2023-01-23

Explainable AI for Software Engineering: A Systematic Review and an Empirical Study

Haji Mohammadkhani, Ahmad

UNIVERSITY OF CALGARY

Explainable AI for Software Engineering: A Systematic Review and an Empirical Study

by

Ahmad Haji Mohammadkhani

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING

CALGARY, ALBERTA
JANUARY, 2023

Abstract

In recent years, leveraging machine learning (ML) techniques has become one of the main solutions to tackle many software engineering (SE) tasks, in research studies (ML4SE). This has been achieved by utilizing state-of-the-art models that tend to be more complex and black-box, such as deep learning, which is led to less explainable solutions. This lack of explainability reduces trust and uptake of ML4SE solutions by professionals in the industry. One potential remedy is to offer explainable AI (XAI) methods to provide the missing explainability.

In this thesis, we aim to explore to what extent XAI has been studied in the SE domain (XAI4SE) and provide a comprehensive view of the current state-of-the-art as well as challenges and a roadmap for future work. In order to do so, we conduct a systematic literature review on 24 (out of 869 papers that were selected by keyword search) most relevant published studies in XAI4SE.

Our analysis reveals that among the identified studies, software maintenance (%68) and particularly defect prediction has the highest share on the SE stages and tasks that leverage XAI approaches. We also found that the published XAI methods are mainly applied to more classic ML models (e.g., random forest, decision trees, and regression models), rather than more complex deep learning and generative models (e.g. Transformer code models). Also, our study shows that XAI4SE is mainly used to improve the accuracy or interpretability aspects of the underlying ML models. Furthermore, we noticed a clear lack of standard evaluation metrics for XAI methods in the literature which has caused confusion among researchers and a lack of benchmarks for comparisons.

To fill one of the mentioned gaps, we conduct an empirical study on the state-of-the-art Transformer-based models (CodeBERT and GraphCodeBERT) on a set of software engineering downstream tasks: code document generation (CDG), code refinement (CR), and code translation (CT).

Initially, we evaluate the validity of the attention mechanism as an explainability method for each particular task. Next, through quantitative and qualitative studies, we show that CodeBERT and GraphCodeBERT learn to put attention to certain token types, depending on the downstream task. Furthermore, we show there are common patterns that cause the model to not work as expected (perform poorly while the problem at hand is easy), such as when there is a long input or when the model fails to pay proper attention to certain token types that are important for that task. Additionally, we suggest recommendations that may alleviate the observed challenges.

Preface

This thesis is an original work by the author, and it follows the "manuscript-based" style defined in the guidelines. During this research, I have made the following contributions:

- Mohammadkhani, A. H., Bommi, N. S., Daboussi M., Sabnis O., Tantithamthavorn, C., Hemmati, H. (2022). "A Systematic Literature Review of Explainable AI for Software Engineering" submitted to Information and Software Technology. Chapter 3 of this thesis is adapted from this paper.

In this paper, I was the main author, and I was responsible for the design, implementation, and writing of the paper. The co-authors helped in the ideation, searching, data extraction, and provided feedback on the design, and edited writings.

- Mohammadkhani, A. H., Tantithamthavorn, C., Hemmati, H. (2022). "Explainable AI for Pre-Trained Code Models: What Do They Learn? When They Do Not Work?" submitted to Empirical Software Engineering. Chapter 4 of this thesis is adapted from this paper.

In this paper, I was the main author, and I was responsible for the experiment design, implementation, and writing of them. The co-authors helped in the ideation, provided feedback on the design and discussions, and edited writings.

Table of Contents

List of Figures

List of Tables

List of Symbols, Abbreviations, and Nomenclature

Abbreviation	Definition
ML	Machine Learning
DL	Deep Learning
DNN	Deep Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
XAI	eXplainable Artificial Intelligence
XAI4SE	XAI for Software Engineering
SLR	Systematic Literature Review
LLM	Large Language Models
MLM	Masked Language Modeling
NSP	Next Sentence Prediction
MSP	Masked Span Prediction
MIP	Masked Identifier Prediction
kNN	k-Nearest Neighbor
SDLC	Software Development Life Cycle
SVM	Support Vector Machine
CDG	Code Document Generation
CR	Code Refinement
CT	Code Translation
PL	Programming Language
NL	Natural Language

RTD	Replaced Token Detection
LD	Levenshtein Distance

Chapter 1

Introduction

1.1 Motivation

In the past few decades, advancement in Machine Learning (ML) has exponentially increased with the combination of powerful machines, robust algorithms, and easier access to vast amounts of data. As a result, at present, ML models have been developed in many critical domains such as in healthcare, banking, finance, and terrorism detection [1, 2, 3].

In the field of software engineering as well, ML has been dominating many studies and activities. AI-based solutions have been applied on many automation tasks such as source code [4], test case [5], patch [6], and specification generation [7], or prediction tasks such as defect or vulnerability prediction [8], and recommendation tasks such as API recommendation [9].

Easy access to an abundance of software repositories (e.g., GitHub, StackOverflow, execution logs, etc.) has made SE an ideal field for data-driven ML algorithms to thrive. During the past decade, there has been many publications in sub-domains of SE research such as Mining Software Repositories (Software Analytics) and Automated Software Engineering that study applying ML techniques on various SE tasks such as API recommendation [9], risk prediction [10], code comprehension [11], code generation [4], code retrieval [12], code summarization [13], bug-fixing processes [14], software testing [5], fault localization [15], effort estimation [16], documentation [17], clone detection [18], program synthesis [19], etc.

Often case, the models with higher complexity such as SVMs [20], and deep neural networks [21] have achieved higher predictive accuracy in these tasks and thus recommended to be used by practitioners. However, similar to other fields, like NLP or Image Processing, as the ML models become more complex and achieve better accuracies, understanding their decision-making process becomes much harder. State-of-the-

art models evolved from using algorithms like decision trees or logistic regression that are intrinsically explainable, to using Deep Learning (DL) architectures, which basically offer no explanation for their decisions. Even though the final results may improve, more questions arise regarding their black-box nature.

This black-box nature of the solution can cause issues with trusting these models at different levels. It will provoke distrust in managers and customers, especially in more critical tasks. For instance, in the case of SE tasks whenever an automated solution is suggested to be deployed, e.g., a generated code or patch, etc.) where the wrong predictions/generations may cause expensive failure or extra manual overhead to check and fix, leading to a lack of adoption in practice [22]. Also, ethical challenges and the risk of immoral biases emerge [23] and it leaves the researchers with less clue about the right direction for improving their models [24].

In other words, acquiring the explainable output from the so-called black-box models, is one of the key problems of the existing machine learning models, to be adopted in practice. Though the black-box models themselves learn the connection between the features and the final output, they do not clearly describe the actual relationship between the given set of features and the individual prediction bestowed by the models. It is extremely important to know why a model makes a specific prediction to retain the model's reliability, in many domain including most SE tasks. For that, the model should be able to offer explanations for an individual prediction or the model's overall approach to the practitioners.

As an example, let's look at the defect prediction problem, which is one of the most studied tasks in SE by researchers when it comes to applying ML in SE. It has taken advantage of ML\DL models for many years. The goal is to predict if a piece of code or file is defective or not. In this scenario, if the model finds a file defective, but not any more information about its decision, the developer will have no clue on what to look for. If the model provides the reasons for the file's being defective or specify some lines or tokens as the possible defective parts, it would be easy for the software engineers to minimize the bugs based on the grounds provided and validate the ML-based recommendation.

In short, model explainability facilitates the debugging process, bias detection (recourse to individuals who are adversely affected by the model predictions), assess when to trust model predictions in decision-making, and determining if they are suitable for deployment.

Thus, there is a compelling need to overcome the trade-off between accuracy and interpretability to provide trustworthy, fair, robust models for real-world applications. As a consequence, AI researchers have focused on the space of eXplainable Artificial Intelligence (XAI) [25], which provides an explainable and understandable interpretation of the behavior of AI systems.

Responding to the growing attraction in the industry and academia to use AI models in SE in recent years and the critical need for interpretation of AI models that are getting more complex, we found that

the space of XAI for Software Engineering (XAI4SE) has been less explored. Therefore, we see a need to systematically review the existing work to (a) understand the current state and (b) identify the roadmap ahead for future research in XAI4SE. Therefore, this thesis starts by a systematic literature review (SLR). The SLR is based on relevant published journal and conference papers that offer explainability methods for SE tasks or propose ML models that include any type of explanation.

As one of the outcomes of the SLR, we notice that Code Models, i.e., pre-trained Large Language Models (LLMs), applied in SE tasks, such as defect prediction, code generation, program repair, comment generation, etc. are among models-tasks that show outstanding results and accuracies in SE, in recent years, but are unexplored in terms of explainability.

Inspired by the success of language models such as BERT [26] in NLP tasks, pre-trained code models are proposed to analyze the big corpora of source code and natural languages collected from open-source platforms (e.g., GitHub and StackOverflow). Such pre-trained code models have been used to automate various software engineering tasks, e.g., code understanding, code generation, code clone detection [27], defect detection [28, 29], and code summarization [30]. Automating such software engineering tasks has been shown to greatly improve software developers' productivity and reduce the costs of software development.

Recent studies proposed Transformer-based pre-trained code models, e.g., CodeBERT [17], GraphCodeBERT [31], CodeGPT [32], CodeT5 [33]. However, most of these studies often focus on improving its accuracy—***without considering its explainability aspect***. Thus, when deploying such models in practice, as explained, practitioners still do not know why such models provide a given recommendation or suggestion.

Therefore, in this thesis, we also conduct an empirical study to analyze the pre-trained code models through the lens of Explainable AI, so that we can understand what these models learn, when do they work, and when they do not work.

1.2 Objectives and RQs

Given the motivation discussed above, in this thesis, we have two explicit objectives:

- To conduct a SLR to provide practitioners and researchers a better insight into the efforts that have been made so far in XAI4SE and to provide a roadmap for future research in this direction.
- To look into one of the less studies but important areas of ML4SE (code models explainability) and using an internal feature of their Transformer architecture (i.e. attention mechanism) as an XAI method, explain different decision-making patterns of two well-known code models in three generation-based

tasks.

With the SLR, we want to answer three research questions, as follows:

- **RQ1: What are the main software engineering areas and tasks for which Explainable AI approaches have been used to date?**

  In this RQ, we explore different SE areas and tasks that have leveraged any XAI approach in their studies. As our analysis show, "software maintenance" is the highest explored area and defect prediction is the most favorable task for the XAI researchers in SE by far.

- **RQ2: What are the Explainable AI approaches adopted for SE tasks?**

  Here we analyze different XAI methods that have been utilized in SE and as it shows, most of the explanations are coming from self-explaining models and in the form of model internals. Also, we can see that LIME and ANOVA are two of the mostly utilized XAI methods.

- **RQ3: How useful XAI have been for SE?** In this RQ, we focus on the usefulness of XAI methods and how the offered explanation has helped the users to improve or better understand the models. One of the most interesting improvements is the usage of XAI to make the defect prediction models more precise while finding bugs and defects. Also, we discussed the lack of evaluation metrics according to the studies and how human-centered evaluations should be beneficial.

Furthermore, we have analyzed the limitations and challenges of the literature so far and provided some roadmap and possible directions for future works.

As a result, we noticed that DL and specifically Transformer based pre-trained code models are tremendously overlooked. To fill this gap, as discussed, we conduct an empirical study to explain these models.

Particularly, we focus on the two state-of-the-art pre-trained code models, i.e., CodeBERT and GraphCodeBERT with the three understanding & generation-specific downstream tasks, i.e., Code Summarization (Code→Text), Code Transformation (Code→Code), and Code Translation (Code→Code) tasks.

To explain the predictions of these models, we leverage an attention mechanism inside the Transformer architecture, which is an intrinsic Explainable AI approach. The attention mechanism allows us to understand what are the most important tokens in the input sequence that contribute the most to the tokens in the output sequences.

The empirical study addresses the following three research questions:

- **RQ1: Is the attention mechanism suitable to explain the pre-trained code models?**

In RQ1, we analyze the evidence of the attention scores' impact and influence on the Transformer-based models' decision-making and output and their suitability of the attention mechanism as an XAI method.

- **RQ2: What do the pre-trained code models learn?**

  In RQ2, we interpret the end-to-end attention scores extracted from CodeBERT and GraphCodeBERT to unveil patterns and insights into the models' decision-making process.

- **RQ3: When do the pre-trained code models not work?**

  Finally in RQ3, we focus on providing an explanation for scenarios that CodeBERT and GraphCodeBERT under-performed, and offer actionable suggestions to fix or overcome the struggles.

1.3 Contribution

To the extent of our knowledge, our systematic review is the first SLR for XAI in software engineering and our empirical study is the first study to discover hidden patterns in different code models while they're performing certain downstream tasks and how their way of learning and understanding the codes differ, according to the tasks they are fine-tuned to do. The key contributions of this study are:

- We present a consolidated body of knowledge of research on different aspects of XAI in software engineering
- By conducting a comprehensive search of the most influential venues in the field, we demonstrated the current state of XAI4SE. We identified the distribution of XAI methods used in the field, ML models that are explored, and representation techniques used by XAI methods in the selected studies.
- We identified gaps and limitations in the literature such as a lack of standard evaluation metrics and less explored SE tasks and ML models in terms of explainability.
- We empirically study the suitability of the attention mechanism as an XAI method to explain Transformer-based models.
- Analyzing the attention scores, we found interesting insights into models' decision-making for different tasks.
- We provide explanations on several scenarios, where CodeBERT and GraphCodeBERT under-perform.
- We offer some actionable recommendations on how to improve the models, in the future, to potentially alleviate the observed weaknesses.

1.4 Organization

The remainder of this thesis is organized as follows: Chapter 2 discusses the background and required information to understand the concepts and notations used in this thesis and also talks through the most important related studies. Afterward, chapter 3 covers the conducted SLR, its methodology, and results and analysis and chapter 4 covers the experiments conducted on pre-trained code models to provide an attention-based explanation and report the results. Chapter 5, conclude our discussions and explain potential future directions for this thesis.

Chapter 2

Background and related works

In this section, we present background knowledge and related work on XAI, ML and DL applications in SE, Transformer-based pre-trained code models, and XAI for SE.

2.1 Explainable AI

One of the common issues in the literature on Explainable AI is that there are several related concepts in this domain that are often used interchangeably in different articles. Therefore, in this section, we provide a brief description and the definitions of the terms that are widely used in Explainable AI jargon which help to understand the related literature and the concepts more precisely. This will also make the basis for the definition of XAI in SE, which will be used in this thesis.

2.1.1 Explainability and Interpretability

The most common term which hinders the establishment of concepts in XAI is the interchangeable use of interpretability and explainability in the literature. Although these terms are closely related there are some works that identify and distinguish these terms.

Though there are no clear mathematical definitions for interpretability and explainability, few works in this line attempt to clarify these terms. According to Miller [34], model interpretability means "the degree to which a human can understand the cause of a decision". Another popular definition came from Doshi-Velez and Kim [35], where they define it as "the ability to explain or to present in understandable terms to a human". Based on these explanations *Interpretability* is more like the cause and effect relationship of inputs and outputs of the model.

In contrast, *Explainability* is related to the internal logic and mechanics that are inside a machine learning algorithm. If the model is explainable, the human can understand the behavior of the model in the training and decision-making process. An interpretable model does not necessarily explain the internal logic of the ML model. This explains that interpretability does not exactly entail explainability, but an explainable model is always interpretable [36]. Therefore, it is necessary to address both interpretability and explainability aspects to better understand the ML model's logical behavior based on its inputs and outputs.

Regardless of this debate, since many studies have used these terms interchangeably and this SLR wants to cover all related studies, in the literature collection and analysis phases in this thesis, we have ignored the differences between these two terms.

2.1.2 Objectives of XAI

Basically, explainability can be used to evaluate, justify, improve, or manage AI, or even learn from it.

In this section, we explain the main objectives of explainability in AI, in general (not limited to SE), so that later we can assess which aspects of generic XAI have been used and identified as useful in the SE domain. It's worth mentioning that these objectives are not separate concepts and may overlap in many aspects or be called with other names in different articles, but here, we gathered the most mentioned objectives in the literature.

An XAI method can aim to explain different aspects of an ML model. For instance, the practitioners may want to focus on the input data in an ML model to help them properly balance the training dataset. While in another work, researchers may focus on the final output of the model to be able to provide a human-understandable explanation to the model's end user. In this section, we will go through some of these aspects of XAI and some suggested questions that can guide researchers' efforts, while focusing on these aspects.

Justification

One of the main objectives of explaining a model is to justify the model's decision in order to increase its credibility. This objective's aim is to gain the trust of users or people who are affected by the AI model. For this purpose, it's not necessary to explain different components or algorithms of a model, but it's required to find the connection between inputs and outputs [36].

To answer this need, it's important to know why an instance gives a specific output. It would also be helpful to elaborate on the feature(s) of the instance that determines the prediction or in some cases, to explore the reasons for getting the same output for two different inputs. Sometimes it's important to know

how much change and in what direction in an instance is required or tolerable, in order to see a different output. Being able to answer such questions will make it easier for the users to trust the models.

Improvement

Improvement is one of the most important and strongest inspirations behind many explainability efforts [37]. Working with black-box complex models, it is a common problem that the model starts to make wrong decisions but the reasoning behind it is unclear to developers or researchers. The problem can be due to imbalanced training data or an inadequate internal part or overfitting of the model on specific features. Providing an explanation can be helpful in these situations.

Understanding

Understanding a model is another motivation that is mentioned in the literature. It's a very general term but that usually pairs up with other objectives or makes them possible. Understanding a model may lead to its improvement, test its fairness, or increase the level of the user's confidence.

Fairness

Fairness is a widely used term in the space of AI and ML. Hence, understanding the term fairness is important before discussing how it relates to Explainability of ML models. Broadly speaking, fairness is the quality or state of being fair or having impartial judgment. Fairness has been discussed in many disciplines such as law, social science, quantitative fields, philosophy etc. [38] A few example definitions are discussed below:

- Law: fairness includes protecting individuals and groups from discrimination or mistreatment with a focus on prohibiting behaviors, biases and basing decisions on certain protected factors or social group categories.
- Social Science: "often considers fairness in light of social relationships, power dynamics, institutions and markets." [38] Members of certain groups (or identities) that tend to experience advantages.
- Quantitative fields (i.e. math, computer science, statistics, economics): questions of fairness are seen as mathematical problems. Fairness tends to match to some sort of criteria, such as equal or equitable allocation, representation, or error rates, for a particular task or problem.
- Philosophy: ideas of fairness "rest on a sense that what is fair is also what is morally right." [38] Political philosophy connects fairness to notions of justice and equity.

In machine learning, fairness can be pursued, in order to prevent biases and discriminations that may happen in different parts of an ML. It can appear in the training data where the dataset is imbalanced, in algorithms where for instance using specific optimization functions may lead to biased decision-making, or in the output representation where wrong conclusions are drawn from an observation [39].

XAI methods can significantly help researchers to consciously detect and eliminate biases and achieve fairness.

Transparency

Transparency is the opposite of black-boxness, that means when an ML model makes a decision, its whole process can be comprehended by a human [40]. Full transparency is usually impossible to fulfill, but it can be achieved in three levels that are: simulatable, decomposable, and algorithmic transparency [41]. Simulatable transparency means the model can be readily interpreted or simulated by a human, which means having a specific input, the user must be able to calculate the output having the model's parameters. Decomposable transparency is when smaller components of the model can be explained or explainable, separately. Finally, algorithmic transparent models are those whose training can get investigated mathematically.

2.1.3 Modality of Explanation

Model interpretability can be achieved if the model is understandable without further explanation [42]. According to the interpretability literature, the model understanding can be achieved by either considering inherently interpretable predictive models (i.e.; Linear regression, Random forest, Decision trees, etc.) or post-hoc interpretability approaches [43].

Intrinsically Interpretable

Intrinsically interpretable models are also called inherently interpretable or self-explaining models. These models usually provide some level of transparency in their design and structure or they might generate additional outputs such as attention maps, disentangles representations, or textual or multi-model explanations alongside their predictions. Decision trees are one of the most famous self-explaining ML models. However, these inherently interpretable models often suffer from the accuracy-interpretability trade-off.

Post-hoc interpretability

This approach approximates the black-box model using a transparent surrogate model without changing the base model. In this approach, the explained model is treated like a black-box and can be obtained even

without any access to its internal components which is ideal when the model is too complex. However, post-hoc methods can also be applied intrinsicly and be model-specific [44]. Some of the most common post-hoc methods are LIME [45] and SHAP [46] which provide an explanation regarding defined features.

2.1.4 Scope of Explanation: Local and Global Explainability

Portability is an important aspect of post-hoc interpretability. It explains how far the explainer method is dependent on access to the internals of the global training model or the explained model.

Explainability methods are called *model-specific* or decompositional or white-box models, if the interpretability algorithm is restricted to explain only one specific family of models. In contrast, the algorithms which explain the output of different global models by considering the global model as a black-box, are called *model-agnostic* [47].

According to the present deployments in the space of interpretability, post-hoc interpretability can be further classified as global interpretability and local interpretability based on the scale of interpretation.

Global Interpretability

In order to explain the model output globally in a comprehensive way, the knowledge about the trained algorithm, trained model, and input data would be sufficient. The basic idea behind the holistic global interpretability is understanding how the model makes decisions based on the entire feature set, parameters, and structures. The outcomes of holistic-level interpretability are to help identify: (a) the features' importance levels, and (b) the feature interactions with each other.

Local Interpretability

When discussing local interpretability for a single prediction, it considers a single instance and argues that how the model's prediction for the particular instance corresponds to that instance's features. The prediction will be on a linear or monotonic relationship of some features in the local context. For example, the condition of all patients might not linearly depend on the patients' blood pressure. But, if we look at one specific patient and examine that patient's condition, while watching the small changes of his blood pressure, there is a higher chance of finding a linear relationship in the sub-region of our data. This is a reason to expect more accurate explanations from local explanations compared to global explanations.

2.1.5 Related Surveys and Literature Reviews

There have been lots of surveys and literature reviews addressing the XAI topic in general or for specific fields. For instance, a study surveyed some traditional ML models such as SVM, linear, and tree-based models and noted a trade-off between their performance and reliability [48]. They also mentioned the ambiguity and confusion among works addressing XAI in terms of taxonomy and definitions.

In an attempt to standardize the efforts in XAI, some studies has tried to clarify some of the definitions and terminology and insisted in distinguishing the terms "interpretability" and "explainability" as different terms with the first only being one of the features of the latter [36]. They also claim that different approaches to solve the problem of black-box models, usually are unilateral and fail to address different aspects of explainability. A similar approach was taken by others that offered a more updated insight and also specified challenges and opportunities ahead[49].

Besides works that surveyed or reviewed the literature on XAI, in general, there were valuable attempts to particularly review the state of XAI in specific fields or models. As expected, medical applications that have taken great advantage of ML models, in recent years, are also among the most critical and sensitive domains. In a research, more than 200 papers that have used XAI in deep learning models for medical image processing are analyzed, which itself shows the high interest of researchers in the field [50]. They also provide some suggestions for future works including the need for medical experts to actively participate in designing interpretation methods.

Natural language processing was also one of the areas that benefitted from surveys and reviews to examine the state of XAI in the literature. For instance, A survey examined 50 papers that are related to XAI in NLP and categorized and analysed them according to different aspects like locality and globality or being self-explaining or post-hoc [43]. Furthermore, they note a debate among researchers about the very existence of a trade-off between explainability and performance. Similar to many other surveys in other fields, they also mention the lack of clear terminology and understanding of XAI methods and provide insightful guidelines.

There are also reviews or surveys that instead of tasks or fields of study focus on different types of data or models. Among these works, there are studies that focus on time-series data specifically [51], or analyze the advances in XAI for tabular data [52]. There are other researches that study different XAI methods for deep neural networks [53] or specifically reviews XAI in deep reinforcement learning models [54].

Software engineering is also a field that has benefited from ML models for a long time and SLRs have been conducted to analyze the literature for ML or DL techniques application in SE [55, 56]. However, to the best of our knowledge, this is the first work offering a systematic literature review for XAI in software engineering.

2.2 AI4SE

2.2.1 ML and DL Applications in SE

Machine Learning is a subfield of AI where a model is able to generalize examples and learn patterns when a fair amount of data is available. Many SE tasks can get automated and fit into ML problem types and also there is an abundance of open-source software projects containing source codes and meta-data, making this field an ideal opportunity for taking advantage of data-driven ML models.

For instance, we can address a defect prediction task as a classification problem where the model is supposed the take an input such as a class, function, source code file, or set of features, and classify whether it's defective or not. An ML model can be trained on previous versions of a project and learn the patterns of buggy commits and then be used for future commits to predict the possibility of a bug's presence.

Considering this potential, in the last decade, researchers applied and took advantage of different ML models to automate SE tasks such as code completion [57], defect prediction [58], code translation [59], etc. Initial efforts took advantage of simple models such as N-gram, regression-based, and kNN models and were able to outperform the previous traditional approaches [60].

