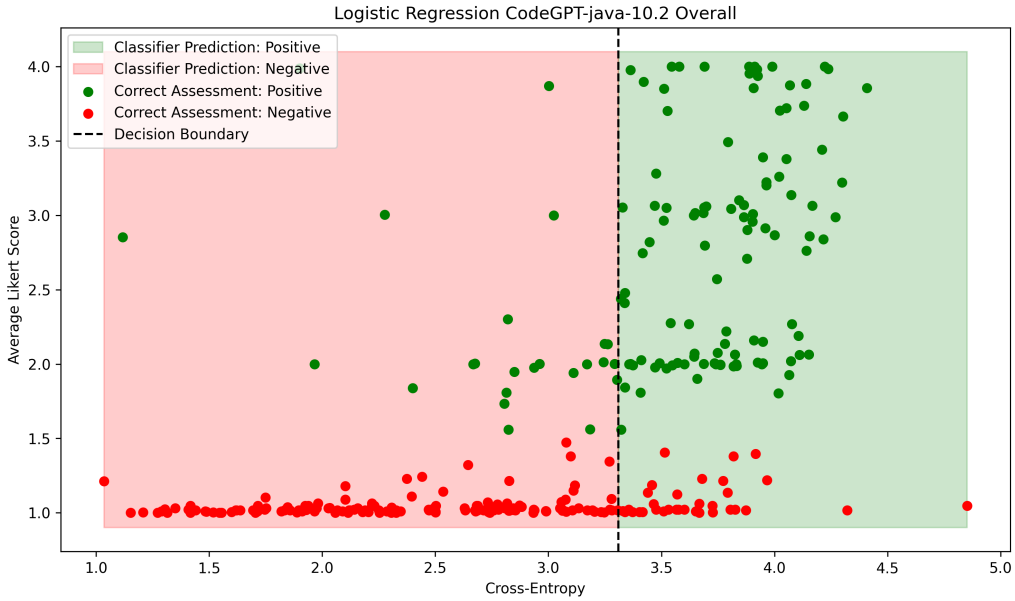

Figure 4.3: Visualization of the Logistic Regression Classifier for the model 'CodeGPT-java-10.2' (M7) and characteristic 'Overall Maintainability'. If the color of a dot and the region color match, the classifier prediction for that sample is correct. If the colors do not match, the classifier prediction is wrong.

From a runtime perspective, we observe a stark contrast between models. While 'CodeGPT-java-10.2' (M7) is exceedingly quick at just 14 seconds, 'CodeLlama-13b' (M10) requires a substantially more extended period, clocking in at 14 minutes and 21 seconds, mirroring the difference in model size. Such differences underscore the efficiency variations in computational complexity and processing power among models with different levels of complexity and size.

Upon examining the decision boundaries, the 'CodeGPT' models (M6, M7) exhibit significantly higher cross-entropy values across all characteristics compared to others. Conversely, the 'CodeLlama' series, especially 'CodeLlama-13b' (M10), showcases the lowest boundary values, hinting at a tighter classification regime.

To better comprehend these values and the results presented in the last sections, we present exemplary visualizations of a logistic regression classifier

(Figure 4.3) and a linear regression (Figure 4.2), illustrating how these classifiers manifest in practical terms. These plots offer an intuitive grasp of the models' decision-making processes, facilitating better interpretations of the data. For a comprehensive visual exploration, the complete set of plots for all models and characteristics is available on GitHub[8] in the folder "plots".

Table 4.8 is thus an important part of grasping the efficiency and decision-making nuances of the discussed models. By showcasing the runtime and decision boundary metrics, researchers and practitioners can get a more rounded perspective, which is vital when opting for models in varying operational contexts.

4.5 Discussion

Delving deeper into the nuanced findings of the previous sections, there is an intricate landscape of insights that emerge about the various models and their performance on distinct code characteristics, including overall maintainability, readability, understandability, complexity, and modularization.