However, a massive advancement in AI applications in SE was due to the successful utilization of DNN models that are able to learn more complex, non-linear relationships in the data. DL models are more flexible than traditional ML models, as they can be applied to a wider range of tasks. This flexibility led to many SE tasks (40 according to a recent review [55]) in different software development stages to utilize different DL architectures such as FNN, RNN, and CNN.

In 2017 researchers at Google introduced Transformer [61] as a new DL model architecture that proved to be quite successful. Transformer architecture is an attention-based neural network architecture, designed to process input sequences efficiently, using self-attention mechanisms to calculate the relationships between different elements in the input. This mechanism allows the model to capture long-range dependencies and contextual information in the input data. Therefore, Transformer models are applied to a variety of SE tasks and outperformed other DL models in multiple tasks [17].

2.2.2 Transformer-based Pre-Trained Code Models

Pre-trained code models are deep learning models (such as transformers) which are trained on a large dataset (e.g., GitHub and StackOverflow) to perform specific source code understanding and generation tasks. Pre-trained code models (aka. language models of code) are normally trained in a self-supervision fashion using various model architectures (e.g., BERT) with various learning techniques. For example, Masked Language

Modeling (MLM) and Next Sentence Prediction (NSP). When the pre-trained code models are trained on a large corpus, it allows them to learn the universal representations of source code and natural languages specific to programming. The development of such pre-trained code models is very beneficial for various downstream tasks, thus avoiding training a new code model from scratch, and making the code models more reusable.

Recently, researchers developed pre-trained code models using various types of Transformer architecture (e.g., CodeBERT, GraphCodeBERT, CodeGPT, and CodeT5).

CodeBERT [17] is a pre-trained code model that is trained on a BERT architecture (i.e, Bidirectional Encoder Representations from Transformers) [26]. It is trained on MLM and NSP tasks using NL-PL pairs in 6 programming languages (Python, Java, JavaScript, PHP, Ruby, Go).

GraphCodeBERT is similar to CodeBERT, but it considers the structure of the code by utilizing its data flow as well for pre-training. It uses MLM, Edge Prediction and Node Alignment as pre-training tasks. CodeGPT is another Transformer-based model but it's pre-trained only on a programming language with tasks similar to GPT-2 [62] CodeT5 is also a pre-trained encoder-decoder model which utilizes token-type information written in code. It is pre-trained on Masked Span Prediction (MSP), identifier tagging, Masked Identifier Prediction (MIP), and bimodal dual generation which are designed to help the model understand token type information.

The existing pre-trained code models have been used for various downstream software engineering tasks. which can be categorized into four types: (1) Text→Text (e.g., language translation of code documentation [32], query reformulation [63]); (2) Text→Code (e.g., code search [64, 65]); (3) Code→Text (e.g., code summarization [66], commit message generation [67, 68]); and (4) Code→Code (e.g., automated program repair [69, 70, 71], programming language translation [72], code completion [4]).

2.3 Explainable AI for Software Engineering (XAI4SE)

As we discussed, explainability is now becoming a critical concern in software engineering. Many researchers often employed AI/ML techniques for defect prediction, malware detection, and effort estimation. While these AI/ML techniques can greatly improve developers' productivity, software quality, and end-user experience, practitioners still do not understand why such AI/ML models made those predictions [73, 74, 75, 76, 77, 78].

In software engineering, explainable AI has been recently studied in the domain of defect prediction (i.e., a classification model to predict if a file/class/method will be defective in the future or not). In particular, the survey study by Jiarpakdee [75] found that explaining the predictions is as equally important and useful

as improving the accuracy of defect prediction. However, their literature review found that 91% (81/96) of the defect prediction studies only focus on improving the predictive accuracy, without considering explaining the predictions, while only 4% of these 96 studies focus on explaining the predictions.

Although XAI is still a very under-researched topic within the software engineering community, very few existing XAI studies have shown some successful usages e.g., in defect prediction. In one example, Wattanakriengkrai [79] and Pornprasit and Tantithamthavorn [80] employed model-agnostic techniques (e.g., LIME) for line-level defect prediction (e.g., predicting which lines will be defective in the future), helping developers to localize defective lines in a cost-effective manner. In another example, Jiarpakdee [76] and Khanan [81] employed model-agnostic techniques (e.g., LIME) for explaining defect prediction models, helping developers better understand why a file is predicted as defective. Rajapaksha [77] and Pornprasit [80] proposed local rule-based model-agnostic techniques to generate actionable guidance to help managers chart the most effective quality improvement plans.

While there exist research efforts on the explainability of classification tasks in SE domains (e.g., defect prediction), little research is focused on Transformer-based pre-trained code models. Particularly, practitioners often raised concerns e.g., *why this source code is generated? why this code token is modified?*. A lack of explainability of code models could lead to a lack of trust, hindering the adoption in practice.

To address this challenge, prior studies in the software engineering domain have employed various Explainable AI approaches on transformer-based code models [82, 83, 84, 85]. For example, Cito [86] proposed an approach to generate a global explanation to understand the weaknesses of the code models. Cito [87] proposed an approach to generate counterfactual explanations to explain the model's behavior by letting the end user know if the source code had been changed in a specific way, how the model's prediction would be. This will help the users to have a more specific answer, when asking the model to do a task like security vulnerability detection.

Also, another research using probing tasks has surprisingly claimed that CodeBERT and GraphCodeBERT which are trained on codes have a very slim difference in code understanding, compared to an NL model such as BERT [88]. Another work has replicated an NLP study [89] on BERT, but using codes as their training benchmark. They focused on the self-attention mechanism of BERT and compared the attention behavior of the model in NLP and code. However, no efforts have been made to study the different ways that each code model learns to perceive the source codes when doing different downstream tasks. Do they learn to focus on different parts of the code based on their given and fine-tuned tasks? Are there meaningful differences between similar code models understanding of the code, doing the exact same task?

Chapter 3

A Systematic Literature Review of Explainable AI for Software Engineering

In this section, we are presenting a systematic literature review of XAI for SE. Initially, we will explain our methodology for the search, paper selection, and data extraction process. Then we will present our results and findings, and finally analyze them by answering three main RQs.

3.1 Methodology

Our methodology for this review was mostly inspired by the guidelines proposed by Kitchenham for conducting an SLR [90]. The proposed method includes three main stages: planning, conducting the review, and reporting the results.

The planning phase comprises two steps. Identifying the need for a review which is discussed in Section 1 and developing a review protocol in order to remove the researchers' bias in the reviewing process. Our review protocol's main components include research questions, search strategy, paper selection, inclusion and exclusion criteria, and data extraction. This review protocol is recurringly modified and evolved during the process. Finally, the third step is basically reporting the results and the analysis.

The first two steps are discussed in the following section while the results are reported in section 3.3.

3.1.1 Research Questions

Regarding the purpose of this SLR which is investigating the utilization of XAI methods for ML\DL models in software engineering, we formulated the following RQs:

1. **RQ1: What are the main software engineering areas and tasks for which Explainable AI approaches have been used to date?**
   (a) What are the main software engineering areas and tasks for which AI has been applied?
   (b) What are the ML4SE works that offer explainability?
   (c) What are the goals and objectives of each research paper reviewed?
2. **RQ2: What are the Explainable AI approaches adopted for SE tasks?**
   (a) What kind of XAI techniques have been used?
   (b) What kind of explanations do they offer?
3. **RQ3: How useful XAI have been for SE?**
   (a) What are the objectives of XAI applying on SE and how they have been useful?
   (b) What are the means of evaluating XAI in SE?

Each of these RQs are aimed to help us shed some light on different areas of our review. RQ1 is focusing on SE tasks that took advantage of any kind of XAI method to explain their models. While RQ2 is concentrated on the XAI methods and how they have been utilized. Finaly, RQ3 cares about how successful XAI has been in doing its specified task. Each of these RQs also has sub-questions that together, form the answer to their respective main question.

3.1.2 Search Strategy

The search strategy that is used in this work, starts by choosing the digital libraries to be searched. To perform the primary search phase, five main scientific publication databases in software engineering and AI have been selected:

- ACM Digital Library
- IEEE
- Springer

Table 3.1: Key search terms and their respective alternative search terms.

Key Search Terms	Alternative Search Terms
Explainable AI	Interpretable, Interpretability, Explainability, local model, global model, model-agnostic, explainer model, explanations, interpretability algorithm, black-box models, rule-based, counterfactual, causal
Software Engineering	SQA, Software Analytics, defect prediction, security, vulnerability detection, intrusion detection system, API recommendation, risk prediction, code comprehension, code generation, code retrieval, code summarization, bug-fixing processes, software testing, effort estimation, documentation generation, clone detection, program synthesis, image to code, security, program repair
Artificial Intelligence	Machine learning, classification, neural networks, deep learning, image processing, text mining, text analysis, classification, clustering, rule mining, association rules, NLP, Natural Language Processing, embedding, transformer, adversarial learning

- Wiley Online Library
- Elsevier

We considered three main categories of "Explainable AI", "Software Engineering", and "Artificial Intelligence" to create our search strings. By integrating alternative terms and synonyms for each of them, we gathered a list of keywords in three categories as shown in Table 3.1.

Using the boolean operators AND and OR, we formulated a search string that is:

("Interpretable" OR "Interpretability" OR "Explainability" OR "local model" OR "global model" OR "model-agnostic" OR "explainer model" OR "explanations" OR "black-box models" OR "rule-based" OR "counterfactual" OR "causal") AND ("SQA" OR "Software Analytics" OR "defect prediction" OR "API recommendation" OR "risk prediction" OR "code comprehension" OR "code generation" OR "code retrieval" OR "code summarization" OR "bug-fixing processes" OR "software testing" OR "effort estimation" OR "documentation generation" OR "clone detection" OR "program synthesis" OR "image to code" OR "security" OR "program repair" OR "vulnerability detection" OR "intrusion detection system" OR "Malware detection") AND ("Machine learning" OR "classification" OR "neural networks" OR "neural networks" OR "image processing" OR "text mining" OR "text analysis" OR "classification" OR "clustering" OR "rule mining" OR "association rules" OR "NLP" OR "Natural Language Processing" OR "embedding" OR "transformer" OR "adversarial learning")

The OR operators will check if any of the search terms in one category exist in the paper and the AND operator will make sure that all three categories are found in the papers.

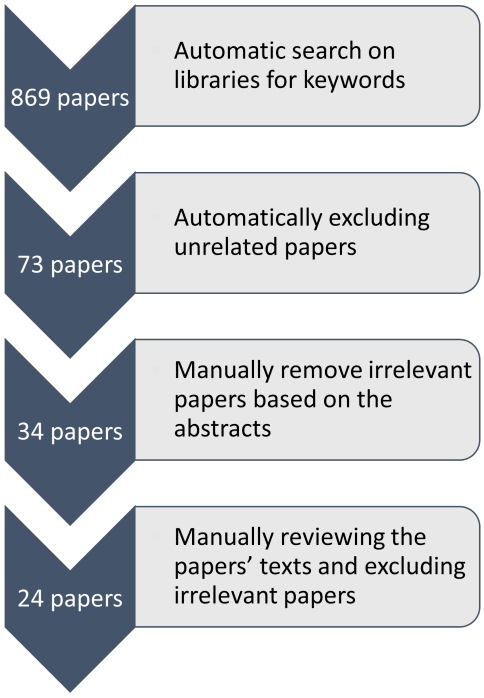


Figure 3.1: An overview of the searching strategy and its different stages with the number of studies in each step.

3.1.3 Study Selection/ Inclusion and Exclusion Criteria

First, we started by searching a smaller version of the alternative search terms (more generic terms) on the full documents of papers in all of the specified databases with no time limit for the papers on the date "06/21/2022". This ended up with hundreds of irrelevant papers as the result. So it was decided to limit the search to the title, abstract, and keywords of the papers. Given that the terms used in title, abstract, and keywords are more specific and not very generic, we augmented the search terms in the new search to the current list to make sure we do not miss any relevant paper. At this point, we could find a total of 869 papers which many of them were unrelated to SE and belonged to Health, Physiology, Medicine, or Security. We used the search engine tools to filter the papers of unrelated fields that left us with 73 papers.

Lots of irrelevant papers were still among the search results so in order to find and select the relevant papers, we manually reviewed the abstracts of all the 73 papers (this job was divided between four authors. Then one author validated all decisions) and included or excluded papers based on the following criteria:

- Inclusion
  - Full text papers published as journal or conference papers that comply with Interpretability and explainability definition provided in section 2.1.
  - Papers that are written in English Language.

- Papers that are related to software engineering.
- Papers that are available in full text.

- Exclusion
  - Books, gray literature (theses, unpublished and incomplete work), posters, secondary or review studies (SLR or SMS), surveys, discussions and keynotes are excluded.
  - Short papers where page count is less than 4 pages.
  - Papers about Software Engineering but not discussing about interpretability and explainability.
  - Papers about interpretability and explainability but not applying it on Software Engineering tasks.

So if a paper meets all of the defined inclusion criteria and also does not meet any of the exclusion criteria, we included it in the final papers list. Finally according to all of these filters and criteria, we were able to extract 25 papers that totally matched our interests. One paper [1] was discarded due to the problem of accessing its full version in the digital libraries and 24 papers remained, which can be seen in Table 3.6.

3.1.4 Data Extraction

For the data extraction phase, we defined a checklist of items that could help us extract the required information from each paper to answer the RQs. The checklist includes both quantitative and qualitative properties. The 24 papers were divided between four authors and one author verified and consolidated the results. We defined these main properties that could help us answer the RQs as below:

1. Publication details (Title, Authors, Published Date, Venue, Authors' affiliation)
2. What is the aim/ motivation/ goal of the research? (RQ1, RQ3)
3. What are the key research problems addressed? (RQ1, RQ3)
4. What phases of the SE are considered in the study? (RQ1)
5. What SE tasks are under experiment in the study? (RQ1)
6. Is the study conducted based on open-source data or data from industry? (RQ1)
7. What are the ML\DL models and techniques considered in the study and how they have been evaluated? (RQ1)

[1] https://ieeexplore.ieee.org/document/1374186

8. What are the XAI methods and techniques used in the study and how they are evaluated or represented (or visualized)? (RQ2)
9. What is the scope and modality of the explanation offered in the study? (RQ2)
10. What is the replication status of the experiments in the study? (RQ1)
11. The number of participants used in the study where human evaluation was involved? (RQ
12. What are the strengths and limitations of the study? (RQ2, RQ3)
13. What are the key research gaps/ future work identified by each study?
14. Is the explainability among the main focuses of the study or just a side-benefit? (RQ3)

3.1.5 Data Synthesis

Our data synthesis focused on summarizing and presenting the extracted data to convey meaningful information regarding the papers to address the RQs. In Section 3.2, we present all of our findings and results using tables, pie charts, bar charts, etc. in order to share insightful information from our analysis and studies.

3.2 Results

This section summarizes the results of the final selected papers after the search process and the meta-analysis on their publication places and years.

3.2.1 Selected Primary Studies

After filtering out or adding different research to our list of papers in multiple steps, finally we came up with a list of 24 papers that totally matched our search criteria. In Table3.6, the paper's title, name of authors, year and source of publication, and publication type (journal or conference) of all these final selected studies are summarized.

3.2.2 Publication Statistics

As shown in Figure 3.2, the first research in our selected studies goes back to 2008, published in Software Quality Journal. This study uses k-nearest neighbor (kNN) and classification and regression trees generated based on the CART algorithm to perform the software effort prediction. Even though this study does not directly mention the XAI concept, it touches upon the explainability aspect in an implicit manner by being

Table 3.2: Authors with more than one contribution in the selected studies.

Author	# of papers
Tantithamthavorn, Chakkrit	5
Jiarpakdee, Jirayus	3
Dam, Hoa Khanh	2
Pornprasit, Chanathip	2
Thongtanunam, Patanamon	2
Matsumoto, Kenichi	2

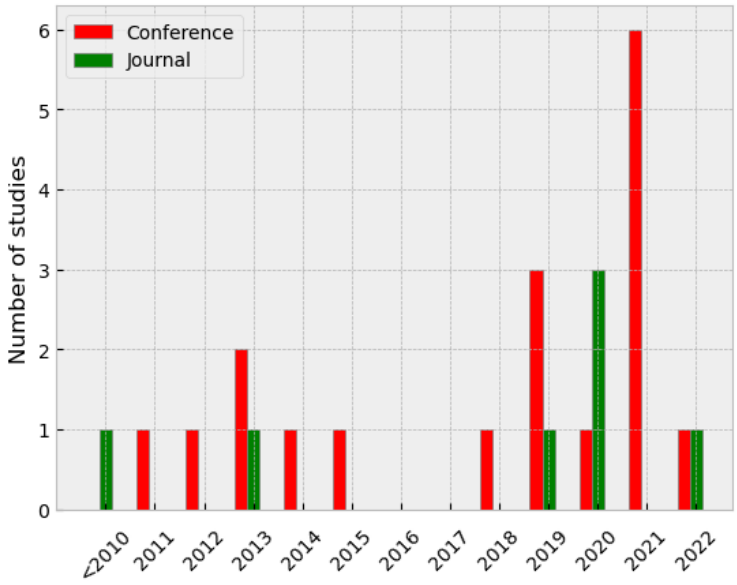


Figure 3.2: Distribution of selected studies over years and types of venues.

able to find the dominant predictor variables and how they're statistically significant using Anova method. After that, there is a small number of XAI works in SE each year but then from 2019, we can see a trending pattern and popularity of related research.

In the selected studies, a total of 75 authors contributed, while six mentioned in Table 3.2 are the authors in more than one paper. Also in terms of conferences and journals, as shown in Table 3.3, only three journals and one conference had more than one paper in the selected studies, which confirms that this field is still a new domain in the SE community.

Another interesting observation is that before 2018, most of the publications were in AI conferences or journals that weren't specific to SE, while after that, all of the publications belong to SE-specific venues which shows the rising interest of the SE community in the field.

Table 3.3: Conferences [C] and journals [J] and number of papers published in each venue. The full name of each venue can be found in Table 3.6

Conference/Journal	# of papers
RAISE [C]	2
MSR [C]	2
Transactions on Software Engineering [J]	2
Software Quality Journal [J]	2
Empirical Software Engineering [J]	2
Symposium on Cloud Computing [C]	1
TOSEM [J]	1
IUI [C]	1
ICSE-NIER [C]	1
Symposium on Applied Computing [C]	1
ICSE-Companion [C]	1
ESEC/FSE [C]	1
RE [C]	1
ICMLA [C]	1
ASE [C]	1
ICECS [C]	1
ISSRE [C]	1
DSA [C]	1
ICTAI [C]	1

3.3 Synthesis and Analysis

3.3.1 RQ1 Results: What are the main software engineering areas and tasks for which Explainable AI approaches have been used to date?

In our selected studies, we surveyed different SE activities and SE tasks that XAI methods and studies have been focused on or utilized. We organized the tasks and activities based on the Software Development Life Cycle (SDLC) stages [55] and in this section, we are going to present and analyze the results to answer RQ1 of this study.

RQ1.1 Results: What are the main software engineering areas and tasks for which AI has been applied?

In our studies as shown in Figure 3.3, the most attractive activity for the researchers was software maintenance with 68% of the tasks being used for interpretation. After that, software development has the largest share with 16%, and software management and software requirements each with two tasks in all of the studies have 8%. It's noteworthy that among the six stages of SDLC, software design and testing are two activities

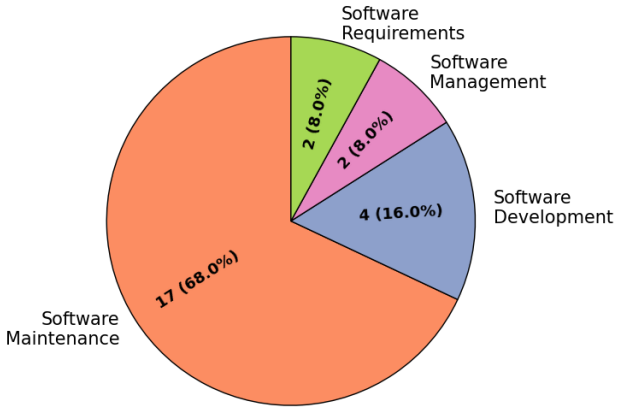


Figure 3.3: Distribution of studies per SDLC stage in the selected studies. Reported by #Papers (% proportions over all).

Table 3.4: The break-down of SDLC stages to tasks and their number of appearance in the selected studies.

Activity	Task	#
Requirement	Requirements engineering	1
	Requirements quality	1
Development	Code translation	1
	Code autocompletion	1
	Natural language to code	1
	Code Summarization	1
QA & Maintenance	Quality assessment	1
	Defect prediction	11
	Code reviews	1
	Reliability prediction	1
	Technical debt detection	1
	Valid bug report detection	1
	Variable misuse detection	1
Management	Effort Prediction	2

that have no presence in all of our selected studies.

In Table 3.4, we can see different tasks in each activity domain's broken-down. The most interesting fact that is seen is the high ratio of the defect prediction task compared to all other tasks in the studies. This task alone has the lead with 44% of the total number with a meaningful gap from the next task, which is effort prediction with only two studies and the other tasks each one being utilized only once. Also in general, QA and software maintenance tasks have the highest diversity with having seven different tasks.

Defect prediction is basically a classification problem to tag a code snippet, class, etc. as faulty or not faulty. This makes it a very ideal task for machine learning algorithms and also for XAI. It's also a very important task that ML models have shown promising results in.

We also examined a survey [55] on deep learning models for software engineering tasks and interestingly, we found 27 different tasks that already have taken advantage of black-box DL models, and yet, no explanation method has been used on them. Tasks such as "Code generation", "Code comment generation", "Test case generation", "Bug classification", etc. This can show the big gap between ML applications in SE and the utilization of XAI in the field.

Furthermore, we observed that among the reviewed studies that include an empirical evaluation, 16 papers only used open-source data, three papers only used publicly unavailable industrial data [91, 92, 93], while two papers used both [94, 95]. The data type in all the studies are either textual which can be source code or NL, or tabular which is a set of extracted features.

RQ1.2 Results: What are the ML4SE works that offer explainability?

To answer this question, we analyzed different ML models and the number of times that they have been used in the selected studies. As the sorted results show in Figure 3.4, while many models are rare and have only one use case, models like random forest and decision tree have been very popular with nine and seven uses. This is probably due to the fact that these classic models are usually being used as the baselines in many papers and also the fact that these models offer a level of self-explainability so it is expected to see them more in our selected studies.

The next popular models are regression models that are used for both regression problems (such as effort prediction) and classification problems (such as defect prediction). Regression models are used six times in total, while four of them are logistic regression, and linear and robust regression each have only one use-case.

Neural networks are also favored by having a total of 15 times being experimented, with adequate diversity among themselves. More complex code models like Code2Vec, Code2Seq, TransCode, and CodeBERT can be found in the studies, as well as simpler networks like feed-forward neural networks, CNN, RNN, or basic transformer models are also among the explained models.

In Table 3.5 ML models are shown alongside the tasks they were used to solve. Models like the random forest or regression models are mostly utilized for defect prediction. The popularity of random forest for defect prediction is probably due to the fact that it's a self-explaining model which is a good fit for this specific task with high accuracy. Also, it is interesting to see the diversity of tasks that take advantage of the decision tree as a simple yet effective model.

DNN code models like Code2Vec, Code2Seq, Transcode, and CodeBert are always used for generation-based tasks such as code summarization or code translation which is understandable due to the complexity of these tasks and traditional models' incompatibility with them.

Regarding a large number of defect prediction studies, we also focused on those papers more specifically.

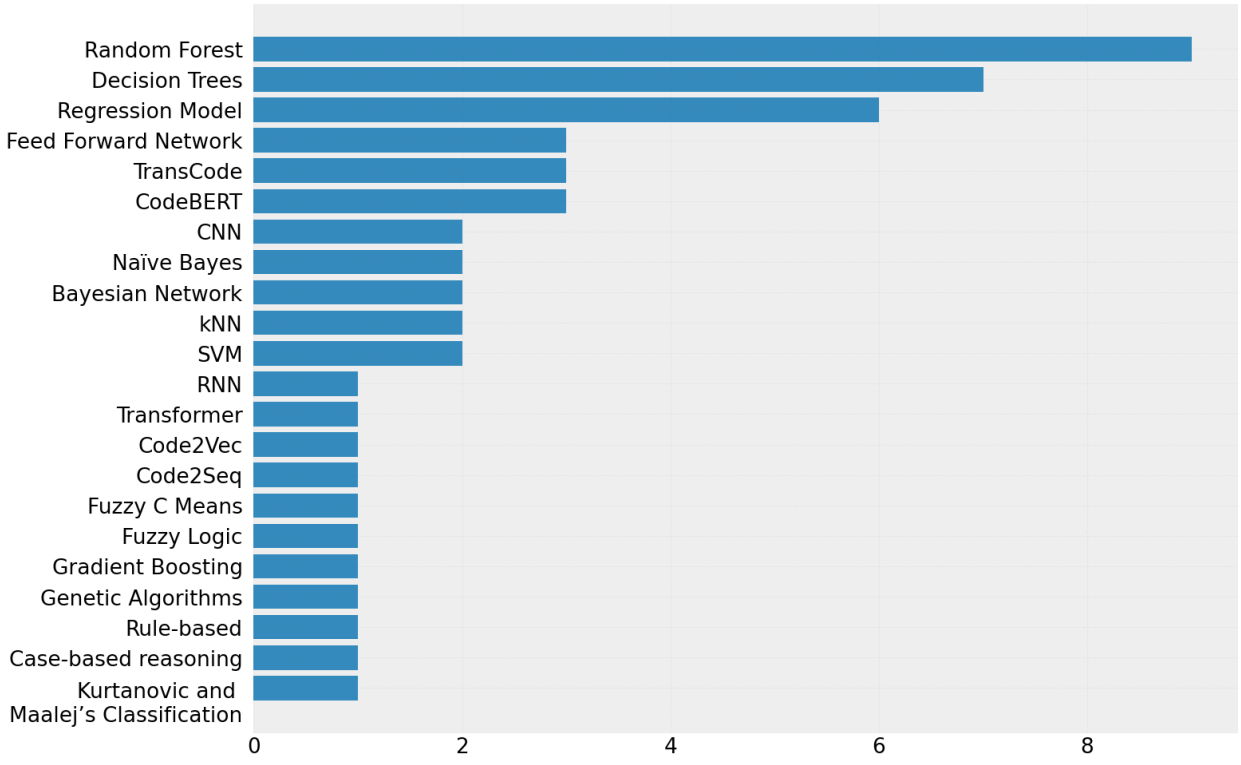


Figure 3.4: Number of experiments each ML model have been used in the selected studies

Among the 11 works focused on this task, there is quite a diversity of ML models being used and even some studies cover different models for the purpose of their research. As mentioned, random forest is the most popular method that has been used in eight different papers either as the main model or as a benchmark to compare with. Logistic regression and deep learning neural networks are the second most popular models being used in five and three papers. There are other models that only have been used in two or one study. Models such as Support Vector Machine (SVM), k-Nearest Neighbours, and C5.0 Boosting. In these defect prediction tasks, standard evaluation metrics for ML models such as precision, recall, F1 score, and AUC are commonly used.

We also analyzed the SE tasks that these ML models have been applied to in terms of their ML problem type. As shown in Figure 3.5, 18 tasks (more than 65% of the tasks experimented in the selected studies) fall into the classification problems. While only three tasks are considered regression problems (reliability prediction, requirements quality, and effort prediction), and four can be called generation problems (code translation, code autocompletion, natural language to code, and code summarization).

Classification tasks are among the simplest models to explain because of the fixed number of possible outputs that leads to a more straightforward evaluation, thus it makes sense for the researchers to focus on them. On the other hand, generation tasks are among the hardest since the output is usually harder to

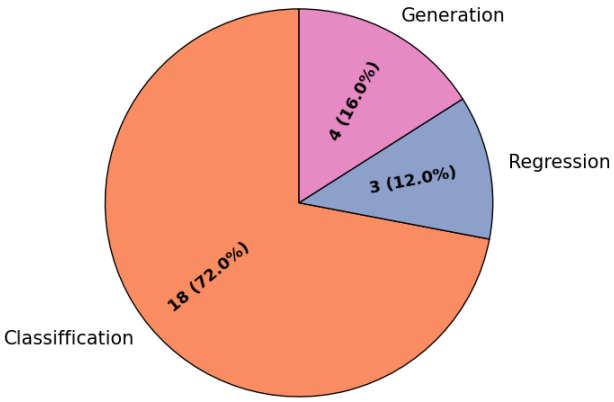


Figure 3.5: Distribution of SE task types in selected studies. Reported by #Papers (% proportions over all).

evaluate objectively. This gets more interesting when we see that three of those generation tasks are actually from one paper [96] that is doing a human evaluation of XAI methods for SE tasks. We will discuss the challenges of evaluation metrics in section 3.3.3.

RQ1.3 Results: What are the goals and objectives of each research paper reviewed?

In section 2.1.2, we demonstrated different objectives of XAI. In our analysis of the selected studies of this work, we were interested to extract different proclaimed objectives of different studies. We found three main objectives explicitly or implicitly mentioned in the selected studies as: accountability, fairness, credibility, understanding, transparency, and improvement. Accountability and credibility have been used interchangeably in different works, meaning the reliability of the model for its users. Note that these objectives are not mutually exclusive and some papers follow multiple goals so the numbers add up to greater than the number of papers.

As shown in Figure 3.6, 20 papers that is the vast majority of studies have claimed improvement as (at least) one of their purposes, accountability in 6, and understanding in 3 different papers have been mentioned. transparency and fairness are less common objectives while being the aim of 2 papers, and credibility has been mentioned only once.

Accountability and fairness are usually used in reviews/surveys [97, 96, 98] together, but the other objectives are coupled with the improvement in different research.

It's worth mentioning that while most of the objectives have been said with the same meaning as what is discussed in section 2.1.2, improvement is an ambiguous term, used for multiple intentions. In the selected studies some researchers consider the explanation as it is, as the improvement [94]. In the other words, they believe that the fact that their model offers an explanation is an improvement compared to other models.

Table 3.5: Different ML models and the tasks they have been used for and number of uses

ML model	Task (# of Uses)	Total # of uses
Random Forest	Defect prediction (8) , Code reviews	9
Decision Tree	Effort prediction, defect prediction (2), Quality assessment, Requirements quality, Requirements engineering, Code reviews	7
Regression Model	Defect prediction (5), Requirements engineering	6
TransCode	Code translation, Code autocompletion, Natural language to code	3
CodeBERT	Code translation, Code autocompletion, Natural language to code	3
Feed Forward Network	Defect Prediction (2), Quality assessment	3
Naïve Bayes	Defect prediction (2)	2
kNN	Defect Prediction, Requirements engineering	2
SVM	Quality assessment, Defect Prediction	2
CNN	self-admitted technical debt detection, Valid Bug Report	2
Bayesian Network	Quality assessment, Defect Prediction	2
RNN	Variable Misuse detection	1
Fuzzy Logic	Software Reliability Prediction	1
Transformer	Variable Misuse detection, Defect prediction	1
Code2Vec	Code Summarization	1
Code2Seq	Code Summarization	1
Fuzzy C Means	Defect Prediction	1
Kurtanovic and Maalej's Classification	Requirements Classification	1
Gradient Boosting	Defect prediction	1
Genetic Algorithms	Software Reliability Prediction	1
Rule-based	Quality assessment	1
Case-based Reasoning	Quality assessment	1

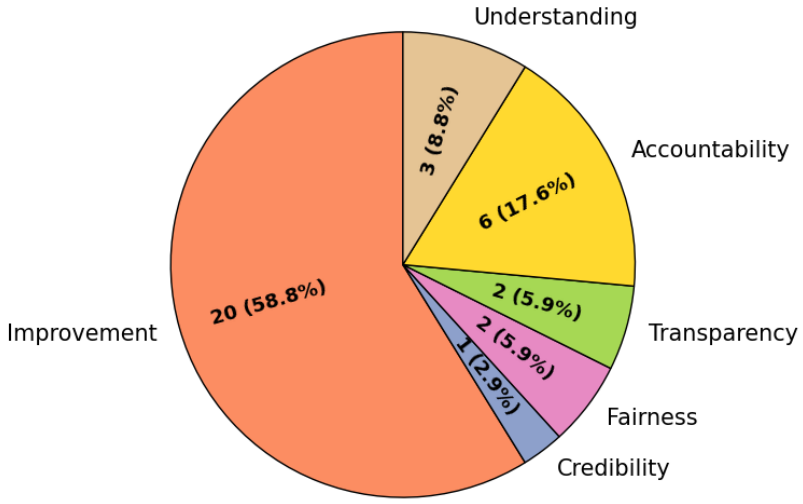


Figure 3.6: The XAI objectives stated (explicitly or implicitly) in the selected studies along with the number of papers that have mentioned them. Reported by #Papers (% proportions over all).

Tree-based models that extract rules for their decision-making and present them are among this category.

Meanwhile, some studies have provided model-agnostic XAI tools and methods to help other researchers to understand and improve their models in the future [99] or even have used XAI methods to literally improve the results of models, especially in defect prediction. We will discuss them later in section 3.3.3 where we discuss the helpfulness of XAI for SE models.

Answer to RQ1: *Among different stages of SDLC, QA and maintenance have been the most popular among the XAI researchers, and this popularity is hugely focused on the defect prediction task. This is likely due to 1) the popularity of this task among researchers and also 2) the adequacy of self-explaining traditional models like decision trees for these tasks.*
Our analysis also shows that, due to their inherent interpretability, classic models, like random forest, decision tree, regression models, and naive Bayes models, are among the most common ML models in the selected studies. DNN models are also another center of attention in terms of ML models, however generation-based tasks that are one of the main strengths of these models are still unexplored.

3.3.2 RQ2 Results: What are the Explainable AI approaches adopted for SE tasks?

In this section, we focus on the XAI methods that have been used to explain the SE models that we discussed in previous sections. This will include post-hoc XAI methods and self-explaining models.

RQ2.1 Results: What kind of XAI techniques have been used?

As we discussed in section 2.1.2, there are two types of XAI approaches: (a) models that offer some level of explanation alongside the output, and (b) post-hoc XAI techniques that are applied on the model afterwards. In this section, we are discuss both cases among our selected studies. According to our analysis, 15 self-explaining models were used in the studies that need no or a small amount of effort to interpret while only 9 models were using post-hoc methods.

Tree-based models are the most popular self-explaining models among these studies with using a diverse set of algorithms. Decision trees using the C4.5 algorithm or its developed version, the PART algorithm that uses fuzzy logic to handle the classification task [93, 100, 101, 92, 102] or random forests are widely used in our primary selected studies [91, 103, 104, 105, 106].

The explanation offered in tree-based models is usually a set of rules or highlighted features that are quite self-explanatory for the user and usually need no further processing. Thus, the explanation in these models are either in form of feature summary statistics, or feature summary visualization using plots and graphs.

Deep neural networks are another observed type of ML models that even though are known as black-box models, but some researchers believe they have internal features that can be used for interpretation, after proper processing. For instance, [107] has used backtracking of a trained CNN model to extract the key phrases of the input and discover meaningful patterns in the code comments. As an explanation, they were able to find some 1-gram, 2-gram, and 3-gram patterns in natural language format that are compared to human-extracted patterns. Their model is able to cover most of the benchmark and also offers further comprehensive patterns with more diversity of vocabulary. In another study [108], the authors also use backtracking of a CNN model but for the valid bug report determination. As the input of this task is in NL format, it is very similar to classification tasks in NLP and it generates a natural language explanation.

Transformer models are also among the promising DNN models that have outperformed state-of-the-art models in some SE tasks by leveraging the attention mechanism. There is a controversial debate about the validation of transformer model's attention mechanism as an explanation method, in recent years. Among the selected studies, there is one that have used the self-attention mechanism of a transformer as an explanation [109]. They claim to find the importance of the input tokens by constructing the attention matrices. The generated matrix as well as a respective saliency map is the explanation that they present, but no evaluation is discussed.

Looking at the post-hoc XAI methods, most techniques that are used in SE are adopted from the NLP domain. LIME (Local Interpretable Model-Agnostic) [45] is one of the widely used XAI methods in NLP

that also showed very useful in SE. In the reviewed papers, four different articles have used LIME directly in their studies [103, 104, 105, 106]. The method works based on feeding a black-box model with perturbed input data and observing the changes that happen on the output weights. This straightforward mechanism makes it ideal for NL and PL data. Perturbation can be defined on multiple levels from files in a commit to tokens in a code snippet and the results will be scores for the defined features (lines, tokens, etc.).

As we mentioned, LIME works based on input perturbation by generating synthetic instances(n) similar to a test instance (x) by a random perturbation method. The level and granularity of this perturbation depend on the task and objectives. In two of the studies that use LIME [103, 106], code tokens are considered as features that are supposed to be perturbed, while another study [105] uses tabular features of a code commit made by the developers (e.g. number of files modified, number of added or deleted lines, Number of developers who have modified the files, etc.). After the generation of synthetic instances, a defect prediction model is used to label them, and then, based on the predictions, a score between -1 to 1 is assigned to each feature. A positive score means a positive impact on the prediction and vice versa.

This model-agnostic and easy mechanism has led to the popularity of LIME among users, as another study, which is focused on the practitioners' perceptions of XAI for defect prediction models [104], claims that LIME gets the highest appeal among different XAI methods in terms of information usefulness, information, insightfulness, and information quality with more than 66% agreement rate.

The second favorite method is ANOVA (ANalysis Of VAriance), which is another model-agnostic method that calculates the importance of different features based on the impact of that feature on the Residual Sum of Squares (RSS) in a model. In our selected primary studies, ANOVA is used three times [104, 110, 111]. In addition, LIME and ANOVA have been selected as the first and second most preferred XAI methods, respectively, in a human evaluation of 50 software practitioners.

SIVAND is another model-agnostic XAI method that works based on Delta Debugging and reducing the input size, without changing the output [112]. The big picture is very similar to LIME but instead of just finding the most important atomic unit (token or character), it removes redundant units so the input gets smaller in size, but the prediction remains the same. They have tested their method on Code2Vec, Code2Seq, RNN, and Transformer models on two tasks of Method Name Prediction and Variable Misuse Detection and have a qualitative example-based evaluation of their models. For the Transformer model, they validate their performance by comparing the similarity of SIVAND and attention scores.

PyExplainer is another XAI method inspired by LIME and focused on Just In Time (JIT) defect prediction. It is a model-agnostic local rule-based technique that challenges the LIME's random perturbation mechanism by generating better synthetic neighbors, and explanation more unique and consistent. According to the mentioned criteria, PyExplainer outperforms LIME on Random Forest and Logistic Regression

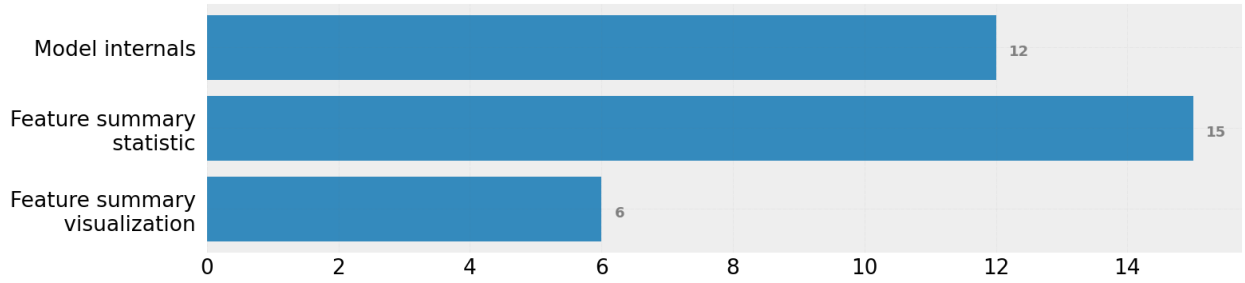


Figure 3.7: Different types of explanation offered in the selected studies.

models for defect prediction.

Beside these model-agnostic methods, there are some other XAI techniques that are used for interpreting specific ML models. For instance, deconvolution is a method that exploits the structure of CNNs to extract the importance of features[107]. Flexible Code Counter/Classifier(CCFlex) is also an interesting model that was originally designed to classify lines of code but is used for code review and finding guideline violations[113].

Some other XAI methods are also mentioned in a study asking about software practitioners' perceptions of the goals of XAI methods for defect prediction models[104]. As mentioned earlier, LIME and ANOVA were the most preferred methods, while other techniques like Variable Importance, Partial Dependence, Decision Tree, BreakDown, SHAP, and Anchor were less favorable.

RQ2.2 Results: What kind of explanations do they offer?

We analyzed the selected studies in terms of the type of explanation that they offer. Following the previous discussion about self-explaining models, many models deliver "Model internals" as the explanation. Extracted rules and structures of tree-based models or the attention matrix of Transformer models are in this category. However, some works go further and more than the local explanations, presenting the "Feature summary statistic" or "Feature summary visualization" of their explanations with different visualization methods. The distribution of each type is shown in Figure 3.7. As it can be understood from the plot, some models have one, while others offer more types of explanation.

In the category of "Feature summary visualization", there are multiple XAI methods presented in the studies. While there are works that are satisfied by "Raw declarative representations" of their interpretations, many XAI methods implement more informative ways. "Natural language explanation" is used five times, while "Saliency maps" and "Plots and charts" each are used three times. There are also two methods that present their results in interactive tools to the users. The distribution of different visualization techniques in the studies is illustrated in Figure 3.8.

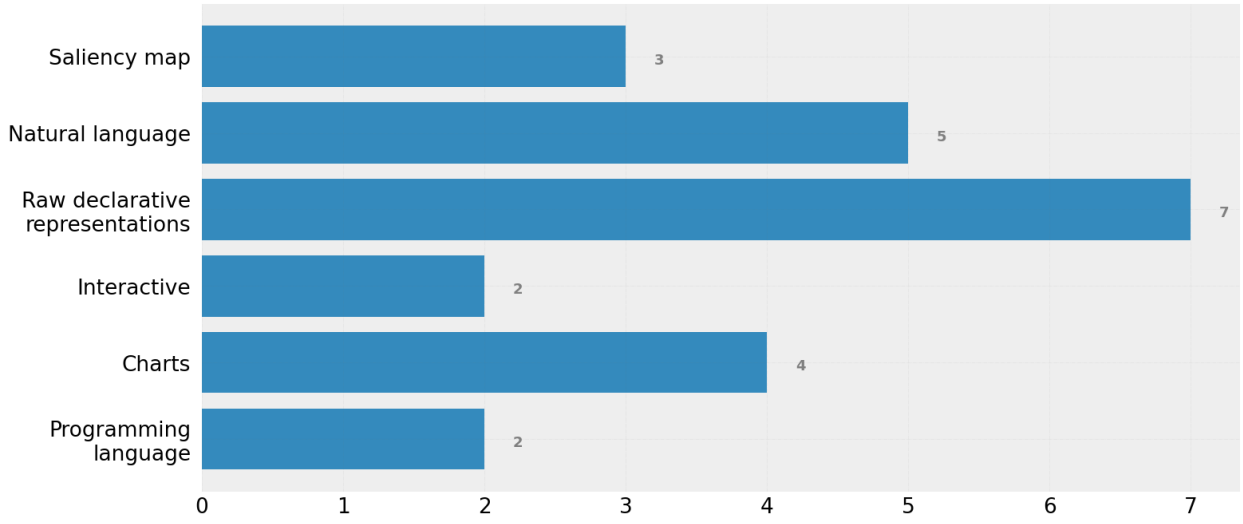


Figure 3.8: Distribution of different visualizations used as explanation methods in the selected studies.

Answer to RQ2: *Our investigation shows that beside different ML models that have a level of interpretebility and are quite common for SE tasks (such as random forest and decision trees), there is a variety of model-agnostic XAI methods that are used in SE to explain black-box models. LIME which is a perturbation XAI method is the most favorite method in the selected studies, while ANOVA is the second most-used method. There are also some other proposed methods used in the studies such as deconvolutioning for CNN models, or SIVAND which has similarities with LIME.*

The explanations offered by XAI methods can get categorized as "Feature summary statistic", "Model internals", or "Feature summary visualization" that have been used in the studies, respectively in descending order.

3.3.3 RQ3 Results: How useful XAI have been for SE?

In this section, we are going to focus on the helpfulness of XAI methods that are used for SE tasks.

RQ3.1 Results: What are the objectives of XAI applying on SE and how they have been useful?

In section 3.3.1, we discussed different objectives of research in our selected studies. Some of them have used explanation as an additional output for their users, while a few others used the XAI methods to improve the performance of the models.

One interesting representation of these improvements can be seen in defect prediction. While many defect prediction models only classify if a file or a class is faulty, some researchers have taken advantage of XAI

methods to specifically find the buggy line or tokens of the code [103, 104, 106]. This is interesting since it improves defect prediction's practicality and its potential for adoption by industry.

SIVAND is another successful XAI method, where its finding improves the performance of the code models it is applied to, and also offers valuable knowledge about them [112]. They are able to considerably reduce the size of the inputs, and yet achieve the same accuracy. This means a great reduction in the models' required time to make a prediction. Furthermore, they offer interesting knowledge about code models by claiming the presence of alleged patterns in them. For instance, they believe while Transformer models can understand more "global" information in codes, RNN models are prone to local information. They also believe these models are quite vulnerable to perturbations as they rely on too few features.

Another noteworthy and exemplary use of XAI can be seen in another work [108] where by backtracking a trained CNN network and manually inspecting the explanation, the authors are able to find valid bug report patterns. Further analysis of the identified bug reports, they verify the importance and effect of XAI on their model and the problem.

RQ3.2 Results: What are the means of evaluating XAI in SE?

Looking at the evaluation of the XAI methods (not the ML models) across the selected papers, we can see that the struggle is apparent. While studying these selected papers, we observed that there is little consistency among selected papers in terms of evaluation. Many studies have totally ignored the evaluation phase and only offer some visualization for their XAI with no evidence of its quality or reliability.

Most existing XAI evaluation methods [114], [107], [113] are qualitative and use human subjects to assess the explanations. The problem with this approach, specially with small scale subject pools, is that they are heavily biased and may not be generalizable to other users.

In an attempt to find out how thoroughly evaluate XAI in SE, in one work [96] the authors asked 43 software engineers in 9 workshops to express their explainability needs for some specific tasks.

With respect to qualitative assessment of XAI methods in SE, there are some works that based on their task (e.g., defect predicion), are able to define more common metrics such as Mean Squared Root (MSR), or Wald test [110]. In a few other studies, researchers defined innovative metrics or used some less-known metrics from generic the XAI domain. For instance, in one work [99] the authors measure the percentage of unique explanations generated by XAI method as a metric. Another work [100] utilizes the average length of the rules that their XAI extracts and another study [105] uses Variable Stability Index (VSI) and Coefficient Stability Index (CSI), as metrics.

Answer to RQ3: *Our findings show that some XAI methods were quite helpful and were applied either for providing proper explanations to researchers that led to finding interesting patterns in the model's decision-making, or for increasing the granularity of models' results, leading to the higher precision for the respective SE task.*
We also noticed some confusion and a lack of standard routines among the researchers in terms of evaluating XAI methods that are usually compensated by defining innovative and task-specific metrics or using qualitative evaluation by a human.

3.4 Discussion

In this systematic literature review, we analyzed the current state of XAI methods in SE tasks. The results showed that software maintenance is the most attractive activity for researchers in the field, with 68% of the tasks being used for interpretation. After that software development, software management, and software requirements each have a smaller share of the total tasks. The most popular task in the studies is defect prediction. We also found that 15 self-explaining models were used in the studies while only 9 models were using post-hoc methods. Among these self-explaining models in the studies the Tree-based models are the most popular.

We also found that some studies have used explanation as an additional output for their users, while a few others used XAI methods to improve the performance of the models. Additionally, we found that there is a lack of consistency among selected papers in terms of evaluation methods and metrics, making it difficult to compare the effectiveness of different XAI approaches. This study highlights the potential of XAI for improving the performance of SE tasks, but also the need for more consistent and thorough evaluation methods in the field.

There are also several threats to the validity of this systematic literature review that should be considered when interpreting the results. First, the selection of studies for this review was based on a set of predefined inclusion and exclusion criteria, which may have led to bias in the selection of papers. Additionally, the search strategy used to identify relevant studies may have missed some relevant papers that were not included in the final analysis.

Another threat to the validity is the potential for researcher bias in the analysis and interpretation of the data. The synthesis and interpretation of the data were based on the subjective judgment of the researchers, which may have led to bias in the conclusions drawn from the studies.

Finally, the lack of standardization in the evaluation methods used in the studies presents a threat to the validity of the results. This lack of consistency in terms of evaluation methods and metrics makes it difficult

to compare the effectiveness of different XAI approaches.

Also it's worth noting that there is some confusion and lack of consistency in terminology in the literature. For instance, some works used the terms "explainability" and "interpretability" interchangeably, which can lead to confusion when trying to compare and understand the results of different studies. This highlights the need for more consistent terminology and definitions in the field of XAI in SE in order to facilitate clearer communication and comparison of results.

It's important to note that these results are based on the knowledge cut-off of 2022, and new developments and research in the field of Explainable AI in software engineering may have emerged after that date.

Overall, our literature review shows that XAI has a lot of potential for improving the performance of SE tasks, but there is a need for more consistent and thorough evaluation methods and terminology in the field. Additionally, there is a gap between the application of machine learning in SE and the utilization of XAI in the field, which presents opportunities for future research as there are many ML models that have already been used in SE, but are not explained at all.