For starters, the complexity of models had a profound implication for their prediction capability. Contrary to the conventional belief, as models become more sophisticated, they also become better attuned to the intricate details of the code. This results in a discernible decline in overall cross-entropy. However, a more subtle nuance gets introduced: as the distinction between high-quality and low-quality code begins to blur, the classifiers trained on the computed cross-entropy find it more challenging to predict maintainability. If the variance is substantial, as in a cross-entropy range between 2 and 10 for example, the distinction is clearer. But if the range narrows, say between 0.8 and 1.3, the decision-making becomes notably intricate. Combined with the fact that more complex models also need more computational resources, our findings suggest that using less complex models, fine-tuned on the specific programming language, is beneficial for predicting maintainability as well as its sub-characteristics. A simpler model, such as GPT-2 for example,

[8] `https://github.com/marcdillmann/DeepCodeMaintainability`

typically results in a greater overall cross-entropy and a broader range since it is not as adept at adjusting to specific details. Even when this model is fine-tuned for a particular programming language, like Java in our case, it still exhibits higher cross-entropy and range compared to more complex models. However, due to the fine-tuning, the cross-entropy becomes more indicative of code quality. This makes the cross-entropy of such a model a more effective foundation for training the final classifier as well as predicting the quality of unseen code.

Drawing from the result tables, it is evident that models, in general, have a stronger aptitude for identifying code with low maintainability than singling out highly maintainable code. This trend also continues for attributes that inherently boost maintainability, such as readability and understandability, and is reversed for characteristics that hurt maintainability like complexity and the need for modularization. This suggests that our approach might be more adept as a tool to flag potentially problematic code, rather than certifying code snippets as maintainable. This is reminiscent of the findings from Sengamedu et al. which primarily aimed at pinpointing detrimental code characteristics like long and complex code.

As already hinted in the results section, 'CodeGPT-java-10.2' needs particular attention as it emerges as an intriguing outlier. While it boasts superior performance metrics, it also manifests a correlation between the cross-entropy and the predicted characteristic that is reversed compared to all other models. Unfortunately, we were not able to identify the dataset that the model was trained on, so the following considerations need to be taken with some level of uncertainty. Because of this uncertainty, we even considered excluding it from our experiments, but because of the impressive performance and the potential indications for future works, we decided to include it and critically examine it. One plausible conjecture to explain this anomaly is that the model was potentially fine-tuned on "bad" or low maintainability code. Hence, when the model finds it easier to predict a piece of code (evidenced by a lower cross-entropy), it could be indicative of that code possessing low maintainability attributes.

One perspective that arises from the performance of 'CodeGPT-java-10.2'

is a potential direction for future endeavors. Considering its superior performance metrics, it might be worthwhile to intentionally train models on code with low maintainability. Such a strategy could harness the observed tendencies of these models and refine them into tools exceptionally adept at identifying subpar code.

Another key insight from our findings is that while some models perform well across various characteristics, no single model excels in every aspect. If factors beyond overall maintainability are crucial in the evaluation, it might be beneficial to integrate multiple models, each tailored to specific characteristics, into a prospective tool. Broadly, this research can lay the groundwork for a future tool. The development process would involve selecting an appropriate model (like 'CodeGPT-java-10.2' based on our results, or another model either identified or specifically fine-tuned in subsequent studies), training a classifier using the evaluation dataset, and then incorporating this classifier into the tool as the final layer. As previously mentioned, this tool's primary application could be to pinpoint code with subpar maintainability.

After discussing and analyzing various details of our approach and important facets of our experimental results, performance analysis, and comparison are equally important to assess the feasibility of our approach. Within the ambit of code maintainability prediction, a recurring challenge has been the relative scarcity of comprehensive approaches, as corroborated by our literature survey in chapter 3. This deficiency has rendered any rigorous comparison somewhat challenging. Nevertheless, for the sake of a holistic evaluation, we anchored our comparison against three key studies that hold relevance in this domain.