Table 3.6: List of the selected 24 papers. Conference: [C], Journal: [J]

Ref	Title	Authors	Publication Venue	Publication Year
[96]	Investigating Explainability of Generative AI for Code through Scenario-Based Design	J. Sun, Q. V. Liao, M. Muller, M. Agarwal, S. Houde, K. Talamadupula, J.D. Weisz	Intelligent User Interfaces [C]	2022
[106]	Predicting Defective Lines Using a Model-Agnostic Technique	S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, K. Matsumoto	Transactions on Software Engineering [J]	2022
[103]	JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction	C. Pornprasit, C. Tantithamthavorn	Mining Software Repositories (MSR) [C]	2021
[105]	Just-in-Time Defect Prediction Technology based on Interpretability Technology	W. Zheng, T. Shen, X. Chen	Dependable Systems and their Applications (DSA) [C]	2021
[104]	Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models	J. Jiarpakdee, C. Tantithamthavorn, J. Grundy	Mining Software Repositories (MSR) [C]	2021
[99]	PyExplainer: Explaining the Predictions of Just-In-Time Defect Models	C. Pornprasit, C. Tantithamthavorn; J. Jiarpakdee, M. Fu, P. Thongtanunam	Automated Software Engineering (ASE) [C]	2021
[97]	Towards Reliable AI for Source Code Understanding	S. Suneja; Y. Zheng; Y. Zhuang; J. Laredo; A. Morari	Symposium on Cloud Computing [C]	2021
[112]	Understanding Neural Code Intelligence through Program Simplification	M. R. I. Rabin, V. J. Hellendoorn, M. A. Alipour	European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) [C]	2021
[92]	Application of machine learning techniques to the flexible assessment and improvement of requirements quality	V. Moreno, G. Génova, E. Parra, A. Fraga	Software Quality [J]	2020
[108]	Deep Learning Based Valid Bug Reports Determination and Explanation	J. He, L. Xu, Y. Fan, Z. Xu, M. Yan, Y. Lei	International Symposium on Software Reliability Engineering (ISSRE) [C]	2020
[113]	Recognizing lines of code violating company-specific coding guidelines using machine learning: A Method and Its Evaluation	M. Ochodek, R. Hebig, W. Meding, G. Frost, M. Staron	Empirical Software Engineering [J]	2020

[110]	The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models	C. Tantithamthavorn, A. E. Hassan, K. Matsumoto	Transactions on Software Engineering [J]	2020
[109]	An Explainable Deep Model for Defect Prediction	J. Humphreys, H. K. Dam	Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE) [C]	2019
[107]	Neural Network-Based Detection of Self-Admitted Technical Debt: From Performance to Explainability	X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, J. Grundy.	Transactions on Software Engineering and Methodology (TOSEM) [J]	2019
[95]	Requirements Classification with Interpretable Machine Learning and Dependency Parsing	F. Dalpiaz, D. Dell'Anna, F. B. Aydemir, S. Çevikol	Requirements Engineering (RE) [C]	2019
[94]	Towards a More Reliable Interpretation of Defect Models	J. Jiarpakdee	International Conference on Software Engineering: Companion (ICSE-Companion) [C]	2019
[98]	Explainable Software Analytics	H. K. Dam, T. Tran, A. Ghose	International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER) [C]	2018
[114]	Superposed Naive Bayes for Accurate and Interpretable Prediction	T. Mori	International Conference on Machine Learning and Applications (ICMLA) [C]	2015
[91]	A novel method for software defect prediction: Hybrid of FCM and random forest	T.P. Pushphavathi, V. Suma, V. Ramaswamy	International Conference on Electronics and Communication Systems (ICECS) [C]	2014
[102]	Software defect prediction using Bayesian networks	A. Okutan, O. T. Yıldız	Empirical Software Engineering [J]	2014

[100]	Towards Interpretable Defect-Prone Component Analysis Using Genetic Fuzzy Systems	T. Diamantopoulos, A. Symeonidis	Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE) [C]	2014
[93]	Software Effort Prediction: A Hyper-Heuristic Decision-Tree Based Approach	M. P. Basgalupp, R. C. Barros, T. S. da Silva, A. C. P. L. F. de Carvalho	Symposium on Applied Computing [C]	2013
[101]	Machine-Learning Models for Software Quality: A Compromise between Performance and Intelligibility	H. Lounis, T. F. Gayed, M. Boukadoum	International Conference on Tools with Artificial Intelligence [C]	2011
[111]	Evaluation of preliminary data analysis framework in software cost estimation based on ISBSG R9 Data	Q. Liu, W. Z. Qin, R. Mintram, M. Ross	Software Quality [J]	2008

Chapter 4

XAI for Pre-Trained Code Models: What Do They Learn? When They Do Not Work?

As discussed in chapter 3, there are several areas in ML4SE where the ML models have been quite studied but there is not much explanaition study for those tasks or models. To fill part of this gap, in this section, we focus on the two state-of-the-art pre-trained code models, i.e., CodeBERT and GraphCodeBERT with the three understanding & generation-specific downstream tasks, i.e., Code Document Generation (CDG), Code Refinement (CR), and Code Translation (CT) tasks. These are among the models and tasks that where not studied in the past but yet have been showed great performance in the ML4SE domain. We start by showing examples of these specific models and tasks that can show the need for explanation. Then, we explain our methodology, and answer the research questions.

4.1 Motivating Analysis

Example 1: CodeBERT correctly generates meaningful documents, which do not exactly match with the ground-truth. Figure 4.1, presents two sample input methods (i.e., *f_translate_key*() and *_equal_values*()). For the method *f_translate_key()*, we found that the generated document (‘‘Translate a key”) is incorrect when compared to the ground-truth (‘‘Translates integer indices into appropriate names”), but semantically conveys the same message. Similarly, the generated document for method *_equal_values()*, (‘‘Compare two arrays”), does not match with the ground-truth (‘‘Checks if the parameter considers two

```
def _equal_values(self, val1, val2):
    if self.f_supports(val1) != self.f_supports(val2):
        return False
    if not self.f_supports(val1) and not self.f_supports(val2):
        raise TypeError('I do not support the types of both inputs (`%s` and `%s`), therefore I
            cannot judge whether the two are equal.' % (str(type(val1)), str(type(val2))))
    if not self._values_of_same_type(val1, val2):
        return False
    return comparisons.nested_equal(val1, val2)
```

Gold document: Checks if the parameter considers two values as equal .
Best prediction: Compare two arrays .

```
def f_translate_key(self, key):
    if isinstance(key, int):
        if key == 0:
            key = self.v_name
        else:
            key = self.v_name + '_%d' % key
    return key
```

Gold document: Translates integer indices into the appropriate names
Best prediction: Translate a key .

Figure 4.1: Two code snippets with their original and the CodeBERT's predicted documents, where the prediction is good.

values as equals"), but describes the same functionality. This motivating example indicates that one explanation for CodeBERT's weak performance may just be an inappropriate evaluation metric. This is a common problem with an automated metric-based evaluation of tasks that contain generating new sentences. Although it is common to use text similarity-based metrics such as BLEU [115] to evaluate a Transformer for text generation, an XAI method can help distinguish between cases where the scores are low because the model is poor and the cases where the scores are low because the evaluation metric is not well suited for the task/data under study, such as this example.

Example 2: CodeBERT incorrectly generates documents for a simple method. Figure 4.2 presents a sample code snippet, the *file*() method, which is short and easy to understand. In this example, we found that the generated document (``Write a file") does not match the ground-truth (``Reads the body to match from a disk file"), but both sentences share some vocabulary (the word "file"). Based on the generated document, it seems that CodeBERT model is able to understand that this method aims to apply an operation on a file, but it fails to determine whether the operation is read or write. Although "*read()*" is called in the source code and there is no token to misguide the model to choose "*write()*", CodeBERT is still incorrectly generated the document. This motivating example indicates that CodeBERT may generate incorrect program documents for even simple codes. Yet, little is known about why it fails in such scenarios.

```
def file(self, path):
    with open(path, 'r') as f:
        self.body(str(f.read()))
```

Gold document: Reads the body to match from a disk file .
Best prediction: Write a file

Figure 4.2: An example of a simple method that CodeBERT fails to predict correctly.

Source code:

```
public int getBeginIndex() {return start;}
```

Target code:

```
public int getBeginIndex() {return start;}
```

Generated code:

```
public virtual int GetBeginIndex(){
    return start;
}
```

Figure 4.3: An example of a simple method that needs no change, but GraphCodeBERT fails to translate correctly.

Example 3: GraphCodeBERT incorrectly translates a simple one-line method. Figure 4.3 presents an example input method, *getBeginIndex*(), which is a one-line simple method. In this example, the source code (in Java) is valid in the target language (C#) as is, and doesn't need any change. However, we can see that the model makes an incorrect change by adding the virtual keyword. Also, it is interesting that the model understands the different naming conventions of Java and C# and capitalizes the first letter of the method name. According to this motivational example, we can see that even though the general accuracy of the model is good (in terms of BLEU score), it may be unable to translate the simplest of codes and the reason is unknown to us.

Prior studies in other fields such as image classification, pointed out that attention mechanisms can be used to understand or compare the models. Sometimes, two different models may predict the correct answer in a sample, but have a totally different focus and attention patterns. In this situation, providing an explanation for the decision-making process of the models may help the users to better compare their quality, or to trust them.

The above three examples suggest that there are serious ambiguities about the decision-making process of CodeBERT and GraphCodeBERT, such as the problem with some very simple codes, despite the good overall accuracy of the model. We believe explaining CodeBERT and GraphCodeBERT can potentially be beneficial to understanding the underlying reasons behind a poor quality output that potentially results in alternatives to improve the models.

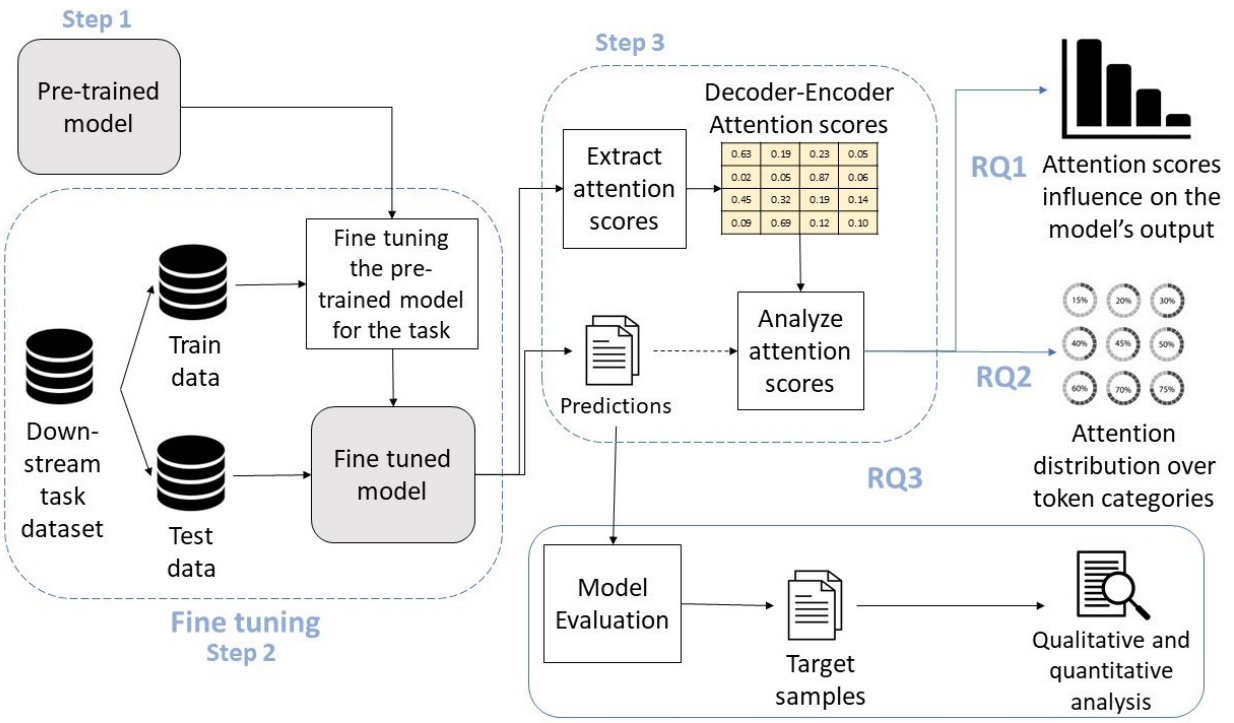


Figure 4.4: An overview of the experiments.

4.2 Research Methodology

In this section, we explain CodeBERT and GraphCodeBERT which are used as pre-trained code models in this study (step 1). Then we describe how we have fine-tuned and used them for each task (step 2) and finally demonstrate the attention scores and how we attained them from the models (step 3). Figure 4.4 illustrates an overview of the process and the following subsections present the details per step.

4.2.1 Collecting Pre-trained Code Models (Step 1)

In this step, we simply download the pre-trained encoders of two transformer-based models, CodeBERT and GraphCodeBERT that will be our base models for fine-tuning.

CodeBERT

CodeBERT [17] is a bimodal multi-lingual pre-trained model for programming language (PL) and natural language (NL). It has a multi-layer Transformer encoder, trained on Masked Language Modeling (MLM) and Replaced Token Detection (RTD) with both NL and PL as inputs. The model has a similar architecture as BERT [26] and showed promising results on multiple downstream tasks such as Code Translation, Clone Detection, Defect Detection, etc [17].

GraphCodeBERT

GraphCodeBERT [31] is a pre-trained model similar to CodeBERT that considers the semantic-level structure of the code as well. It uses data-flow in the pre-training stage and uses MLM, alongside Edge Prediction and Node Alignment as pre-training tasks. Having this feature included, the model is able to improve the results on its benchmark tasks comparing to CodeBERT, but as we will show in our experiments, in tasks like Code Document Generation, it shows a great drawback.

4.2.2 Fine-tuning Models (Step 2)

In this study, we used the default settings for fine-tuning the two models that we used in our experiments (CodeBERT and GraphCodeBERT) as they have led to the best possible results as reported in the literature. For CodeBERT experiments, we used the pipeline provided by the CodeXGLUE [32] benchmark. This model has a six-layer Transformer decoder on top of the pre-trained CodeBERT encoder, followed by a dense layer to generate the output token-by-token. Similarly, for the GraphCodeBERT model, we used the default settings of the model to replicate the reported results in the literature. This may affect the generalizability of our findings to other settings or models. However, this choice allows us to compare our results with the state-of-the-art and to ensure the reproducibility of our results.

During training, we use default values as follows: max source length of 256 and max target length of 128 with the learning rate of 5e-4, with 16 as batch size and training it for 100 epoches.

Both models follow the same steps to generate the output. After training a model on a downstream task on the respective training dataset, the model generates the output for each test data item, token by token. That is, in the inference time, in each step, some tokens (depending on the beam size, set in the model) are being chosen from the potential predicted candidates, and this process is repeated (new tokens are added to the candidate output string) until the model generates the end-of-sentence token, which indicates the end of prediction.

Downstream tasks

In this study, we choose three different downstream tasks which are: CDG, CR, and CT. Having three generation-based tasks, consisting of a uni-lingual (CR) and a bi-lingual (CT) code-to-code and also a code-to-NL (CDG) downstream task under experiment, we believe this research covers a diverse set of common CodeBERT and GraphCodeBERT applications.

Code Document Generation

CDG as our first downstream task is generating natural language comments for a given method's source code. For this task, CodeBERT achieves an overall BLEU score of 17.83 (19.06 and 17.65 per Python and Java, respectively).

As we mentioned in 4.2.1, GraphCodeBERT did not have this task on its benchmark, so we implemented the model and fine-tuned it for this task. The BLEU scores that model achieved for this task are 5.3 and 4.21 for Java and Python, respectively. These results are interestingly lower than CodeBERT.

We use the preprocessed and filtered Python and Java dataset of the CodeSearchNet [116] for this task. In this dataset, examples that can not be parsed into an AST, or have documents that are not in English or have some special tokens (e.g. <img ...> or https:...) or are shorter than 3 or longer than 256 tokens, are filtered. This dataset contains 280,652 samples of Python code from 12,361 unique repositories and 35,170 samples of Java code, from 4,123 unique repositories 4.1.

Each sample contains one method, with its actual documentation, the "docstring" description of the method, that we call the "gold document" in this study, its repository, path, and the method name.

Code Refinement

The goal of this task is to automatically fix the bugs in a given faulty source code. The input is a buggy Java function and the output is the fixed code. We use the medium dataset from a previous work [14] which contains 65,454 buggy Java functions with their fixed versions. In this dataset, all function and variable names have been normalized 4.1. BLEU scores for this task are 91.07 and 91.31 and exact match accuracies are 5.16 and 9.1 for CodeBERT and GraphCodeBERT, respectively.

Code Translation

This task aims to migrate source codes from a programming language to another one. We use CodeXGLUE dataset which is collected from multiple public resources with 11,800 samples of Java and C# functions mapped to each other. We take Java as our source language and C# as the destination 4.1. The BLEU score and exact match accuracy for CodeBERT on Java to C# task are 79.92 and 59.0, respectively. GraphCodeBERT however, achieves better results with 80.58 and 59.4 as BLEU score and exact match accuracy.

4.2.3 Extracting the Attention Scores (Step 3)

Transformer models are able to comprehend long dependencies among words in a sentence (or tokens in a code snippet) by benefiting from the attention mechanism. As it is shown in the following formula, an attention

Table 4.1: Dataset statistics

Task	Input\Output Languages	Training	Validation	Test	Total
Code Translation	Java\C#	10,300	500	1,000	11,800
Code Document Generation	Java\NL	164,923	5,183	10,955	181,061
	Python\NL	251,820	13,914	14,918	280,652
Code Refinement	Java\Java	52,364	6,545	6,545	65,454

mechanism basically works with a key (k), query (q), and value (v), and d_k denotes the dimensionality of the key. In its simplest form, it calculates the similarity between the query and key-values as:

$$\text{Attention}(q, k, v) = \text{softmax}\left(\frac{qk^T}{\sqrt{d_k}}\right)V$$

Where keys and queries are the elements in the sequence. Calculating all the dot products of $q_i.k_j$ will result in a matrix where each row represents the attention weight for a specific element i to all other elements in the sequence. Afterward, a softmax layer and multiplication with the value vector will be applied to the matrix to obtain the weighted mean. This means every dependency between each two elements will be considered in the final output.

As explained in Section 2.2.2, attention is a reasonable XAI method to explain CodeBERT and GraphCodeBERT. To calculate the attention per token, we need the weights for the encoder-decoder attention layers. Since we have six Transformer decoder layers stacked up, we have six attention layers. Rather than somehow aggregating these 6 layers' attention values into one metric, we decided to keep all layers' data and analyze the different layers' roles in explaining the outputs. The attention weights are available inside the model, but the Transformers library that is used by CodeBERT and GraphCodeBERT, doesn't provide them by default. Hence, we changed the model's implementation to collect them, as well.

As we mentioned before, the model has an encoder-decoder attention mechanism and output tokens are generated one by one. So basically for each generated output, we have an attention vector, distributed over all of the input tokens.

The attention weight for each token is the output of a softmax layer, therefore it has a value between 0 and 1. However, in practice, most attention scores are in a much smaller range. So for the sake of visualization and only in our visualized examples, we normalize the scores in each sample, to a zero-to-one range. We represent each attention value as the token's background color opacity.

Preprocessing. The pre-trained CodeBERT model uses the "Roberta-base" tokenizer and the GraphCode-

BERT model, use "Microsoft/graphcodebert-base" tokenizer, and both tokenizers are not code-specific. So as shown in Figure 4.5 example, the actual "code tokens" may be broken into smaller pieces, each considered one token, having its own attention score. For example, as you can see in Figure 4.5, the model tokenizes the name of the method, which is "delete_function" (a single token, tokenized by python interpreter), as three different tokens of "delete", "_", "function" and this means it generates three different attention scores for it.

This is not ideal for our analysis because we care about the code tokens as a whole and their roles (e.g., keyword, variable name, method name, etc). To handle this discrepancy, we use Tree-sitter [1] to parse our source codes to get the actual "code tokens". Then for each token, we consider the mean of its component's attention as the final attention for that token. From now on, when we talk about tokens, we mean code tokens.

4.2.4 Evaluating the Models (Step 4)

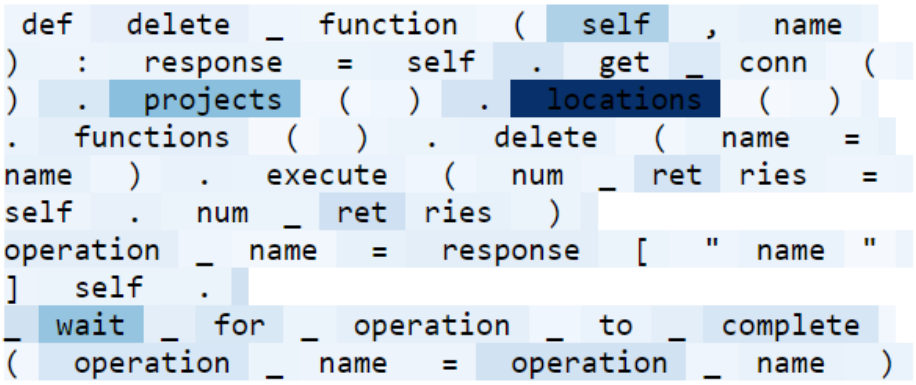
To evaluate the CDG downstream task, we use smoothed BLEU-4 score that is used in CodeBERT's original study [17] and is commonly used by baseline document generation techniques [117]. For other tasks, we use simple BLEU-4 score. The BLEU-4 is the only evaluation metric we use in this study and from now on, unless we explicitly say otherwise, when we use BLEU score we are referring to BLEU-4 score.

In our study, BLEU score [115] calculates the n-gram overlap of the prediction and the gold document or code snippet, for all n-gram orders up to four. In other words, it counts the number of all n-gram (from one to four) sub-sequences of the predicted output that are also in the gold string and it gives all n-gram scores an equal weight to calculate the final score. Since in CDG, the generated sentences are usually short, and the higher-level n-gram are not likely to have an overlap, CodeBERT uses a smoothed version [118] that compensates it, by giving additional counts to higher-level n-gram overlaps.

4.3 Research Questions

The primary goal of this study is to explain in which scenarios a pre-trained code model works well and in which cases it may not do well. So that the research community can adjust their effort to designing better models for the software engineering tasks that benefit from a pre-trained code model. The motivation behind this goal is that embedding models are usually complex and applying them on a software downstream task such as CDG, CT, and CR sometimes creates nice outputs and sometimes disappoints. Now the question is can an XAI method such as "attention mechanism" help us shed light on the limitations and strengths of

[1] https://github.com/tree-sitter/tree-sitter



Gold document: Deletes the specified Cloud Function .
Best prediction: Deletes a function .
BLEU score: 0.27
Overlap: 0.20

Figure 4.5: An example of the model's tokenization and the attention assignments, per token.

code embedding? To achieve this goal, we focus on two state-of-the-art neural program embedding models (CodeBERT and GraphCodeBERT) and borrow techniques from XAI to explain the underlying decisions made by these models. Among the existing XAI methods, in this work, we study the attention mechanism, as it is one of the main approaches in XAI and is indeed already a part of some of the mainstream neural code embedding techniques. Therefore, our empirical study targets the following research questions:

(RQ1) Is the attention mechanism suitable to explain the pre-trained code models?

Motivation. Prior studies pointed out that attention mechanisms can be used to understand which tokens contribute to predictions of attention-based models [43, 119]. However, some studies also argued against it [120]. They claim that in many situations, there is no correlation between the attention weights and other methods like gradient-based feature importance analysis. These studies have also observed that in some cases, input samples with different attention distributions have ended up with equal outcome. In this RQ, we want to justify the use of attention as our XAI method. We specifically want to know if attention outputs on CodeBERT and GraphCodeBERT, provide any meaningful information that helps explain (and potentially improve) these models. To do so, the hypothesis to test is that "attention has a significant contribution in the final decision of the model" and thus studying the attention values will shed light on why certain decisions have been made. We expect that when the model generates a token X in the output, if X or a token similar to X (with a specific definition of similarity) exists in the input, then the model focuses its attention on X more than other tokens.

Design.

In order to find out if attention is a suitable mechanism to explain CodeBERT and GraphCodeBERT, we

		Output				
		public	override	Object	Clone	
input	public	**0.4**	0.18	0.22	0.05	
	Record	0.3	0.46	0.11	0.28	
	clone	0.06	0.15	0.23	**0.27**	
	()	0.09	0.03	0.13	0.15	
	{	0.04	0.05	0.14	0.09	
	return	0.05	0.04	0.1	0	…
	copy	0.02	0.06	0.04	0.04	
	()	0.03	0.02	0.02	0.09	
	;	0.01	0	0	0.01	
	}	0	0.01	0.01	0.02	
Sum		1.00	1.00	1.00	1.00	
Similar token rank		1	not found	not found	2	

Input:

```
public Record clone() {return copy();}
```

Output::

```
public override Object Clone(){return this;}
```

Figure 4.6: An example of the attention impact calculated for an example input and output with attention weights. Each column in the table is the attention scores for the output token, distribution over different input tokens. Blue cells show top-3 attentions in each column. If one of the top-3 tokens in each column corresponds to a row with identical (or similar) token, we count that as a hit.

need to inspect if attention weights have a relatively direct and strong relationship with the model's output and we expect such a dependency to exist. That will mean analyzing and explaining the attention score, will be a good representation of the model.

To evaluate this relationship between attention scores and model's outputs, we take two different approaches, one for the CDG task which has NL output, and another for CT and CR. Both approaches share the same idea (examining the impact of attention weights on the model's outputs), but are different in implementation, to adjust to their corresponding task's input/output type.