Firstly, the work by Schnappinger et al. [73] merits attention. They embarked on a multifaceted approach, deploying and assessing BERT-based models, traditional machine-learning models, and even image-based ones. While their BERT models achieved an F1-Measure of up to 0.68, it was the traditional SVM model that was assessed to be the most effective with an F1-Measure of up to 0.827. Secondly, Hu et al. [69] crafted a sophisticated framework, integrating multiple attention-based dense networks, a spectrum of LSTMs, and a dense network. Their method focused on a gamut of ex-

tracted features, culminating in a unified maintainability index. The inclusion of code metrics led to their omission from our prior literature study. Instead of conventional metrics like F1-Measure, they gauged performance based on pair comparisons, discerning the maintainability order within each pair, and clocked an average accuracy of 0.875. Lastly, our foundational study by Sengamedu et al. [72] achieved an average accuracy of 0.664, although they abstained from computing the F1-Measure.

Given the disparate methodologies and datasets employed, direct comparisons do become difficult. However, some inferences can still be distilled. While our approach's best F1-Measure surpasses the deep learning models deployed by both Schnappinger et al. and Sengamedu et al., the framework proposed by Hu et al. seems to hold an edge. But context is crucial. Their multi-faceted deep learning structure, substantiated by a plethora of extracted features, demands significant preprocessing overhead. In contrast, our methodology stands out due to its minimalistic and streamlined approach. The absence of almost no prerequisite preprocessing, combined with our competitive performance metrics, underscores the potency and efficiency of our approach for predicting code maintainability.

To wrap up, we were able to confirm our research question and show the feasibility of using the cross-entropy computed based on the next-token probabilities from text-generating transformer-based large language models as an indicator for code maintainability and its sub-characteristics. The interplay of model complexity, their prediction abilities on diverse code attributes, and the subtle influences of the training data offer profound insights. While the immediate results offer valuable perspectives for the developer community, they also chart out intriguing avenues for upcoming research and innovation in the domain of code maintainability assessment.

Chapter 5

Conclusion

In the realm of software engineering research, the evaluation of code quality, particularly maintainability, remains a pivotal area of interest. This thesis embarked on a systematic exploration into the applicability of deep learning models for this purpose. Through an exhaustive literature review, a discernible gap was identified pertaining to the utilization of deep learning models for maintainability prediction. This observation delineated the trajectory of our subsequent research endeavors.

The experimental methodology adopted in this study was characterized by its innovative departure from conventional techniques. Eschewing the often cumbersome preprocessing or feature extraction processes inherent to many code quality evaluation methodologies, we capitalized on the next-token probability intrinsic to transformer-based large language models. By quantifying the cross-entropy between the source code and the model's predictions, we sought to prognosticate maintainability and its associated subcharacteristics. The salient feature of this methodology is its inherent simplicity, obviating the prerequisite for code preprocessing, a frequent impediment in many extant techniques.

The empirical outcomes of our experiments were both illuminating and encouraging. The feasibility of the proposed approach was established, and its efficacy was benchmarked with competitive performance metrics in this domain. This achievement is accentuated by the streamlined nature of our

methodology, which contrasts markedly with the multifaceted preprocessing and feature extraction paradigms prevalent in the field.

In conclusion, this thesis has provided a structured investigation into an evolving domain. The findings presented have not only contributed to the academic understanding of the field but also have pragmatic implications for software engineering practices.

5.1 Future Work

Drawing from the insights unearthed during the experimental discussions, a few potential avenues for future research emerge:

1. **Focused Model Training:** Given the observed relationship between prediction ease and code quality in models like 'CodeGPT-java-10.2', there is a rationale to purposefully train models on 'bad' or low maintainable code. Such a strategic skew in the training dataset might further accentuate the models' ability to discern and flag low maintainability code segments.
2. **Complex Model Analysis:** While our results hint that increased model complexity does not always correlate with improved predictions, it would be interesting to discern the optimal balance. Are there sweet spots in model complexity that yield the best results?
3. **Expanding the Dataset:** While our experiments leveraged a comprehensive dataset, expanding this further might yield even more nuanced insights. Diverse codebases from varying industries and domains could help in refining our approach further.
4. **Building a Framework:** Our approach and implementation only serve as a proof-of-concept and an evaluation of the theoretical basics of the core idea. The next step could be to choose a specific model based on the assessed performance, train a classifier specifically for that model based on the evaluation dataset, and then package those into a framework.