As we mentioned in 4.2.1, in our generation-based tasks, the models generate the outputs in multiple steps token by token, and for each step, we have the respective attention weights. Therefore, we analyze the attention scores in each step.

To test our hypothesis in this RQ ("Are source code words that are used in the output among the high attention ones?"), in each step that one output token is generated, we look for its equivalent (case insensitive equal) token in the input and its attention rank, compared to others in the sample. If there is no such token, we ignore it. For instance, when a "for" token is generated by the model as an output token, we look at the attentions on input tokens and sort them based on their attention score. If there is no "for" token, in the input, then it's a new token that was generated (and not being copied) and we skip it. But if there is one, then we record its rank.

It is worth mentioning that our experiment design is a bit conservative, since we ignore the potential

indirect impact of attention scores on output. For example, assume a new token "C" is generated in the output (chosen from the vocabulary). Although there is no equivalent token to "C" in the input but the model may have chosen "C" partly because of high attention to other tokens "A" and "B", where they are always in the same context as "C" in the training phase. This means that the actual contribution of the attention mechanism to the model's decision can be higher than what our experiments will suggest.

For instance in Figure 4.6, each column and row represent an output and input token, respectively. The first column is the first step in output generation, where the model has generated the token "public". Now each row for this column shows the attention of each input token for "public".

In this example, we can see that the input token "public" has the highest attention score among input tokens for output token "public". Similarly, studying the attention scores for the generated token "clone"; the similar token "clone" has the second rank after "record". We then repeat this process for every generated token by the model to see whether we can find a pattern between generating a token in the output and its corresponding input token's high attention score or not.

We use the rank rather than the raw scores, since even though the attention score range is always between 0 and 1, it may vary a lot between different samples. In addition, the tokens in one code snippet may have very close attention scores. Therefore, in order to compare the token's attention scores, we sort the tokens based on their attention score in their respective code snippet and ignore the absolute values.

Furthermore, using absolute ranks we be deceiving when input length is variable, in different samples. For instance, while having the rank 9 among 100 input tokens is good, having the same rank among 12 inputs is not. Therefore, we are reporting the normalized rank of the tokens between 0 and 100. So a rank less than 1 means the token was among the top 1% attention scores.

A final design decision for RQ1 is that we have multiple layers in our decoder, each having its own attention weight vector. Since each layer would have its unique attention focus mechanism (e.g., one layer maybe focusing on names and another one focusing on syntax, etc.), we apply the analysis per layer and report them all.

Results. As we said earlier, we are interested in tokens that appear in the output and also exist in the input. These are basically the situations where the model has decided to copy a token directly from input, instead of choosing one from the vocabulary. We analyzed our test dataset to analyze the frequency of such decisions.

As in Table 4.2, for the CR task, more than 94% and 99% of the output tokens are available in the input for GraphCodeBERT and CodeBERT, respectively. This ratio drops to 52% and 70% in CT for GraphCodeBERT and CodeBERT, respectively; which is expected because of the uncommon tokens between the source and destination languages. For the CDG task, this ratio is between 30% and 40% for different

Table 4.2: The proportion of the tokens that appear in the output and also exist in the input for each sample and the total number of generated tokens for each task

Task	Model	% of repeated tokens	Total number of generated tokens
Code Translation	CodeXGLUE	70.55%	62,958
	GraphCodeBERT	52.53%	62,715
CDG_java	CodeXGLUE	35.50%	83,015
	GraphCodeBERT	30.45%	78,988
CDG_python	CodeXGLUE	40.63%	102,804
	GraphCodeBERT	31.76%	105,690
Code Refinement	CodeXGLUE	99.97%	845,692
	GraphCodeBERT	94.72%	838,631

languages and models. This is an anticipated fall since the output for this task is natural language and therefore, there are lots of NL tokens in the output to form a meaningful sentence. The models learn to rely less on copying mechanism, and rather focus on generating tokens. Even in cases that the model decides to use a word from the input code, it's expected that its form would be changed to be suitable for a human readable sentence (e.g. tenses in verbs).

It's noteworthy that the ratio of newly generated tokens for GraphCodeBERT is always higher than in CodeBERT for all tasks. This means GraphCodeBERT tends to be more generative and less prone to copy the same tokens from the input for the output.

Figure 4.7 shows the average normalized rank for an input token X based on its attention score, whenever the model have generated a token equivalent to X (case insensitive equal), as output. As we mentioned earlier in the design section, we only considered the output tokens that their equivalent token also existed in the input. This means if a new token is generated (not available in the input), we have skipped it (and reported them separately, see Table 4.2).

An interesting pattern is the decrease of the described rank throughout the layers. In all tasks and models, the last three layers have a much smaller average normalized rank compared to the first layers. This means that tokens with high attention scores in the last layers, have a greater chance of ending up as the output token.

In general, these results show the significant influence of attention scores on the final output, especially in CR and CT tasks. For example, in CT with both models, in the last three attention layers as in Figure 4.7a, the average normalized rank of the token is less than 2% in CodeBERT and less than 3% for GraphCodeBERT. Also, further experiments showed that more than 80% of the times, the rank is less than 3 in both models. The code refinement task also shows a similar result in Figure 4.7d where the average normalized

rank for the last three layers is always less than 4%. Also, our analysis shows that almost 70% of these tokens are among top-3.

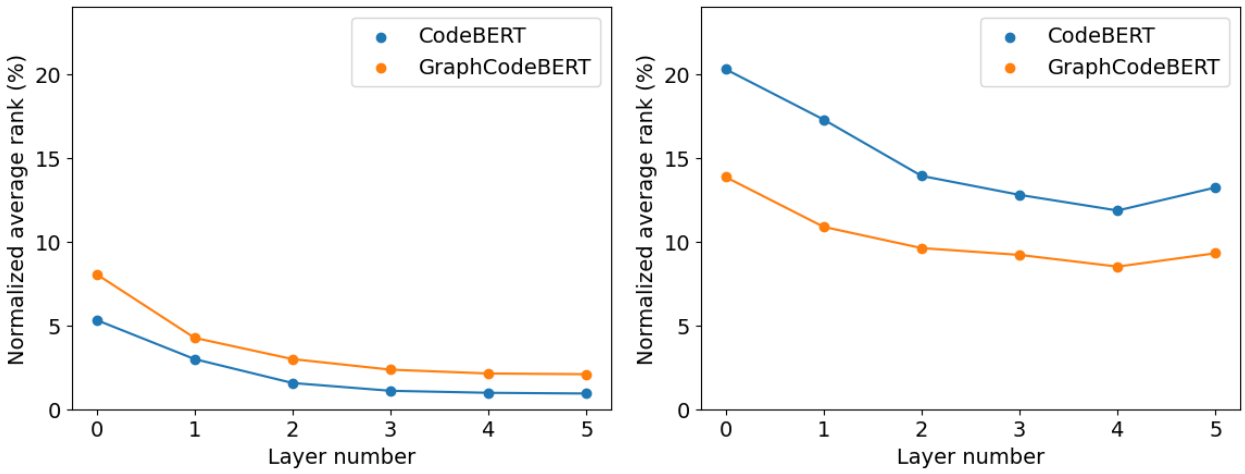
Code document generation, however, has a larger average normalized rank, since the task has different types of input and output. As in Figure 4.7c and Figure 4.7b, The source codes are much longer than the NL outputs so there is a less one-to-one correspondence between output and input tokens. Meaning for each decision, the attention scores will focus on a wider range of tokens rather than a few. The average normalized rank for this task is less than 10% for GraphCodeBERT and less than 13% for CodeBERT, in the last three layers. Further analysis shows that between 20% and 30% of them have a rank equal to or less than 3 depending on task and language.

As a final answer to our RQ1, even though we can not claim causality, we can assert that the attention mechanism's focus on tokens, has a considerable correlation with the model's decision. We only considered the more direct and obvious kind of impact, but the total contribution of the attention mechanism is definitely higher than what is reported here. We also observe that, this direct effect is higher in tasks with the same types of input and output (code-to-code). This justifies the attention mechanism's suitability as an explanation method for the Transformer-based models.

> ***Answer to RQ1:*** *Our findings show that in CT, more than 80% of the tokens that are generated by the model as the output, and also are available in the input, have one of the top-3 attention scores in their corresponding step in last three layers. While the average normalized rank is less than 3% for those layers.*
> *The average normalized rank for CR is also less than 4% for last layers. For CDG it's less than 10% for GraphCodeBERT and less than 13% for CodeBERT which is expected, since CDG has different types of input and output.*
> *Therefore, we conclude that the attention mechanism has a considerable impact on the model's output and hence, it is a suitable XAI method to explain the strengths and weaknesses of our code models under study, in the next RQs.*

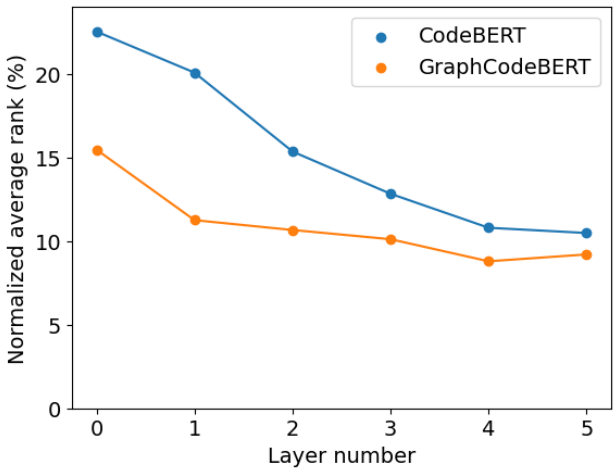
(RQ2) What do the pre-trained code models learn?

Motivation. Applying a Transformer-based model on a source code, for any downstream task, accepts the source code snippets as sequences of tokens. Yet, little is known about what the code models learn from that code snippets. For example, what kinds of code tokens are often highlighted by the model in the learning process and whether such highlighted tokens are semantically important or not. In general, knowing what has been learnt is important from two perspectives: (1) Trust: depending on what tokens are impacting the output one can rely more or less on the recommendations. For example, if XAI reveals that a model

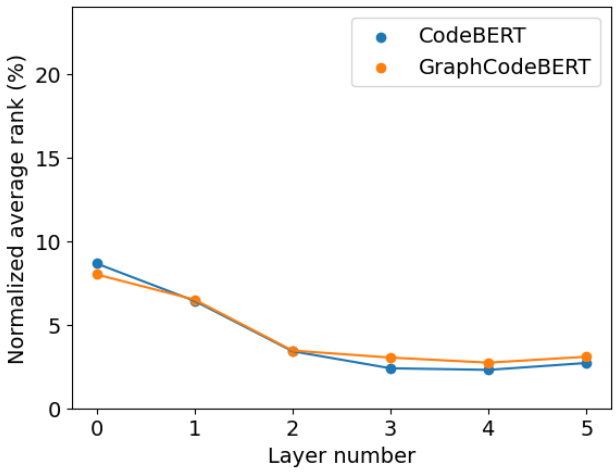


(a) CT

(b) CDG for Java



(c) CDG for Python



(d) CR

Figure 4.7: Average normalized attention rank of the equivalent input token for each output token, per layer, over all samples in all experiments.

makes its decision mainly based on parentheses and indentations in the source code, the user will not (and should not) trust its recommendations, (2) Debugging: knowing which tokens have the highest influence on a generated wrong output, one can devise mechanisms to change the training data, models, or the training process to fix the issue.

Therefore, in this RQ, we study the attention weights of different token types provided by CodeBERT and GraphCodeBERT's internal layers, to see which kind of source code tokens have the highest influence on each generated output, for a given downstream task. This helps us understand whether these models' success and failure are related to focusing on the right/wrong tokens or whether the highlighted tokens are reasonable and the potential problems' root causes are somewhere else (e.g., inappropriate evaluation metrics).

Design. As discussed, in 4.3, to explain the models, we analyze the attention weights per token type and not individual tokens. To do so, we first need to define a list of token types. This is a subjective decision on what tokens types are of interest for our study. We opt for a set of seven token types that cover all tokens and group them according to their semantic relevance, as follows:

Method name: The method under study's name can be one of the main decision factors on perceiving what a method does which is a very important step for the model, specially in some tasks such as CDG. This only includes the main method's name in each sample (reminder that each input sample is the source code for one method) and not the methods called within the main method's body.

Type identifiers: This category represents all the keywords that are used for identifying token types in our source languages Python and Java. For more information on these tokens, look at "type_identifier" and "*_type" node types in tree-sitter for each language.

Language keywords: Control flow command tokens are all bundled together for each language in this category. These tokens can be found in Table 4.3.

Method calls: This category includes all the tokens that are the name of methods invoked within the body of the method under study. These method calls can also be instrumental in describing what the method is doing and may contain bugs to fix.

Local variables: Here we consider all variables that are used only in the body of the method under study. That is, the input arguments of the method are excluded.

Input variables: This category only contains the input arguments of the method under study. We have separated it from the Local variables category.

Others: This category represents all the tokens that are not included in any of the categories above. Mostly tokens like punctuation, constant values, parentheses, etc.

Note that although these token types are chosen subjectively, the results will show that they are among

Table 4.3: Control flow command tokens for Java and Python.

Language	Tokens
Python	False, None, True, and, as, assert, async, await, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield
Java	if, else, switch, case, while, class, enum, interface, annotation, public, protected, private, static, abstract, final, native, synchronized, transient, volatile, strictfp, assert, return, throw, try, catch, finally, default, super, do, for, break, continue, super, void, import, extends, implements, import, instanceof, new, null, package, this, throws

the most important tokens and not many contributing tokens are left for the "Others" category. We should also emphasize that the level of abstraction on what constitutes a "token category" is up to the XAI user. For instance, we decided to separate the Input Arguments category from the Variable Names category, to better analyze their effects individually, but merging the two categories is a valid design choice as well (just in different level of abstraction).

In RQ2, for each sample in test data, we follow these steps to find the distribution of attention scores over different categories:

For each generated token (each step), we take its attention weights toward the input tokens and find their corresponding type. Then, we accumulate the attention scores of all the tokens in each category to get a total score for that category in that sample. Our categories cover all tokens so the sum of all scores for each step is equal to one. We go through the same process for all output tokens in all code snippets of the testing dataset and gather the accumulation of attention scores for each category.

It's noteworthy that the size of token categories is quite imbalanced. For example, there is only one method name for each sample but many tokens in the "others" category. Therefore, we normalize the total score of each category, according to its population as in Table 4.4. This gives us the attention score per token for each category. Finally, we normalized scores of all categories, between 0 and 100 for the purpose of easier comparison.

Among these defined categories, "Method name", "Input variables", and "Local variables" are more representative of the naming aspects of a source code. So we group them in a higher-level category of "Naming", while "Method calls", "Type identifiers", and "Language keywords" are more related to the

Table 4.4: Number of tokens in each category for each task

	Method name	Input variables	Method call	Variable	Type identifier	Language keywords	Others	Total
CT	1,033	2,859	2,065	4,289	2,815	3,408	21,731	38,200
CDG_Java	12,991	44,175	42,285	101,352	63,143	73,644	446,419	784,009
CDG_Python	16,633	110,630	19,567	197,878	0	77,662	487,860	910,230
CR	7,761	19,316	33,717	70,750	52,969	36,408	319,544	540,465

Table 4.5: Normalized attention score of the two high-level categories of tokens, for different code models and tasks. The results are the average of all six layers for each task.

Task	Model	Naming	Structural	Others
Code Translation	CodeXGLUE	42.36%	51.38%	6.27%
	GraphCodeBERT	42.60%	51.30%	6.09%
CDG_java	CodeXGLUE	63.31%	29.19%	7.50%
	GraphCodeBERT	64.51%	28.19%	7.30%
CDG_python	CodeXGLUE	67.97%	23.75%	8.28%
	GraphCodeBERT	74.63%	17.83%	7.54%
Code Refinement	CodeXGLUE	56.46%	37.96%	5.58%
	GraphCodeBERT	55.49%	38.86%	5.65%

structure of the code. Thus, we consider them as the higher-level category of "Structure". Also note that we only report the average score of all six layers for each task and model here, since reporting all results per layer would be too lengthy and also this RQ results were quite similar over different layers.

Results. In the three downstream tasks under study, we expect different normalized attention scores per high-level category, as follows: (a) CT is a task that heavily relies on the structure, since the model must learn the source language structure, and generate the equivalent structure in the target language. (b) In CDG, the structure is less important (nested blocks and syntax trees have less to do with the output document). On the other hand, names are very important in this task, since they basically describe the functionality of the source code. (c) Finally, we expect code refinement to be in the middle of these two ends, since both names and the structure are important in debugging a code.

Table 4.5 shows the normalized total attention score of each high-level category and validates our hypothesis. Code Document Generation, as a task that heavily relies on naming, has a considerably high normalized attention score of over 63% for the Naming category, while it pays much less attention to the Structural token types category, compared to other tasks. It has also a higher number for the Other category which can be understandable considering the fact that NL comments in the code are also part of this category. Code Translation, on the other hand, is the only task that has more than 50% normalized attention score

for Structural tokens and less than any task for the Naming category. Code Refinement in this comparison holds the middle ground between the two mentioned tasks in both categories.

In Table 4.6, we have a more detailed analysis for each of our categories. We saw that both models pay more attention to the structural tokens for CT. Getting into more details, the results show that this attention is more focused on Method calls, and Type identifiers rather than Language keywords, roughly having only 6% of the normalized score. It is very interesting that studying individually, the Method name category has the second highest score after Method calls. Even though in translating a code, the method name is usually unchanged, this shows that **the models considerably utilize the name of the method to understand its functionality.**

In the CDG task, we observed a great reliance on Naming categories, the results show that the Method name category plays the most significant role. It always has a normalized score of close to 40% or higher. In the absence of Type identifiers, in GraphCodeBERT CDG_python, this category has the highest score ever among all the token types across all tasks/models. Intuitively, this amount of importance is justifiable, given that even humans rely a lot on the method names to understand their functionality. In addition, in three out of four different experiments for this task, the Input variables have the second highest normalized attention score (between 11% to 16%). This basically means that **the models have learned that while generating document for a method, the main and most interesting part is the signature of the method and not the body.**

Code Refinement which holds a middle ground between two other tasks has a very close score for four categories of Method name, Input variable, Method call, and Variable. Since in this task the model is supposed to look for bugs and try to fix them, it seems that **the models have learned that fewer bugs happen in categories like Type identifier, Language keywords, and Others.** This makes sense because we know that databases for this task are gathered from public projects and they probably don't have syntactical errors. The method name is also less likely to have a bug, but as we saw in other tasks, this category always has a minimum appeal for the models.

It is interesting that according to the table, Type identifiers which is a category purely related to the syntax of the code and don't include any naming, have the most contribution to CT with a gap compared to others. It has a score of 20.67% and 21.66% for CodeBERT and GraphCodeBERT respectively while its score in other categories is below 14%. Also, it is worth mentioning that GraphCodeBERT which uses the dataflow of the code in order to capture its structure; always has a higher score for this category compared to CodeBERT.

The same patterns appear to be valid for Variable names and for CR. The score of this category for CR is 17.85% and 18.93%, while in other tasks the score is always below 12%.

Table 4.6: Normalized attention score of different token types, for different code models and tasks. The results are the average of all six layers for each task.

Task	Model	Method name	Input variables	Method call	Variable	Type identifier	Language keywords	Others
CT	Code XGLUE	21.36%	9.63%	24.26%	11.36%	20.67%	6.45%	6.27%
	Graph CodeBERT	22.78%	7.89%	23.46%	11.93%	21.66%	6.18%	6.09%
CDG_java	Code XGLUE	39.44%	13.88%	10.49%	10.00%	13.07%	5.63%	7.50%
	Graph CodeBERT	41.04%	15.10%	8.44%	8.38%	13.13%	6.62%	7.30%
CDG_python	Code XGLUE	46.17%	11.96%	16.00%	9.83%	0.00%	7.76%	8.28%
	Graph CodeBERT	54.21%	12.79%	10.79%	7.64%	0.00%	7.03%	7.54%
CR	Code XGLUE	22.01%	16.60%	20.33%	17.85%	9.82%	7.81%	5.58%
	Graph CodeBERT	19.36%	17.19%	21.15%	18.93%	10.02%	7.69%	5.65%

Similarly, the Method name category has a harshly higher score in CDG. For this task in python, this category has a score of 39.44% and 41.04%, and in Java, it has 46.17% and 54.21% for CodeBERT and GraphCodeBERT, respectively.

Answer to RQ2: *Having all these observations, we can see a pattern of importance comparing different tasks together. The method name and input variable (basically the first line of the code samples) are the most important categories for CDG; Method calls and local variables play the most significant role on code refinement, alongside the method name with slightly lower importance. On the other hand, code translation is concerned with type identifiers and language keywords, more than any other task, while still caring about some naming categories, as well. In Table 4.5, we have aggregated the numbers for our two main categories and we can see a pattern that we expected. Naming tokens are important for all tasks, but less for code translation which alternatively, cares more about structural tokens compared to other tasks.*

(RQ3) When do the pre-trained code models not work?

Motivation. In RQ1, we made sure attention is a reasonable tool to explain CodeBERT and GraphCodeBERT, and in RQ2 we show that the tokens are mostly picked up correctly by the model, so the models learn the right tokens per task, in most cases. In this RQ, we delve deeper and explore the scenarios where the

code models did not perform well but the problem at hand was not that difficult. Answering this question will help understand what should be done to make the model at least perform consistently well, on simple data items.

Design . To provide explanations on when CodeBERT and GraphCodeBert perform well and when they fail, in this RQ, we start by a qualitative analysis of some sample predictions. Then we make hypotheses based on our observations and finally verify them quantitatively on the whole dataset. To define the strong and weak performances of the models, we cannot simply rely on the absolute values of the evaluation metric (BLEU). Since the magnitude of the BLEU score partially depends on how difficult or easy the document generation task is, per sample code. Therefore, we need to somehow measure the difficulty level of the document generation task, given a source code.

To achieve our purpose in this RQ, firstly, we need to define a few metrics that can represent the easiness/difficulty of a specific sample, and secondly, evaluate the model's performance on that sample.

As we said, in CR and CT, expectedly, the model is busy copying and inserting the same tokens from the input to output and doesn't usually have to generate new tokens. Having this fact, we expect the Levenshtein Distance (LD) of the input, and their corresponding expected true output (let's call it "gold output" from now on) as a suitable metric for the difficulty-level of that sample. So in CR and CT, for each task, we calculate the Levenshtein Distances for all samples in the dataset, and consider the first one-third samples with the least LDs as the easier targets.

For the CDG task, we keep the same general approach (that is considering the similarity of input and gold output), but since in this scenario, the output is in NL and the input is in PL, we define slightly different steps.

To define the difficulty-level for the CDG task, we use the intersection between the set of preprocessed tokens of the gold output document, per method, and the set of preprocessed tokens of the method's source code.

To find the overlap between source code and output tokens, we follow these preprocessing steps: First, we remove punctuations and tokens shorter than three characters, in the output document. Then, lemmatize those tokens using the standard Wordnet [121] lemmatizer, offered in the NLTK package. Next, we tokenize the source code using a parser for the respective language. Note that as explained in Section 4.2.3, due to CodeBERT and GraphCodeBERT's tokenizations, which may split one meaningful word into multiple tokens, we don't use their tokenization for this analysis. Finally, we create a set of case-insensitive tokens that fall at the intersection of processed output and source code tokens.

Now an easy/difficult document generation task is when the overlap between the two sets is high/low. Therefore, the same as the cut-offs for CR and CT, we consider the first one-third samples with the highest

overlaps as easy, and the one-third cases with the least overlaps as hard problems, and ignore the rest (average difficulty-level).

The above process tells us which samples are considered hard and which ones are considered easy. Now we need to measure the performance of models. To do so, we use BLEU score since it is the most accepted and reliable score that is applied in these tasks in the literature. For the accuracy, we also take the one-third of the samples with the highest BLEU scores as $High$ and the samples with the one-third least BLEU scores as Low.

Having these definitions, there will be four categories (for the tuples of <easiness level, model accuracy>) as below:

$Easy - High$**:** This category contains test data items that are easy problems (meaning high similarity between the input and the gold data) that the models have achieved a High BLEU score on them.

$Hard - High$**:** This category contains the samples labeled as $Difficult$ and $High$. This means even with the lack of common tokens, the model was able to achieve a satisfactory result in these cases.

$Hard - Low$**:** This category includes the cases that are $difficult$ again, and expectedly, end up with Low accuracy for the model's prediction.

$Easy - Low$**:** This category is the most interesting one in this thesis, since it can show the potential weaknesses of the model and is very suitable for being analyzed and "explained". Samples in this group, are among the samples with higher overlaps in their corresponding dataset that means that the model is having rather an easy job predicting. However, the BLEU score as our indicator of the model's accuracy is showing poor performance comparing to other samples.

In order to perform our manual observation (qualitative study) for this RQ, after grouping our test dataset into these four categories, we randomly choose 100 samples from our target category ($Easy - Low$), and manually analyze their outputs and attention weights.

For each sample, we record the most interesting findings to identify the most frequent patterns. This way we develop some hypotheses. Finally, we try to verify these hypotheses by quantitatively studying the whole test dataset, with respect to the hypotheses. The output of this quantitative phase is in the form of some descriptive statistics to either confirm or reject the observations made based on the 100 samples.

Results. Having the data divided according to the defined groups, Table 4.7 shows the ratio of the target category population to the whole dataset for each task-model and also, Figure 4.11 and Figure 4.12 show the distribution of the target category. Next, we will explain the observations from the manual analysis.

Observations 1: The pre-trained code models don't work well, when the output gold document is long.

Our first observation regarding the CDG task is that in cases with long gold documents the BLEU scores

Table 4.7: The ratio of the Easy-Low category population to the whole dataset, for each task-model.

	CodeBERT	GraphCodeBERT
CT	11.70%	11.41%
CDG / Java	5.12%	5.59%
CDG / Python	5.87%	5.85%
CR	2.49%	4.37%

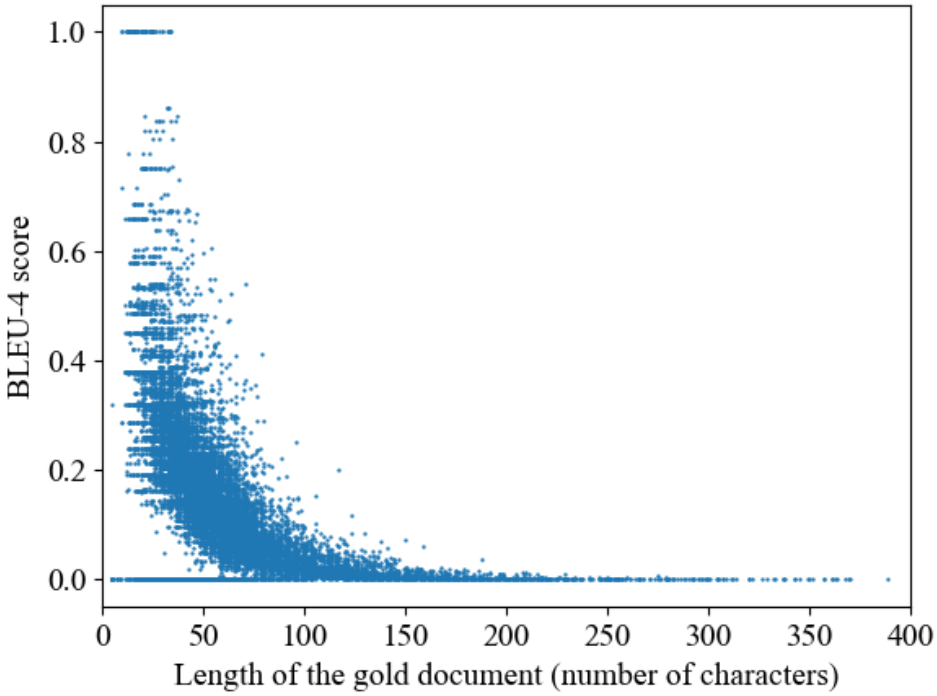
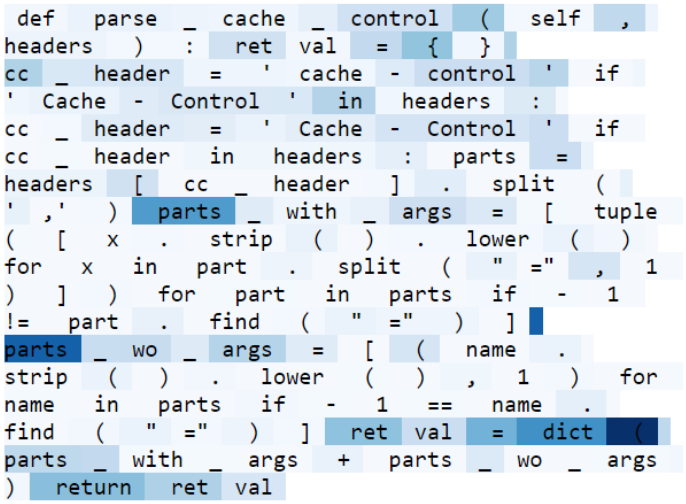


Figure 4.8: Distribution of BLEU scores according to the gold document code length.

tend to be low! We plotted the distribution of BLEU scores, according to the gold document's length, in Figure 4.8. According to the plot, most high BLEU scores happen when the length of the gold document is less than 50 characters.

In general, the idea of the BLEU score is about counting the number of n-grams that are common between the reference and the output. Model-generated documents are usually short so for longer sentences, there is less chance that the model chooses the same phrasing and words with the same order. Another plausible explanation for this observation is that a longer document means the method implements a more complex task and thus it is harder for the model to generate the right documentation for the complex method.

One potential solution for this problem is forcing the model to generate longer sequences as the output which will increase the chance of a high BLEU score, in cases with long reference documents. But obviously,



Gold document: Parse the cache control headers returning a dictionary with values for the different directives .
Best prediction: Parse a dict of headers
BLEU score: 0.08
Overlap: 0.54

Figure 4.9: A sample method, broken into tokens, the gold document, the best prediction by CodeBERT, the BLEU score, and the amount of overlap between the method's code and the gold document. The attention values of the last layer of CodeBERT executed on this code are also shown, as shades of blue (the darker, the higher). We used the attention scores of the last generated token("headers" in this example), in the last layer, for the visualization.

since this is not actually the model's fault and in these cases, a low BLEU score does not necessarily indicate a bad prediction (like the example shown in Figure 4.9), the best way to handle this problem is considering other evaluation metrics and ideally more subjective ones.

Observation 2: The pre-trained code models don't work well, when the input source code is complex.

Another interesting observation is about the length of the source code. The results show that in cases with longer code, the BLEU score is usually lower. We started the initial analysis with the CDG data and the results, which are summarized in Figure 4.10, show a decreasing trend of BLEU scores, by the increase of the source code length. For example, the average BLEU score for cases shorter and longer than 300 tokens is 0.161 and 0.149, respectively.

There are two explanations for this observation. The first one is the fixed length of the inputs in models, which basically means if the source code's length is higher than a fixed value (in our experiments, 256 tokens), then the input will be truncated. This means some tokens will not make it to the decoder, and thus we have a potential degradation in the final scores.

Another reason is the increased complexity of the code. Similar to Observation 1, longer source code

means more complex logic, more objects, and functionalities to consider for the model that probably leads to a poorer results.

According to these reasons, two basic solutions can be suggested: (a) increasing the input threshold and (b) decreasing the input's length. Input threshold can be easily modified in the training process of the model and only requires more resources. The second option, however, is a more complex solution that is already kind of naively implemented by the truncation. Another potential alternative is to refactor long methods to multiple smaller methods, then pass each method to the models to generate documents for, and finally merge all output documents as one document.

Next, to expand this observation on all tasks-models, and find more statistical results, we used some common code complexity metrics and conducted an analysis on all 8 model-tasks to get a better understanding of the root cause of the poor results for long code snippets. We chose 'number of tokens', 'cyclomatic complexity', 'nested block depth', 'number of variables' as complexity metrics. To measure the difficulty of the task, we also included the same 'Levenshtein distance' for CT and CR and 'overlap' for CDG, in our analysis.

For each task-model, we have five different metrics to study, so we have one plot per metric. In each plot, the distribution of samples in the respective dataset, regarding that metric is shown with blue bars and the same distribution but only for the target category (Easy-Low) is shown by red. With this visualization, we can identify any difference in terms of trends on a specific metric in the target category vs. the whole dataset.

Figure 4.13 and Figure 4.14 show the results for code refinement (CR). As illustrated in the plots, the Easy-Low category has a very similar distribution to the whole dataset, except a slight increasing trend for some reported metrics like number of tokens and number of variables This means that the model tends to make bad decisions, whenever the source code gets more complex in terms of number of tokens and variables, even if the overlap of tokens is high. For instance, considering the number of tokens, as a measure of code complexity, the proportion of samples with more than 100 tokens is mostly higher in the Easy-Low category than the share of the same samples in total. It means that the samples with many tokens, are more likely to be assumed "Easy" in our categories (more overlaps between input and outputs) but in fact they are harder for the model to understand (given the long length of the code snippet).

Figure 4.15 and Figure 4.16 show the same results for code translation (CT). For this task, number of variables follow the same pattern as the CR task, that is Easy-Low category is harder based on those metrics. On the other hand, considering the number of tokens, nested block depth, and even cyclomatic complexity, there is a reversed connection. In other words, samples that have smaller values of these metrics, have higher density in Easy-Low category. For instance, in both models, samples with tokens less than 20, are around

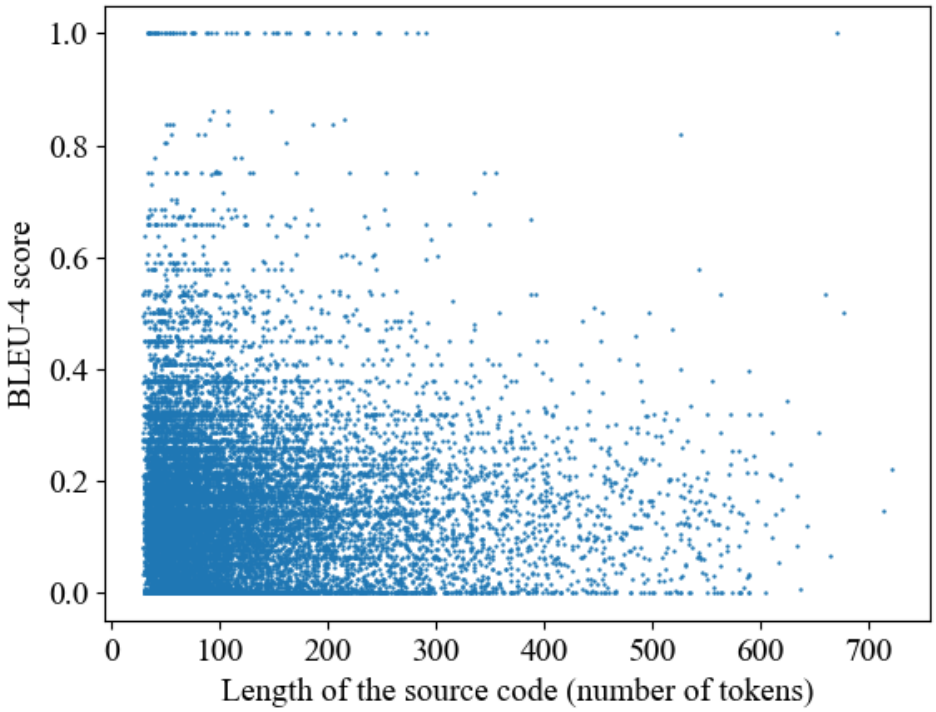


Figure 4.10: Distribution of BLEU scores according to the source code length.

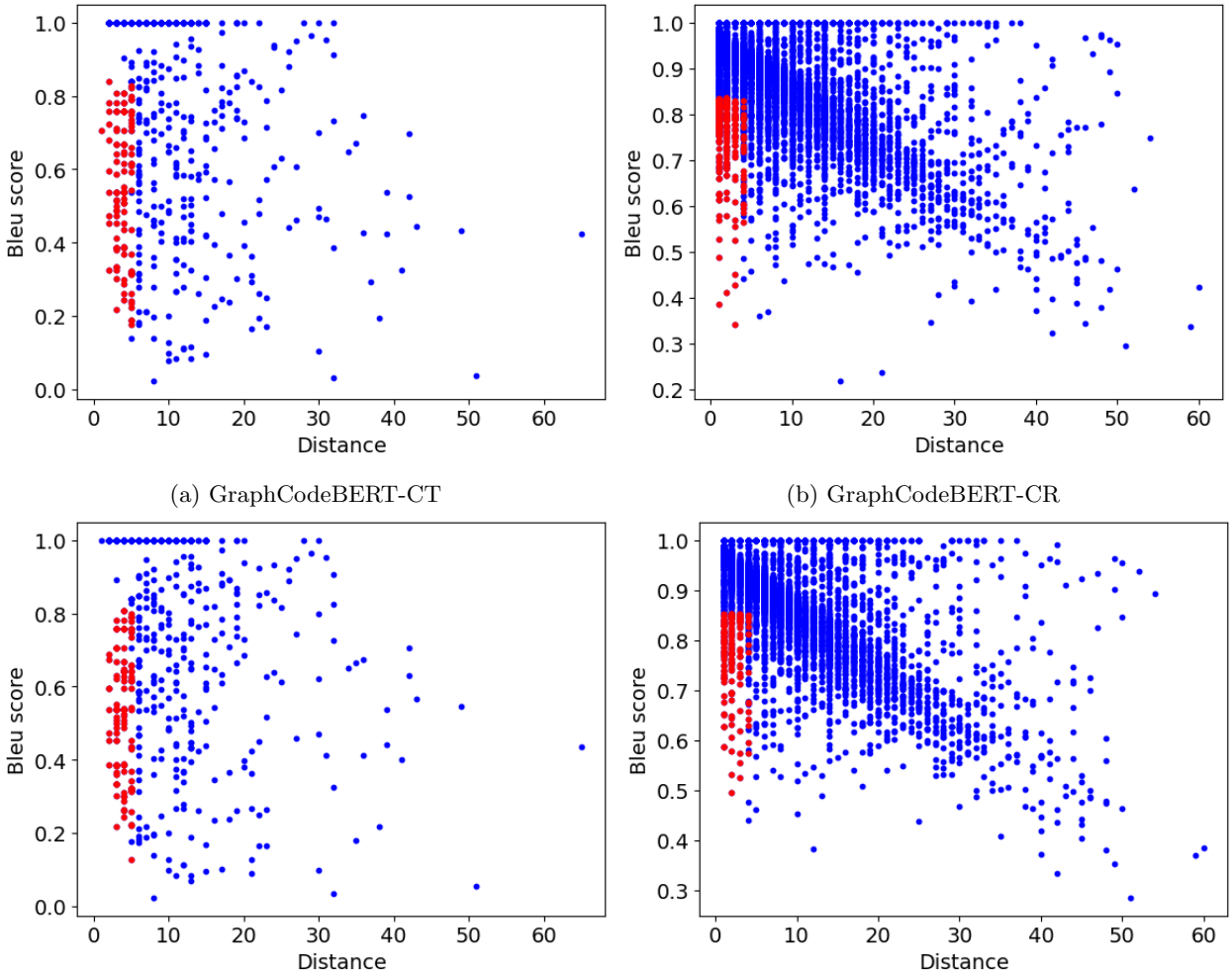
50% of the target category population, even though they occupy a very small portion of the total dataset. One plausible explanation is that when the source code is too short (very small number of tokens and very few nested blocks), the model fails to translate it properly, due to the lack of enough information/context. Another explanation is the fact that it's harder to maintain a high BLEU score when the input is very short.

Figure 4.17,Figure 4.18,Figure 4.19, and Figure 4.20, illustrate the same results this time for code document generation (CDG). In this downstream task, we can see patterns more dependent on the language, rather than the model. In all cases, the number of variables, number of tokens, and nested block depth follow the same general pattern.

In these experiments, we can see that both models struggle with samples with less complexity in Java and on the other hand, have problems figuring out the more complex samples in Python. For instance in Python, samples with more than 80 tokens or 8 variables, have always a higher density in the Easy-Low category compared to all of the dataset. Meanwhile, it's interesting that considering the cyclomatic complexity, both models in both languages struggle with samples with higher complexity.

So all-in-all, one can conclude that code models perform poorly on the code snippets with extreme values of complexity-related metrics on either direction (i.e., both long code with many nested blocks and tokens and also very short code with only a few variables and tokens are hard for the models).

Observation 3: The pre-trained code models don't work well, when the models fail to focus



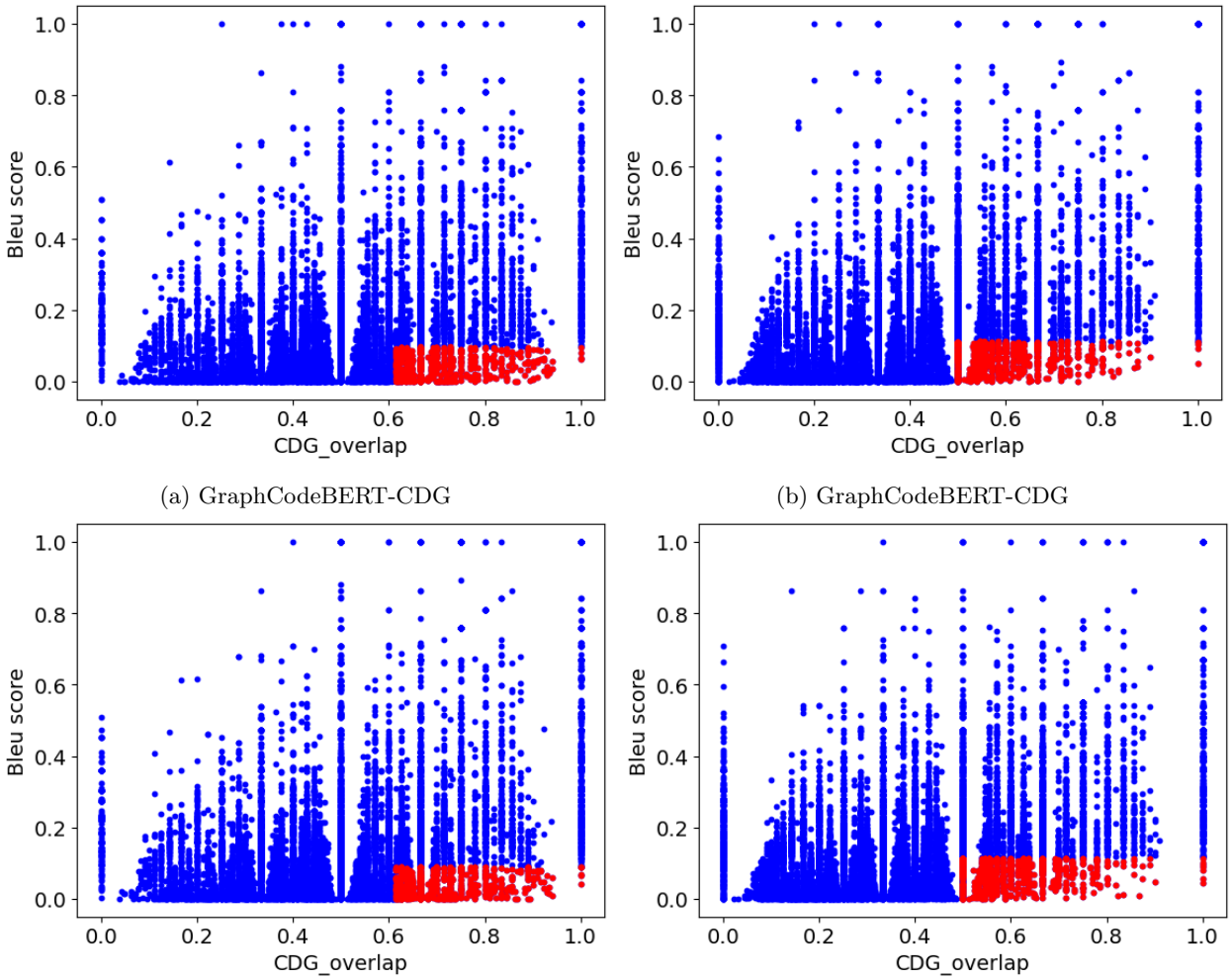
(a) GraphCodeBERT-CT

(b) GraphCodeBERT-CR

(c) CodeBERT-CT

(d) CodeBERT-CR

Figure 4.11: Distribution of the the whole dataset (in blue) and the Easy-Low category (in red), based on the BLEU score and the Levenshtein distance of the input and gold output for CT and CR



(a) GraphCodeBERT-CDG

(b) GraphCodeBERT-CDG

(c) CodeBERT-CDG

(d) CodeBERT-CDG

Figure 4.12: Distribution of the the whole dataset (in blue) and the Easy-Low category (in red), based on the BLEU score and the Levenshtein distance of the input and gold output for CDG

Table 4.8: Normalized attention score of the Easy-Low samples, in three general categories of tokens, for different code models and tasks. The results are the average of all six layers for each task.

Task	Model	Naming	Structural	Others
CT	CodeXGLUE	48.85%	44.51%	7.70%
	GraphCodeBERT	49.51%	43.49%	7.57%
CDG_java	CodeXGLUE	60.08%	33.28%	8.89%
	GraphCodeBERT	63.85%	37.06%	7.43%
CDG_python	CodeXGLUE	66.37%	29.14%	8.61%
	GraphCodeBERT	73.68%	24.16%	7.87%
CR	CodeXGLUE	56.64%	46.29%	5.63%
	GraphCodeBERT	55.82%	48.35%	5.62%

Table 4.9: The difference of normalized attention scores for the Easy-Low samples and all samples, in two general categories of tokens, for different code models and tasks. The results are the average of all six layers for each task.

Task	Model	Naming	Structural	Others
CT	CodeXGLUE	6.49%	-7.92%	1.43%
	GraphCodeBERT	6.90%	-8.38%	1.48%
CDG_java	CodeXGLUE	-3.24%	1.85%	1.39%
	GraphCodeBERT	-0.66%	0.53%	0.13%
CDG_python	CodeXGLUE	-1.59%	1.27%	0.33%
	GraphCodeBERT	-0.96%	0.63%	0.33%
CR	CodeXGLUE	0.18%	-0.23%	0.06%
	GraphCodeBERT	0.33%	-0.30%	-0.03%

on important categories:

Finally, we analyzed the contribution of token categories similar to what we had in RQ2, but specifically for the target category of $(Easy - Low)$. In Table 4.8 we have the normalized score of two main categories for the target samples. We were interested to compare the results for this category and the previous results for the whole test dataset. Hence, we calculated the difference between these two from Table 4.8 and Table4.5, and the outcomes are shown in Table 4.9. The negative numbers in this table indicate a decrease in the target category.

As the results show, there is a considerable decrease in the scores for the Structural category in CT. While answering to RQ2, we showed that this task mostly relied on this group of tokens 4.3. Likewise, we have a slighter decrease in the score of the Naming category in CDG while the naming category also proved to be the more important category for CDG.

Both of these clues, lead us to the conclusion that the results are less satisfying, whenever the model fails to pay enough attention to the corresponding important token category for a specific task. Based on this observation, potential research questions to investigate in the future are: "Will the model work better if we help it by tagging the token types? Can manually amplifying the attention scores of specific categories according to the task beneficial for the code models".

4.4 Limitations

One limitation of this study is that our experimnts only cover Java datasets in CT and CR and for CDG, it's only Java and Python. Even though our experiments and conclusions are not bound to anything specific to one language; since we needed syntactic information on tokens (and for this purpose, parsing the test cases was required), we couldn't run our experiments on the whole CodeSearchNet dataset for CDG. Other datasets for two other tasks also needed lots of pre-processing to become consistent with our designs and requirements. We plan to extend these analyses to other languages as well in the future.

Another limitation is we used the BLEU score as our evaluation metric for the accuracy of the model, which is commonly used in document generation downstream tasks to reduce the subjectivity of the results. However, as we mentioned in the study, it is not a comprehensive metric as it is unable to find rephrasings or cases that the prediction is not wrong, but doesn't exactly match with the gold label.

The observations we made are also limited to the main patterns we have observed in the 100 samples, manually. Although we later quantitatively validate them, it is of course possible that there exist some other explanations as well that we have missed observing, due to the samples we have chosen.

We have also subjectively selected a set of six token types that we believed could have the highest impact on our models. Although, the RQ2 results showed that these categories actually contributed the most out of all tokens; the level of abstraction (grouping of tokens into categories) is a matter of design choice, which may reveal other findings (at different levels). In other words, although our observations in this level of abstraction are correct, they are not the only way to look at the tokens' attention scores and other categorizations can be studied in the future.

Finally, the study is only limited to three downstream tasks (CT, CDG, and CR) and two code models (CodeBERT and GraphCodeBERT). More work is required to generalize the findings for other Transformer-based models, in the future.