List of Figures

Bibliography

[1] ISO/IEC 25010, "ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models," 2011.

[2] N. E. Fenton and S. L. Pfleeger, *Software metrics: a rigorous and practical approach*. Boston: PWS Publ. Comp, 2. ed., rev. print ed., 2003.

[3] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[4] M. H. Halstead, *Elements of software science*. No. 2 in Operating and programming systems series, New York: North Holland, 3. print ed., 1979.

[5] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[6] B. Mahesh, "Machine learning algorithms -a review," 01 2019.

[7] W. Zhang, G. Yang, Y. Lin, C. Ji, and M. M. Gupta, "On definition of deep learning," in *2018 World Automation Congress (WAC)*, pp. 1–5, 2018.

[8] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6, 2017.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[11] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm networks," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 4, pp. 2047–2052 vol. 4, 2005.

[12] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," 2014.

[13] A. Sherstinsky, "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, mar 2020.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," 2017. Publisher: arXiv Version Number: 7.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," May 2019. arXiv:1810.04805 [cs].

[16] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," Sept. 2023. arXiv:1910.10683 [cs, stat].

[17] G. Fischer, J. Lusiardi, and J. Wolff von Gudenberg, "Abstract syntax trees - and their role in model driven software development," in *Inter-*

national Conference on Software Engineering Advances (ICSEA 2007), pp. 38–38, 2007.

[18] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, p. 1–19, jul 1970.

[19] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[20] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," 2017.

[21] M. Xu, "Understanding graph embedding methods and their applications," 2020.

[22] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014.

[23] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," tech. rep., Technical report, EBSE Technical Report EBSE-2007-01, 2007.

[24] Y. Yin, Y. Shi, Y. Zhao, and F. Wahab, "Multi-graph learning-based software defect location," *Journal of Software: Evolution and Process*, p. e2552, Mar. 2023.

[25] S. Qiu, H. Huang, W. Jiang, F. Zhang, and W. Zhou, "Defect Prediction via Tree-Based Encoding with Hybrid Granularity for Software Sustainability," *IEEE Transactions on Sustainable Computing*, pp. 1–12, 2023.

[26] Z. Tian, B. Tian, and J. Lv, "Combining AST Segmentation and Deep Semantic Extraction for Function Level Vulnerability Detection," in *Advances in Natural Computation, Fuzzy Systems and Knowledge Discovery* (N. Xiong, M. Li, K. Li, Z. Xiao, L. Liao, and L. Wang, eds.), vol. 153, pp. 93–100, Cham: Springer International Publishing, 2023. Series Title: Lecture Notes on Data Engineering and Communications Technologies.

[27] C. Pornprasit and C. K. Tantithamthavorn, "DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction," *IEEE Transactions on Software Engineering*, vol. 49, pp. 84–98, Jan. 2023.

[28] F. Qiu, Z. Gao, X. Xia, D. Lo, J. Grundy, and X. Wang, "Deep Just-In-Time Defect Localization," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.

[29] C. Mamede, E. Pinconschi, and R. Abreu, "A transformer-based IDE plugin for vulnerability detection," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, (Rochester MI USA), pp. 1–4, ACM, Oct. 2022.

[30] Y. Xing, X. Qian, Y. Guan, B. Yang, and Y. Zhang, "Cross-project defect prediction based on G-LSTM model," *Pattern Recognition Letters*, vol. 160, pp. 50–57, Aug. 2022.

[31] X. Cheng, G. Zhang, H. Wang, and Y. Sui, "Path-sensitive code embedding via contrastive learning for software vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, (Virtual South Korea), pp. 519–531, ACM, July 2022.