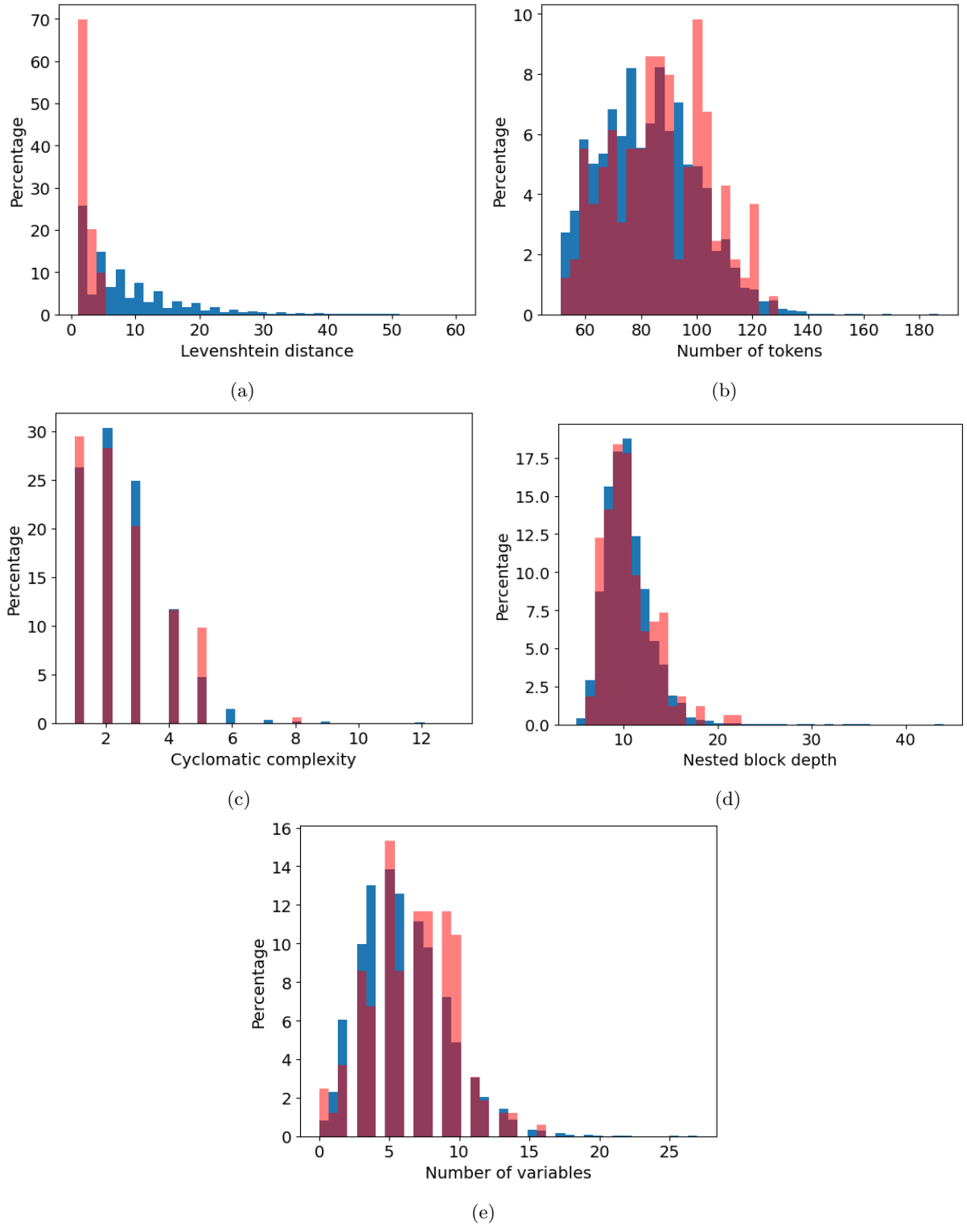


Figure 4.13: Distribution of the the whole dataset (in blue) and the Easy-Low category (in red), according to code complexity metrics, for code refinement on CodeBERT

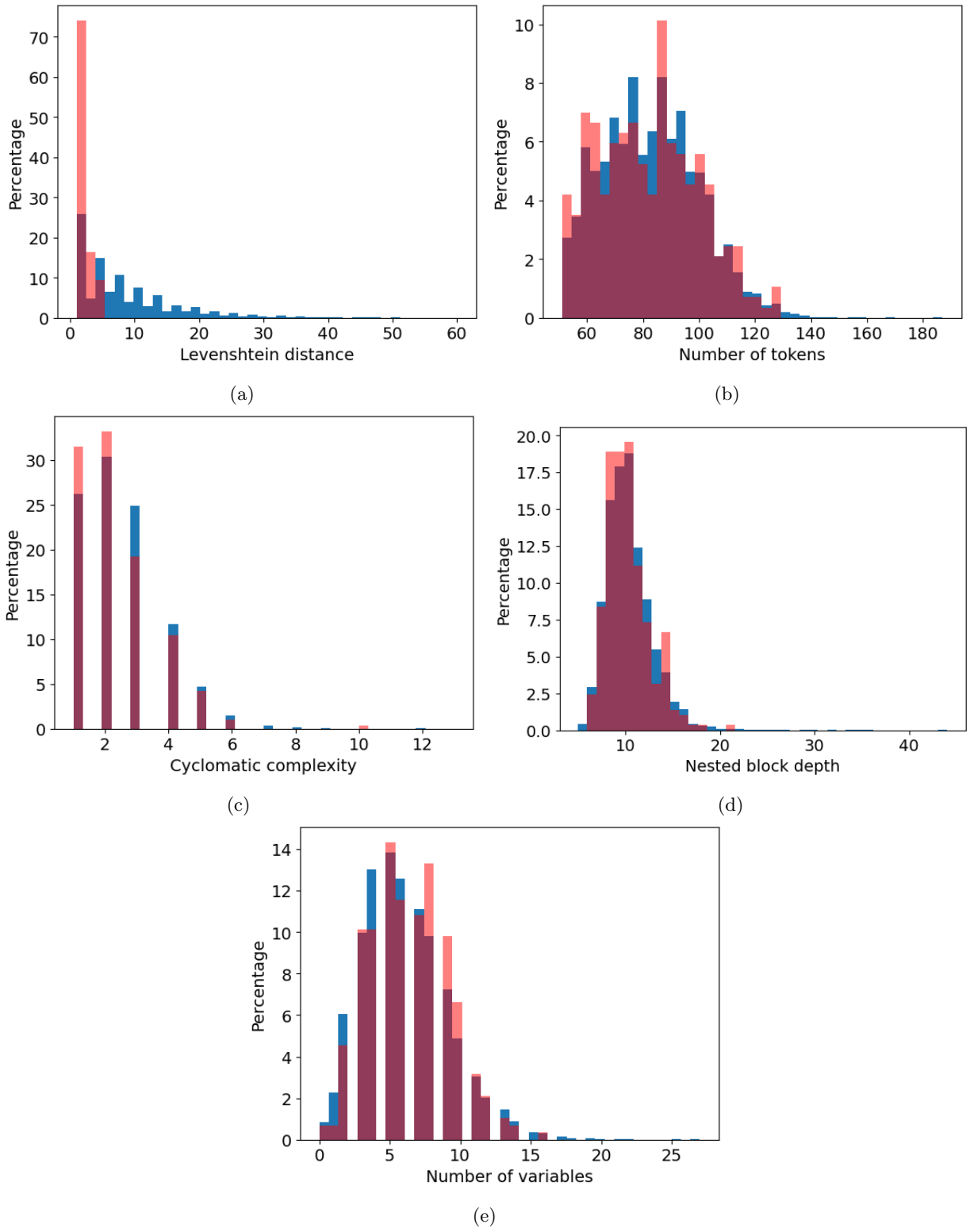


Figure 4.14: Distribution of the the whole dataset (in blue) and the Easy-Low category (in red), according to code complexity metrics, for code refinement on GraphCodeBERT

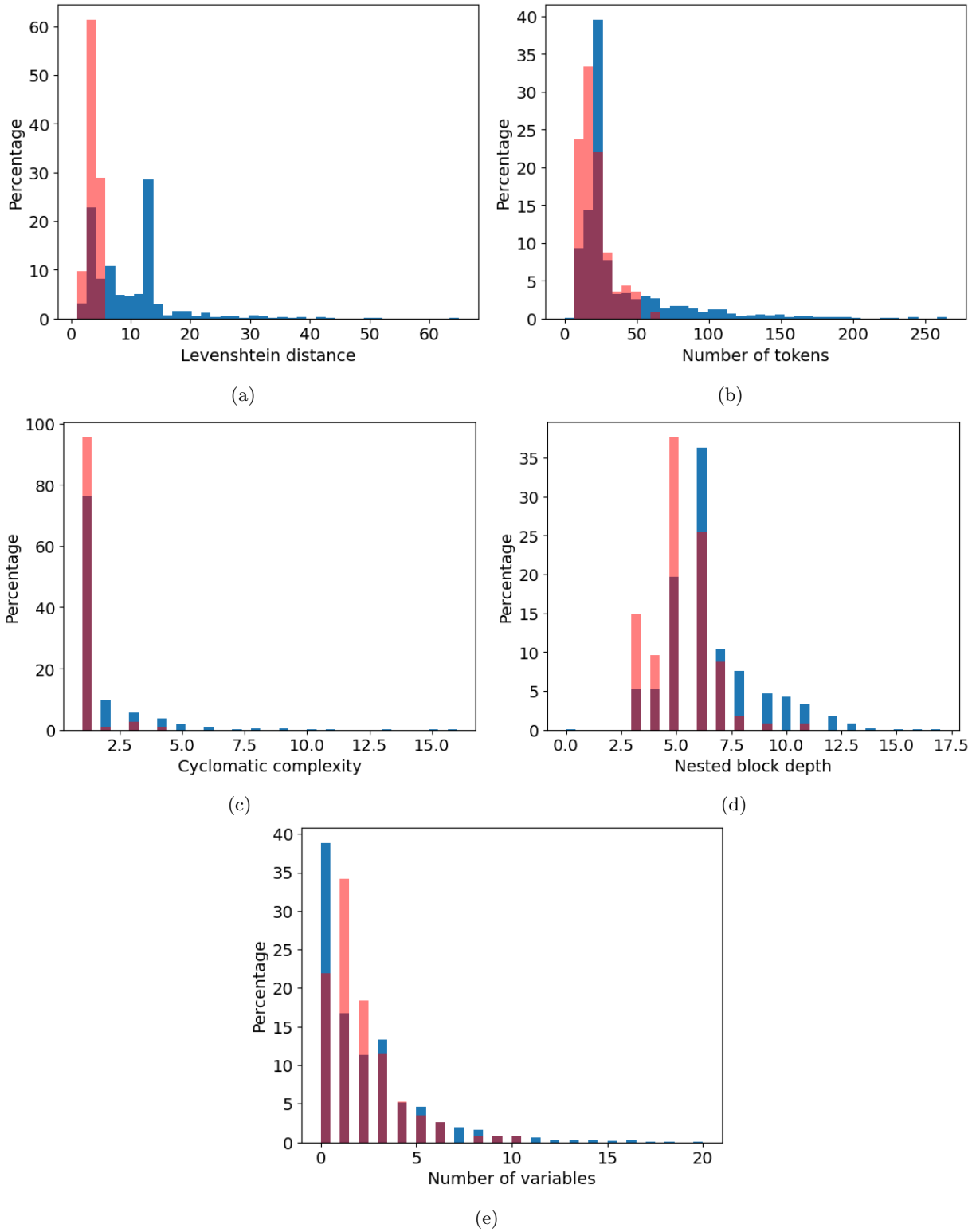


Figure 4.15: Distribution of the the whole dataset (in blue) and the Easy-Low category (in red), according to code complexity metrics, for code translation on GraphCodeBERT

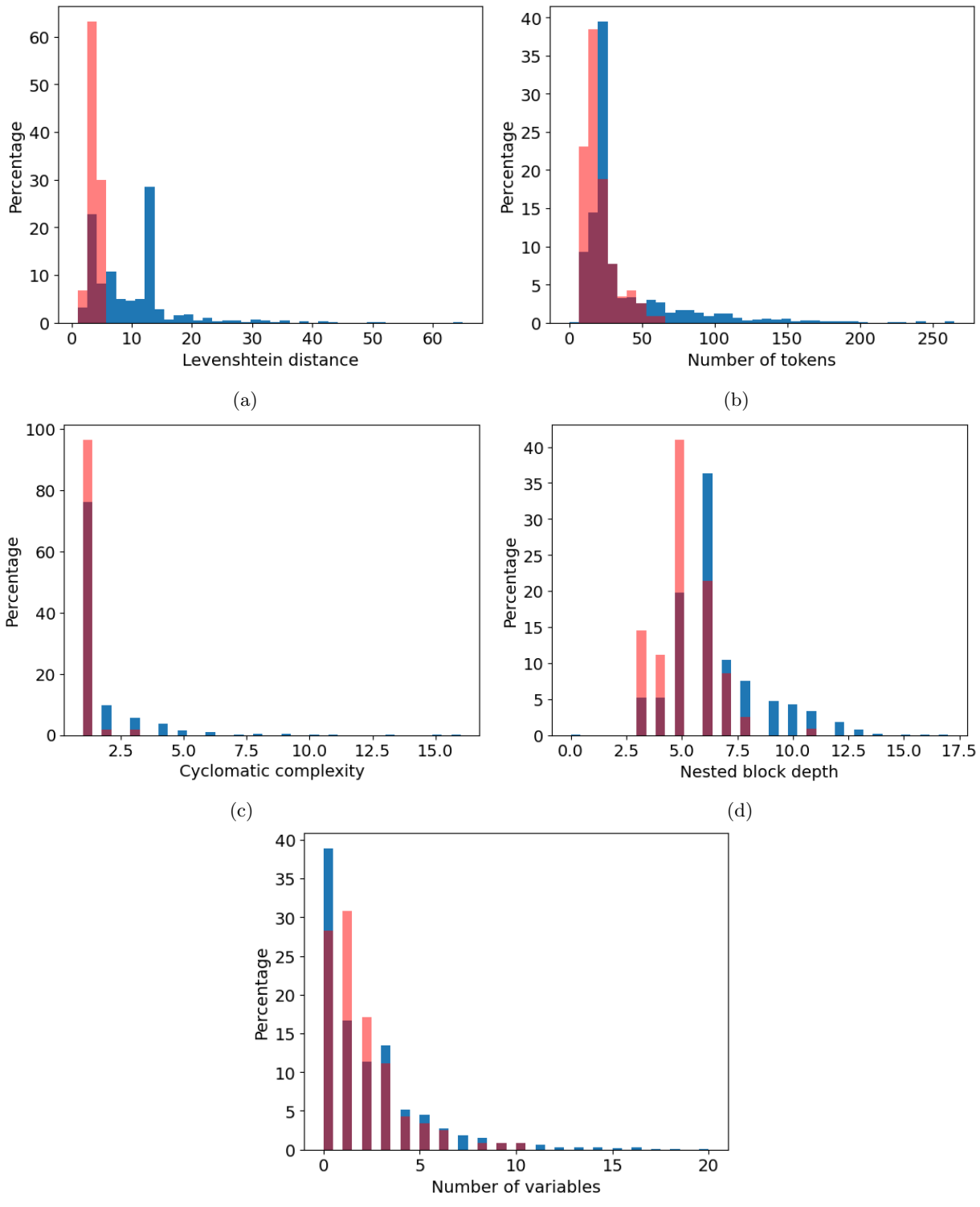


Figure 4.16: Distribution of the the whole dataset (in blue) and the Easy-Low category (in red), according to code complexity metrics, for code translation on CodeBERT

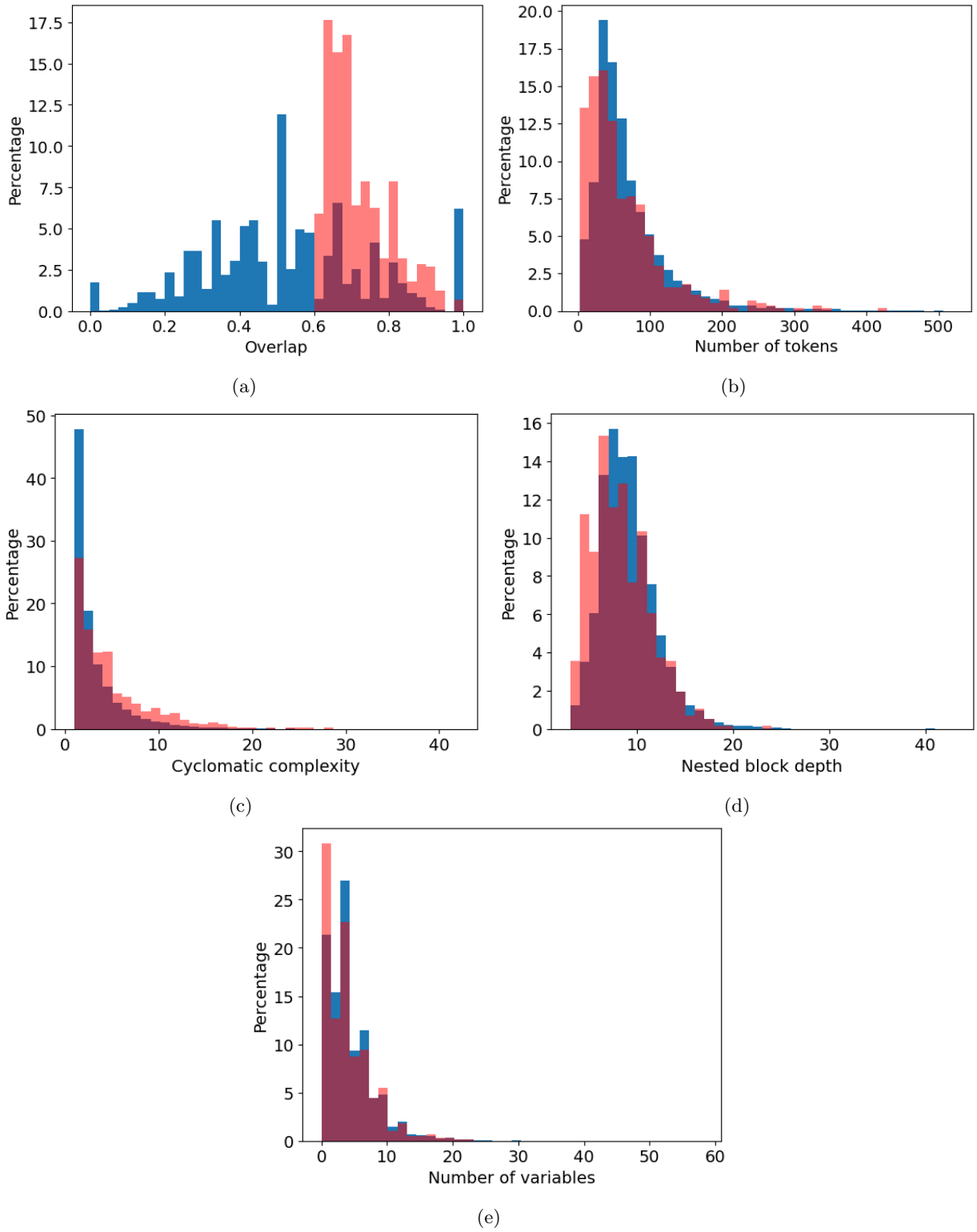
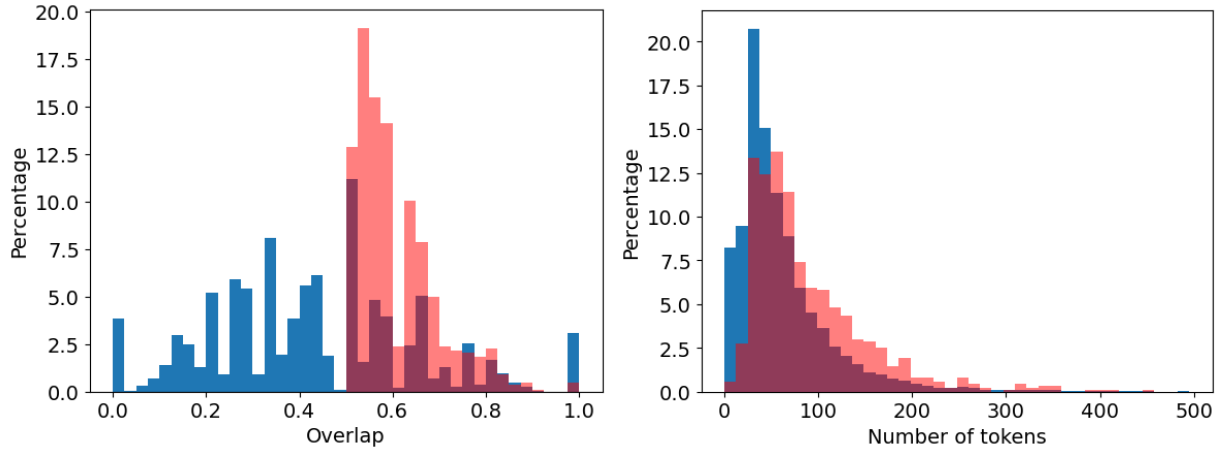
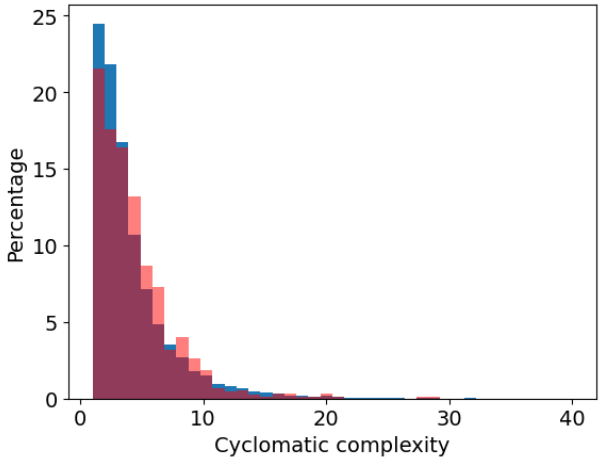


Figure 4.17: Distribution of the the whole dataset (in blue) and the Easy-Low category (in red), according to code complexity metrics, for code document generation on CodeBERT on Java

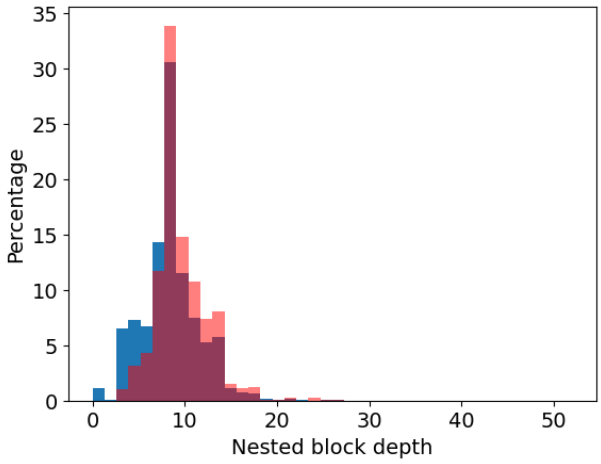


(a)

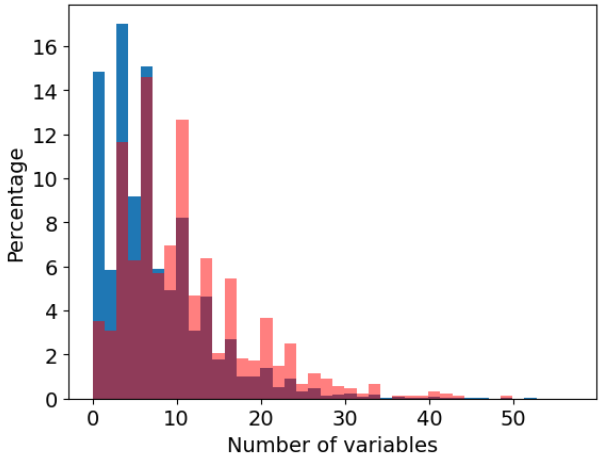
(b)



(c)



(d)



(e)

Figure 4.18: Distribution of the the whole dataset (in blue) and the Easy-Low category (in red), according to code complexity metrics, for code document generation on CodeBERT on Python

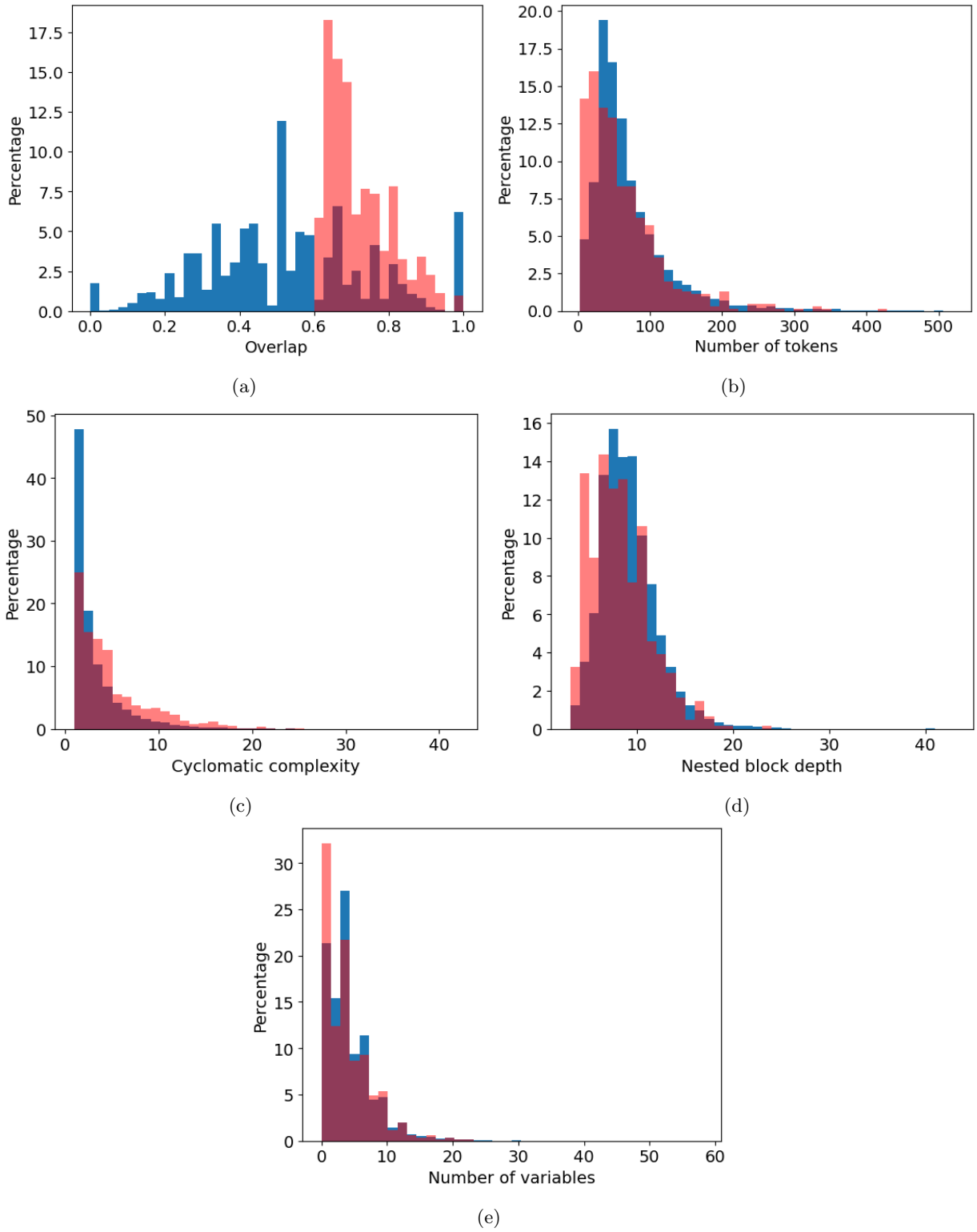


Figure 4.19: Distribution of the the whole dataset (in blue) and the Easy-Low category (in red), according to code complexity metrics, for code document generation on GraphCodeBERT on Java