[32] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-Aware Code Change Embedding for Better Patch Correctness Assessment," *ACM Transactions on Software Engineering and Methodology*, vol. 31, pp. 1–29, July 2022.

[33] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python," *Information and Software Technology*, vol. 144, p. 106809, Apr. 2022.

[34] B. Qi, H. Sun, W. Yuan, H. Zhang, and X. Meng, "DreamLoc: A Deep Relevance Matching-Based Framework for bug Localization," *IEEE Transactions on Reliability*, vol. 71, pp. 235–249, Mar. 2022.

[35] S. Qiu, S. Wang, X. Tian, M. Huang, and Q. Huang, "Visualization-Based Software Defect Prediction via Convolutional Neural Network with Global Self-Attention," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, (Guangzhou, China), pp. 189–198, IEEE, Dec. 2022.

[36] K. Yang, P. Miller, and J. Martinez-Del-Rincon, "Convolutional Neural Network for Software Vulnerability Detection," in *2022 Cyber Research Conference - Ireland (Cyber-RCI)*, (Galway, Ireland), pp. 1–4, IEEE, Apr. 2022.

[37] W. Chen, T. Vanderbruggen, P.-H. Lin, C. Liao, and M. Emani, "Early Experience with Transformer-Based Similarity Analysis for DataRaceBench," in *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*, (Dallas, TX, USA), pp. 45–53, IEEE, Nov. 2022.

[38] A. T. Elbosaty, W. M. Abdelmoez, and E. Elfakharany, "Within-Project Defect Prediction Using Improved CNN Model via Extracting the Source Code Features," in *2022 International Arab Conference on Information Technology (ACIT)*, (Abu Dhabi, United Arab Emirates), pp. 1–8, IEEE, Nov. 2022.

[39] Y. Zhou, C. Zhang, K. Jia, D. Zhao, and J. Xiang, "A Software Aging-Related Bug Prediction Framework Based on Deep Learning and Weakly Supervised Oversampling," in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, (Charlotte, NC, USA), pp. 185–192, IEEE, Oct. 2022.

[40] K. Kant Sharma, A. Sinha, and A. Sharma, "Software Defect Prediction using Deep Learning by Correlation Clustering of Testing Metrics," *International journal of electrical and computer engineering systems*, vol. 13, pp. 953–960, Dec. 2022.

[41] C. Zhu, G. Du, T. Wu, N. Cui, L. Chen, and G. Shi, "BERT-Based Vulnerability Type Identification with Effective Program Representation,"

in *Wireless Algorithms, Systems, and Applications* (L. Wang, M. Segal, J. Chen, and T. Qiu, eds.), vol. 13471, pp. 271–282, Cham: Springer Nature Switzerland, 2022. Series Title: Lecture Notes in Computer Science.

[42] M. Zhang and J. Jia, "Feature Envy Detection with Deep Learning and Snapshot Ensemble," in *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*, (Wulumuqi, China), pp. 215–223, IEEE, Aug. 2022.

[43] H. Hanif and S. Maffeis, "VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection," in *2022 International Joint Conference on Neural Networks (IJCNN)*, (Padua, Italy), pp. 1–8, IEEE, July 2022.

[44] Y. Li and X. Zhang, "Multi-Label Code Smell Detection with Hybrid Model based on Deep Learning," pp. 42–47, July 2022.

[45] S. Zhao, C. Shi, S. Ren, and H. Mohsin, "Correlation Feature Mining Model Based on Dual Attention for Feature Envy Detection," pp. 634–639, July 2022.

[46] G. Yan, S. Chen, Y. Bail, and X. Li, "Can Deep Learning Models Learn the Vulnerable Patterns for Vulnerability Detection?," in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, (Los Alamitos, CA, USA), pp. 904–913, IEEE, June 2022.

[47] V. Cochard, D. Pfammatter, C. T. Duong, and M. Humbert, "Investigating Graph Embedding Methods for Cross-Platform Binary Code Similarity Detection," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, (Genoa, Italy), pp. 60–73, IEEE, June 2022.

[48] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with BERT," in *Proceedings of the 44th International Conference on Software Engineering*, (Pittsburgh Pennsylvania), pp. 946–957, ACM, May 2022.

[49] Y. Chen, C. Xu, J. Selena He, S. Xiao, and F. Shen, "Compiler IR-Based Program Encoding Method for Software Defect Prediction," *Computers, Materials & Continua*, vol. 72, no. 3, pp. 5251–5272, 2022.

[50] S. Tummalapalli, L. Kumar, and N. Lalita Bhanu Murthy, "Web Service Anti-patterns Prediction Using LSTM with Varying Embedding Sizes," in *Advanced Information Networking and Applications* (L. Barolli, F. Hussain, and T. Enokido, eds.), vol. 449, pp. 399–410, Cham: Springer International Publishing, 2022. Series Title: Lecture Notes in Networks and Systems.

[51] S. Liu, G. Lin, L. Qu, J. Zhang, O. De Vel, P. Montague, and Y. Xiang, "CD-VulD: Cross-Domain Vulnerability Discovery Based on Deep Domain Adaptation," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, pp. 438–451, Jan. 2022.

[52] H. Wei, G. Lin, L. Li, and H. Jia, "A Context-Aware Neural Embedding for Function-Level Vulnerability Detection," *Algorithms*, vol. 14, p. 335, Nov. 2021.

[53] S. Goswami, R. Singh, N. Saikia, K. K. Bora, and U. Sharma, "TokenCheck: Towards Deep Learning Based Security Vulnerability Detection In ERC-20 Tokens," in *2021 IEEE Region 10 Symposium (TENSYMP)*, (Jeju, Korea, Republic of), pp. 1–8, IEEE, Aug. 2021.

[54] H. Yu, X. Sun, Z. Zhou, and G. Fan, "A novel software defect prediction method based on hierarchical neural network," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, (Madrid, Spain), pp. 366–375, IEEE, July 2021.

[55] X. Huo, F. Thung, M. Li, D. Lo, and S.-T. Shi, "Deep Transfer Bug Localization," *IEEE Transactions on Software Engineering*, vol. 47, pp. 1368–1380, July 2021.

[56] S. Jeon and H. K. Kim, "AutoVAS: An automated vulnerability analysis system with a deep learning approach," *Computers & Security*, vol. 106, p. 102308, July 2021.

[57] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi, "Asteria: Deep Learning-based AST-Encoding for Cross-platform Binary Code Similarity Detection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, (Taipei, Taiwan), pp. 224–236, IEEE, June 2021.

[58] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network," *ACM Transactions on Software Engineering and Methodology*, vol. 30, pp. 1–33, July 2021.

[59] J. Yu, C. Mao, and X. Ye, "A Novel Tree-based Neural Network for Android Code Smells Detection," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, (Hainan, China), pp. 738–748, IEEE, Dec. 2021.

[60] S. A. Brown, B. A. Weyori, A. F. Adekoya, P. K. Kudjo, S. Mensah, and S. Abedu, "DeepLaBB: A Deep Learning Framework for Blocking Bugs," in *2021 International Conference on Cyber Security and Internet of Things (ICSIoT)*, (France), pp. 22–25, IEEE, Dec. 2021.

[61] Z. Haojie, L. Yujun, L. Yiwei, and Z. Nanxin, "Vulmg: A Static Detection Solution For Source Code Vulnerabilities Based On Code Property Graph and Graph Attention Network," in *2021 18th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, (Chengdu, China), pp. 250–255, IEEE, Dec. 2021.

[62] A. B. Farid, E. M. Fathy, A. Sharaf Eldin, and L. A. Abd-Elmegid, "Software defect prediction using hybrid model (CBIL) of convolutional neural network (CNN) and bidirectional long short-term memory (Bi-LSTM)," *PeerJ Computer Science*, vol. 7, p. e739, Nov. 2021.