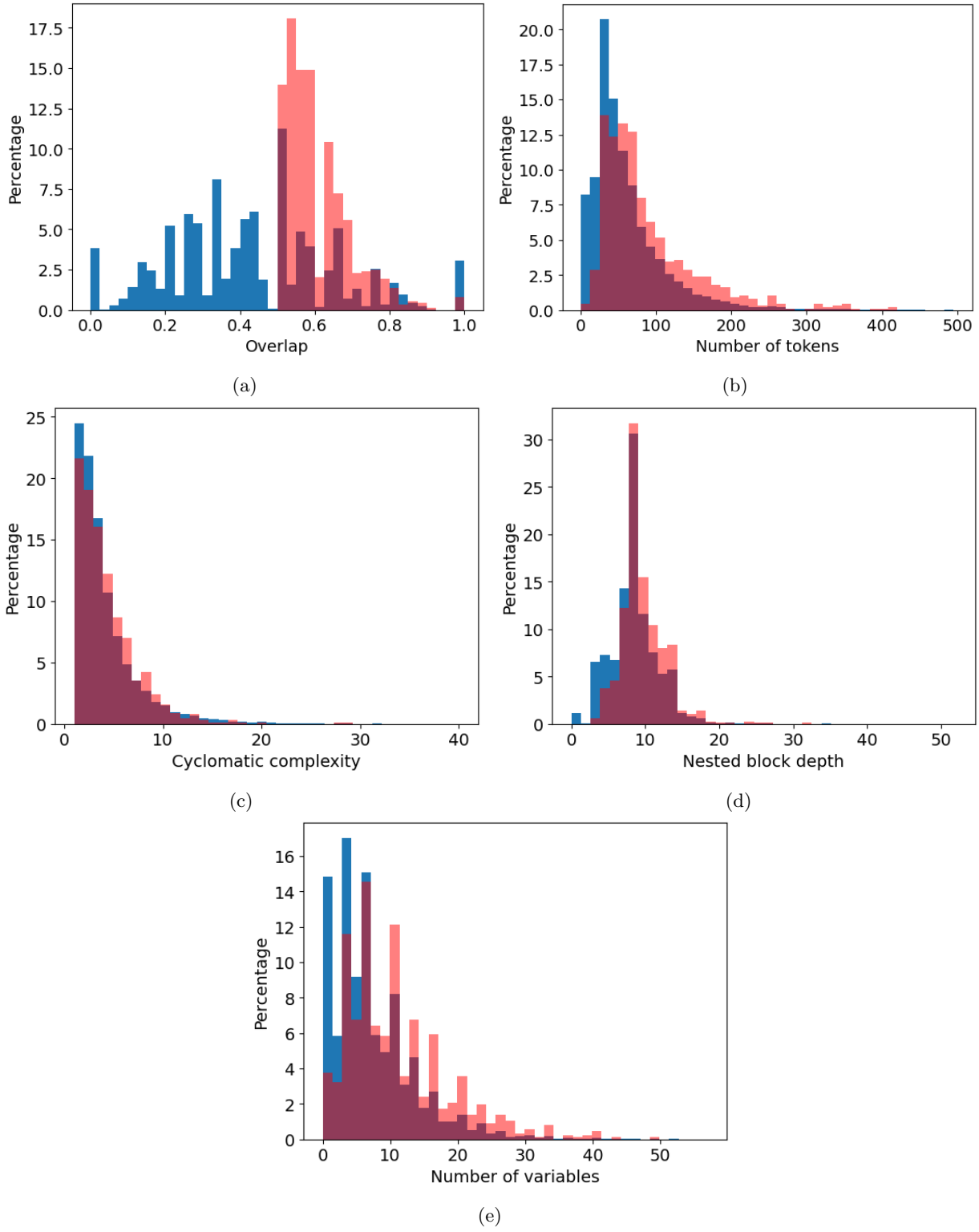


Figure 4.20: Distribution of the the whole dataset (in blue) and the Easy-Low category (in red), according to code complexity metrics, for code document generation on GraphCodeBERT on Python

Chapter 5

Conclusion and future work

5.1 Conclusion

In this thesis, we conducted a systematic literature review on XAI4SE by reviewing 24 relevant papers (out of over 800 automatically selected ones) and analyzing the collected data, we report the current state of research in this area and identify challenges and road ahead for future research.

In particular, the review shows that among different stages of SDLC, QA and maintenance have been the most popular among the XAI researchers, and this popularity is hugely focused on the defect prediction task. On the other hand, generation-based tasks (e.g. code generation, test case generation, and program repair) which are popular in the ML4SE community are rarely studied in the XAI4SE literature. We also observe that self-explainable ML models, such as classic random forest, decision tree, regression models, and naive Bayes are among the most used XAI techniques, and post-hoc techniques are less used in the community, so far. In terms of the type of explanations, XAI4SE has been mainly focused on ML developers as the end user and provides low-level debugging type explanations such as "feature summary statistic" and "model internals", and less on higher-level natural language-based explanations or visualization. Finally, a lack of standard ways for evaluating XAI methods was clear among the studies, which has led to many different project-specific evaluation metrics, as well as human subject assessments.

We partially fill the mentioned gaps about generation-based tasks and more complex models, by conducting an empirical study. For this purpose, we used the attention mechanism as an internal feature to explain specific Transformer-based code models (CodeBERT and GraphCodeBERT) on three generation-based tasks (i.e code document generation, code translation, and code refinement). As a result, we found that code models learn specific patterns, of putting attention to different token types, depending on the task.

While in CDG, they have more focus on naming tokens (e.g. variable names, and method names), for CT, Structural tokens (e.g. type identifiers, and language keywords) are more important, and CR task has the middle ground.

We also found specific scenarios where these code models fail to perform as expected, and we show that in those cases, there is a lack of proper attention distribution in models, regarding token types. Moreover, we recommend potential solutions that may alleviate these observed challenges.

5.2 Future Works

In general, there are valuable works that have been published in the field of XAI for SE. Interesting ideas have been suggested, and an encouraging growing trend has started. However, despite all of these efforts, there is a long way ahead. ML models are improving their performance and growing more complex gradually and yet there is a noticeable lack of interest from software practitioners in using them in the real world, partly due to the lack of transparency and trust issues.

As we mentioned in 3.3.1 there are still unexplored areas such as software design and testing, and tasks in SE that already have taken advantage of ML models and yet have not been studied in terms of explainability. Code comment generation, code generation, vulnerability detection, test case generation, bug classification, and program repair are some of the most important tasks that require more attention. Another note about these unexplored tasks and in general, all generation-based tasks is that their advancements rely heavily on deep neural networks. DNNs have been a great asset for researchers when it comes to searching in a huge space (i.e. vocabulary of words), thus they are ideal for this type of task. Code models and their success in code document generation, code translation, and code repair are good examples. However, they are incomparably more complex than classical ML models, so they are harder to explain. Nonetheless, this complexity is the exact reason that they require explanation.

We tried to cover this gap in our empirical study. However, there is still plenty of unexplored potentials hidden in this field. Other tasks and code models can be explained using our suggested methodology, and other XAI methods can be used to provide different types of explainability for the same models. One good direction is using post-hoc model-agnostic XAI methods as a pair with internal features such as self-attention or end-to-end attention mechanism and comparing their results' overlap. Also, The insight that attention-based explanation offered for each model and task can be used to alter the internal features of the model in a way that can potentially improve the performance. For instance, the attention score of token categories that are shown to be less effective in a specific task can be manually suppressed in the prediction phase to reduce the noise of the model. On the other hand, if the category is supposed to have a higher impact but

the model was unable to learn it in the training, it can get amplified to potentially help the model's decision.

Finally, if we look at our selected studies, we can see a clear lack of evaluation consistency among them. Some studies just explain the ML models for the sake of explaining and pay no more interest in further analysis, evaluation, or justification for the provided explanations. Considering the fact that explanations are supposed to help the models' users, we believe when objective evaluation is not possible or sufficient, trying human evaluation by experts, is a satisfactory solution.

Bibliography

[1] Finale Doshi-Velez, Yaorong Ge, and Isaac Kohane. Comorbidity clusters in autism spectrum disorders: an electronic health record time-series analysis. *Pediatrics*, 133(1):e54–e63, 2014.

[2] Amir E Khandani, Adlar J Kim, and Andrew W Lo. Consumer credit-risk models via machine-learning algorithms. *Journal of Banking & Finance*, 34(11):2767–2787, 2010.

[3] Hong Hanh Le and Jean-Laurent Viviani. Predicting bank failure: An improvement by implementing a machine-learning approach to classical financial ratios. *Research in International Business and Finance*, 44:16–25, 2018.

[4] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

[5] Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi. Pyse: Automatic worst-case test generation by reinforcement learning. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 136–147. IEEE, 2019.

[6] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 479–490. IEEE, 2019.

[7] Foozhan Ataiefard, Mohammad Jafar Mashhadi, Hadi Hemmati, and Neil Walkinshaw. Deep state inference: Toward behavioral model inference of black-box software systems. *IEEE Transactions on Software Engineering*, 2021.

[8] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 47(1):67–85, 2018.

[9] Xincheng He, Lei Xu, Xiangyu Zhang, Rui Hao, Yang Feng, and Baowen Xu. Pyart: Python api recommendation in real-time. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1634–1645. IEEE, 2021.

[10] Alexsandro Souza Filippetto, Robson Lima, and Jorge Luis Victória Barbosa. A risk prediction model for software project management based on similarity analysis of context histories. *Information and Software Technology*, 131:106497, 2021.

[11] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. *Advances in Neural Information Processing Systems*, 31, 2018.

[12] Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R Lyu. Cradle: Deep code retrieval based on semantic dependency learning. *Neural Networks*, 141:385–394, 2021.

[13] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019.

[14] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

[15] Nima Miryeganeh, Sepehr Hashtroudi, and Hadi Hemmati. Globug: using global data in fault localization. *Journal of Systems and Software*, 177:110961, 2021.

[16] Passakorn Phannachitta. On an optimal analogy-based software effort estimation. *Information and Software Technology*, 125:106330, 2020.

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[18] Daniel Perez and Shigeru Chiba. Cross-language clone detection by learning over abstract syntax trees. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 518–528. IEEE, 2019.

[19] Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. Program synthesis and semantic parsing with learned code idioms. *Advances in Neural Information Processing Systems*, 32, 2019.

[20] Bowen Xu, Amirreza Shirani, David Lo, and Mohammad Amin Alipour. Prediction of relatedness in stack overflow: deep learning vs. svm: a reproducibility study. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2018.

[21] Mohammad Jafar Mashhadi and Hadi Hemmati. Hybrid deep neural networks to infer state models of black-box systems. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 299–311. IEEE, 2020.

[22] Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. Explainable software analytics. In *Proceedings of the 40th international conference on software engineering: New ideas and emerging results*, pages 53–56, 2018.

[23] Joymallya Chakraborty, Tianpei Xia, Fahmid M Fahid, and Tim Menzies. Software engineering for fairness: A case study with hyperparameter optimization. *arXiv preprint arXiv:1905.05786*, 2019.

[24] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[25] David Gunning and David Aha. Darpa's explainable artificial intelligence (xai) program. *AI magazine*, 40(2):44–58, 2019.

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[27] G Shobha, Ajay Rana, Vineet Kansal, and Sarvesh Tanwar. Code clone detection—a systematic review. *Emerging Technologies in Data Mining and Information Security*, pages 645–655, 2021.

[28] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. Pyexplainer: Explaining the predictions of just-in-time defect models. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 407–418. IEEE, 2021.

[29] Jack Humphreys and Hoa Khanh Dam. An explainable deep model for defect prediction. In *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pages 49–55. IEEE, 2019.

[30] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*, pages 184–195, 2020.

[31] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[32] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[33] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[34] Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial intelligence*, 267:1–38, 2019.

[35] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[36] Leilani H Gilpin, David Bau, Ben Z Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*, pages 80–89. IEEE, 2018.

[37] Feiyu Xu, Hans Uszkoreit, Yangzhou Du, Wei Fan, Dongyan Zhao, and Jun Zhu. Explainable ai: A brief survey on history, research areas, approaches and challenges. In *CCF international conference on natural language processing and Chinese computing*, pages 563–574. Springer, 2019.

[38] Deirdre K Mulligan, Joshua A Kroll, Nitin Kohli, and Richmond Y Wong. This thing called fairness: Disciplinary confusion realizing a value in technology. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–36, 2019.

[39] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. A survey on bias and fairness in machine learning. *ACM Computing Surveys (CSUR)*, 54(6):1–35, 2021.

[40] Ribana Roscher, Bastian Bohn, Marco F Duarte, and Jochen Garcke. Explainable machine learning for scientific insights and discoveries. *Ieee Access*, 8:42200–42216, 2020.

[41] Peter ED Love, Weili Fang, Jane Matthews, Stuart Porter, Hanbin Luo, and Lieyun Ding. Explainable artificial intelligence: Precepts, methods, and opportunities for research in construction. *arXiv preprint arXiv:2211.06579*, 2022.

[42] Julia El Zini and Mariette Awad. On the explainability of natural language processing deep models. *ACM Computing Surveys (CSUR)*, 2022.

[43] Marina Danilevsky, Kun Qian, Ranit Aharonov, Yannis Katsis, Ban Kawas, and Prithviraj Sen. A survey of the state of explainable ai for natural language processing. *arXiv preprint arXiv:2010.00711*, 2020.

[44] Diogo V Carvalho, Eduardo M Pereira, and Jaime S Cardoso. Machine learning interpretability: A survey on methods and metrics. *Electronics*, 8(8):832, 2019.

[45] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[46] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

[47] Zachary C Lipton. The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery. *Queue*, 16(3):31–57, 2018.

[48] Filip Karlo Došilović, Mario Brčić, and Nikica Hlupić. Explainable artificial intelligence: A survey. In *2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 0210–0215. IEEE, 2018.

[49] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-López, Daniel Molina, Richard Benjamins, et al. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information fusion*, 58:82–115, 2020.

[50] Bas HM van der Velden, Hugo J Kuijf, Kenneth GA Gilhuijs, and Max A Viergever. Explainable artificial intelligence (xai) in deep learning-based medical image analysis. *Medical Image Analysis*, page 102470, 2022.

[51] Thomas Rojat, Raphaël Puget, David Filliat, Javier Del Ser, Rodolphe Gelin, and Natalia Díaz-Rodríguez. Explainable artificial intelligence (xai) on timeseries data: A survey. *arXiv preprint arXiv:2104.00950*, 2021.

[52] Maria Sahakyan, Zeyar Aung, and Talal Rahwan. Explainable artificial intelligence for tabular data: A survey. *IEEE Access*, 9:135392–135422, 2021.

[53] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Methods for interpreting and understanding deep neural networks. *Digital signal processing*, 73:1–15, 2018.

[54] Alexandre Heuillet, Fabien Couthouis, and Natalia Díaz-Rodríguez. Explainability in deep reinforcement learning. *Knowledge-Based Systems*, 214:106685, 2021.

[55] Yanming Yang, Xin Xia, David Lo, and John Grundy. A survey on deep learning for software engineering. *ACM Computing Surveys (CSUR)*, 54(10s):1–73, 2022.

[56] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–58, 2022.

[57] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):1–31, 2015.

[58] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[59] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 544–547, 2014.

[60] Romi Satria Wahono. A systematic literature review of software defect prediction. *Journal of Software Engineering*, 1(1):1–16, 2015.

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[62] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[63] Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. Automated query reformulation for efficient search based on query logs from stack overflow. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1273–1285. IEEE, 2021.

[64] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[65] Thanh Nguyen, Peter C Rigby, Anh Tuan Nguyen, Mark Karanfil, and Tien N Nguyen. T2api: Synthesizing api code usage templates from english texts with statistical translation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1013–1017, 2016.

[66] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. Improved automatic summarization of subroutines via attention to file context. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 300–310, 2020.

[67] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–146. IEEE, 2017.

[68] Lizhen Liu, Xiao Hu, Wei Song, Ruiji Fu, Ting Liu, and Guoping Hu. Neural multitask learning for simile recognition. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1543–1553, 2018.

[69] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[70] Yi Li, Shaohua Wang, and Tien N Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 602–614, 2020.

[71] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959, 2019.

[72] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611, 2020.

[73] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. Actionable analytics: Stop telling me what it is; please tell me what to do. *IEEE Software*, 38(4):115–120, 2021.

[74] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. Explainable AI for Software Engineering. *arXiv preprint arXiv:2012.01614*, 2020.

[75] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and John Grundy. Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, page To Appear, 2021.

[76] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Hoa Khanh Dam, and John Grundy. An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering (TSE)*, page To Appear, 2020.

[77] Dilini Rajapaksha, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Christoph Bergmeir, John Grundy, and Wray Buntine. SQAPlanner: Generating Data-Informed Software Quality Improvement Plans. *IEEE Transactions on Software Engineering (TSE)*, 2021.

[78] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. PyExplainer: Explaining the Predictions of Just-In-Time Defect Models. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[79] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering (TSE)*, 2020.

[80] Chanathip Pornprasit and Chakkrit Tantithamthavorn. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021.

[81] Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon, Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, and Thanwadee Sunetnanta. Jitbot: An explainable just-in-time defect prediction bot. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1336–1339. IEEE, 2020.

[82] Eoin M Kenny and Mark T Keane. Explaining deep learning using examples: Optimal feature weighting methods for twin systems using post-hoc, explanation-by-example in xai. *Knowledge-Based Systems*, 233:107530, 2021.

[83] Akash Kumar Mohankumar, Preksha Nema, Sharan Narasimhan, Mitesh M Khapra, Balaji Vasan Srinivasan, and Balaraman Ravindran. Towards transparent and explainable attention models. *arXiv preprint arXiv:2004.14243*, 2020.

[84] Goro Kobayashi, Tatsuki Kuribayashi, Sho Yokoi, and Kentaro Inui. Attention is not only a weight: Analyzing transformers with vector norms. *arXiv preprint arXiv:2004.10102*, 2020.

[85] Shengzhong Liu, Franck Le, Supriyo Chakraborty, and Tarek Abdelzaher. On exploring attention-based explanation for transformer models in text classification. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 1193–1203. IEEE, 2021.

[86] Jürgen Cito, Isil Dillig, Seohyun Kim, Vijayaraghavan Murali, and Satish Chandra. Explaining mispredictions of machine learning models using rule induction. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 716–727, 2021.

[87] Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, and Satish Chandra. Counterfactual explanations for models of code. *arXiv preprint arXiv:2111.05711*, 2021.

[88] Anjan Karmakar and Romain Robbes. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1332–1336. IEEE, 2021.

[89] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. What does bert look at? an analysis of bert's attention. *arXiv preprint arXiv:1906.04341*, 2019.

[90] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.

[91] TP Pushphavathi, V Suma, and V Ramaswamy. A novel method for software defect prediction: hybrid of fcm and random forest. In *2014 International Conference on Electronics and Communication Systems (ICECS)*, pages 1–5. IEEE, 2014.

[92] Valentín Moreno, Gonzalo Génova, Eugenio Parra, and Anabel Fraga. Application of machine learning techniques to the flexible assessment and improvement of requirements quality. *Software Quality Journal*, 28(4):1645–1674, 2020.

[93] Márcio P Basgalupp, Rodrigo C Barros, Tiago S da Silva, and André CPLF de Carvalho. Software effort prediction: A hyper-heuristic decision-tree based approach. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1109–1116, 2013.

[94] Jirayus Jiarpakdee. Towards a more reliable interpretation of defect models. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 210–213. IEEE, 2019.

[95] Fabiano Dalpiaz, Davide Dell'Anna, Fatma Basak Aydemir, and Sercan Çevikol. Requirements classification with interpretable machine learning and dependency parsing. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*, pages 142–152. IEEE, 2019.

[96] Jiao Sun, Q Vera Liao, Michael Muller, Mayank Agarwal, Stephanie Houde, Kartik Talamadupula, and Justin D Weisz. Investigating explainability of generative ai for code through scenario-based design. In *27th International Conference on Intelligent User Interfaces*, pages 212–228, 2022.

[97] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim A Laredo, and Alessandro Morari. Towards reliable ai for source code understanding. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 403–411, 2021.

[98] Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. Explainable software analytics. In *Proceedings of the 40th international conference on software engineering: New ideas and emerging results*, pages 53–56, 2018.

[99] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. Pyexplainer: Explaining the predictions of just-in-time defect models. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 407–418. IEEE, 2021.

[100] Themistoklis Diamantopoulos and Andreas Symeonidis. Towards interpretable defect-prone component analysis using genetic fuzzy systems. In *2015 IEEE/ACM 4th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pages 32–38. IEEE, 2015.

[101] Hakim Lounis, Tamer Fares Gayed, and Mounir Boukadoum. Machine-learning models for software quality: a compromise between performance and intelligibility. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pages 919–921. IEEE, 2011.

[102] Ahmet Okutan and Olcay Taner Yıldız. Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19(1):154–181, 2014.

[103] Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 369–379. IEEE, 2021.

[104] Jirayus Jiarpakdee, Chakkrit Kla Tantithamthavorn, and John Grundy. Practitioners' perceptions of the goals and visual explanations of defect prediction models. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 432–443. IEEE, 2021.

[105] Wei Zheng, Tianren Shen, and Xiang Chen. Just-in-time defect prediction technology based on interpretability technology. In *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, pages 78–89. IEEE, 2021.

[106] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering*, 2020.

[107] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. Neural network-based detection of self-admitted technical debt: From performance to explainability. *ACM transactions on software engineering and methodology (TOSEM)*, 28(3):1–45, 2019.

[108] Jianjun He, Ling Xu, Yuanrui Fan, Zhou Xu, Meng Yan, and Yan Lei. Deep learning based valid bug reports determination and explanation. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 184–194. IEEE, 2020.

[109] Jack Humphreys and Hoa Khanh Dam. An explainable deep model for defect prediction. In *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pages 49–55. IEEE, 2019.

[110] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*, 46(11):1200–1219, 2018.

[111] Qin Liu, Wen Zhong Qin, Robert Mintram, and Margaret Ross. Evaluation of preliminary data analysis framework in software cost estimation based on isbsg r9 data. *Software quality journal*, 16(3):411–458, 2008.

[112] Md Rafiqul Islam Rabin, Vincent J Hellendoorn, and Mohammad Amin Alipour. Understanding neural code intelligence through program simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 441–452, 2021.

[113] Miroslaw Ochodek, Regina Hebig, Wilhelm Meding, Gert Frost, and Miroslaw Staron. Recognizing lines of code violating company-specific coding guidelines using machine learning. In *Accelerating Digital Transformation*, pages 211–251. Springer, 2019.

[114] Toshiki Mori. Superposed naive bayes for accurate and interpretable prediction. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 1228–1233. IEEE, 2015.

[115] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[116] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[117] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE, 2018.

[118] Chin-Yew Lin and Franz Josef Och. Orange: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, 2004.

[119] Sarah Wiegreffe and Yuval Pinter. Attention is not not explanation. *arXiv preprint arXiv:1908.04626*, 2019.

[120] Sarthak Jain and Byron C Wallace. Attention is not explanation. *arXiv preprint arXiv:1902.10186*, 2019.

[121] Edward Loper and Steven Bird. Nltk: The natural language toolkit. *CoRR*, cs.CL/0205028, 2002.