[63] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, "Deep Just-In-Time Inconsistency Detection Between Comments and Source Code," 2020. Publisher: arXiv Version Number: 2.

[64] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2021.

[65] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep Semantic Feature Learning for Software Defect Prediction," *IEEE Transactions on Software Engineering*, vol. 46, pp. 1267–1293, Dec. 2020.

[66] J. Deng, L. Lu, and S. Qiu, "Software defect prediction via LSTM," *IET Software*, vol. 14, pp. 443–450, Aug. 2020.

[67] K. Shi, Y. Lu, J. Chang, and Z. Wei, "PathPair2Vec: An AST path pair-based code representation method for defect prediction," *Journal of Computer Languages*, vol. 59, p. 100979, Aug. 2020.

[68] R. Singh, J. Singh, M. S. Gill, R. Malhotra, and Garima, "Transfer Learning Code Vectorizer based Machine Learning Models for Software Defect Prediction," in *2020 International Conference on Computational Performance Evaluation (ComPE)*, (Shillong, India), pp. 497–502, IEEE, July 2020.

[69] Y. Hu, H. Jiang, and Z. Hu, "Measuring code maintainability with deep neural networks," *Frontiers of Computer Science*, vol. 17, 01 2023.

[70] C. Chen, R. Alfayez, K. Srisopha, B. Boehm, and L. Shi, "Why is it important to measure maintainability and what are the best ways to do it?," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 377–378, 2017.

[71] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, (London England United Kingdom), pp. 1–10, ACM, May 2014.

[72] S. Sengamedu and H. Zhao, "Neural language models for code quality identification," in *Proceedings of the 6th International Workshop on Machine Learning Techniques for Software Quality Evaluation*, pp. 5–10, ACM, Nov. 2022.

[73] M. Schnappinger, S. Zachau, A. Fietzke, and A. Pretschner, "A preliminary study on using text- and image-based machine learning to predict software maintainability," in *Software Quality: The Next Big Thing in Software Engineering and Quality* (D. Mendez, M. Wimmer, D. Winkler, S. Biffl, and J. Bergsmann, eds.), (Cham), pp. 41–60, Springer International Publishing, 2022.

[74] M. Schnappinger, A. Fietzke, and A. Pretschner, "Defining a software maintainability dataset: Collecting, aggregating and analysing expert evaluations of software maintainability," in *ICSME 2020. IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2020.

[75] M. Schnappinger, A. Fietzke, and A. Pretschner, "A software maintainability dataset," Sep 2020.

[76] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[77] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open Foundation and Fine-Tuned Chat Models," 2023. Publisher: arXiv Version Number: 2.

[78] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and Efficient Foundation Language Models," 2023. Publisher: arXiv Version Number: 1.

[79] N. Muennighoff, T. Wang, L. Sutawika, A. Roberts, S. Biderman, T. L. Scao, M. S. Bari, S. Shen, Z.-X. Yong, H. Schoelkopf, *et al.*, "Crosslingual generalization through multitask finetuning," *arXiv preprint arXiv:2211.01786*, 2022.

[80] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint*, 2022.

[81] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code Llama: Open Foundation Models for Code," 2023. Publisher: arXiv Version Number: 2.

[82] J. Fürnkranz, P. Chan, S. Craw, C. Sammut, W. Uther, A. Ratnaparkhi, X. Jin, J. Han, Y. Yang, K. Morik, M. Dorigo, M. Birattari, T. Stützle, P. Brazdil, R. Vilalta, C. Giraud-Carrier, C. Soares, J. Rissanen, R. Baxter, and L. De Raedt, *Mean Squared Error*. 01 2010.

[83] T. D. Gauthier, "Detecting trends using spearman's rank correlation coefficient," *Environmental Forensics*, vol. 2, no. 4, pp. 359–362, 2001.

[84] C. Andrade, "The P Value and Statistical Significance: Misunderstandings, Explanations, Challenges, and Alternatives," *Indian Journal of Psychological Medicine*, vol. 41, pp. 210–215, May 2019.