REFINECODE: ENHANCING CODE QUALITY THROUGH ACTIONABLE CODE REVIEW RECOMMENDATIONS AND INTELLIGENT ISSUE RESOLUTION

SHADIKUR RAHMAN

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 AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
JANUARY 2024

Abstract

Code review is essential for maintaining software development standards, yet achieving effective reviews and issue resolution remains challenging. This thesis introduces RefineCode, an application tool to find actionable code reviews and provide similar code reviews as references within an organization, aiding developers in resolving issues effectively. To this end, we collected 9,500 code reviews from five private projects in an industrial setting and empirically evaluated various classification methods for identifying actionable code reviews. RefineCode automatically recommends relevant solutions from Stack Overflow based on textual similarity and entity linking between code reviews and Stack Overflow issues. Additionally, it integrates a chatbot feature, leveraging large language models to propose potential solutions for actionable code reviews. These features empower developers to make informed decisions, enhancing code quality by guiding issue resolution without reinforcing misunderstandings.

Acknowledgements

To begin, I wish to offer my sincere and humble recognition of the immense blessings and divine guidance bestowed upon me by Almighty Allah. It is through His boundless mercy and grace that I have achieved the successful completion of my Master's degree in Computer Science at York University, with a particular emphasis on a thesis-based program.

I extend my heartfelt gratitude to **Prof. Enamul Hoque Prince**, my thesis supervisor, whose expertise and unwavering support have played a pivotal role in shaping the quality and direction of my research endeavors. His mentorship has been invaluable.

I would also to thank **Prof. Marin Litoiu** for being a supervisory committee member of my thesis.

Moreover, I am deeply appreciative of my beloved wife, **Umme Ayman Koana**, for her constant encouragement and support, which have been a source of strength and motivation throughout this academic odyssey.

Lastly, I wish to thank my parents, siblings, and cherished friends, who, by Allah's will, have been instrumental in my academic journey through their unwavering support and encouragement. Their presence has been a divine blessing, Without them, this achievement would not have been possible.

Preface

This thesis is submitted to the Faculty of Graduate Studies in partial fulfillment of the requirements for a Master of Science Degree in Computer Science. The entire work presented here is done primarily by Shadikur Rahman under the supervision of Dr. Enamul Hoque Prince. This thesis has been submitted for publication as:

- Rahman, Shadikur, Umme Ayman Koana, and Enamul Houque Prince. "RefineCode: Enhancing Code Review Solutions to Improve Code Quality with Language Models." Proceedings of the Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)**(Submitted)**.

The collaborative efforts of the three authors in this paper are distinctly reflected in their respective contributions. The first author Shadikur Rahman undertook the crucial tasks of data collection and preparation, model implementation and evaluation, along with manuscript writing. Umme Ayman Koana assisted to the literature review and its associated write-up. Finally, Enamul Hoque Prince, served

as the supervisor, offering guidance throughout the entire process. This included overseeing the overall work, providing insights on paper organization, and offering instructive mentorship.

Table of Contents

List of Tables

List of Figures

1 Introduction

Code review is a fundamental practice in software development that aims to improve code quality, fix bugs, and increase developer productivity. This process involves project developers reviewing each other's code to determine if it meets the necessary quality standards for inclusion in the project's main codebase [8]. However, with the growing size and complexity of software projects, developers often face the challenge of navigating through a large volume of code review comments. Furthermore, there is a high rate of irrelevant comments and a lack of personalization, making it difficult for developers to efficiently find and utilize relevant comments. The manual inspection of code review can present a significant challenge, as it is both complex and time-consuming [55]. As a result, in contemporary code review practices, the utilization of tool-assisted reviews, specifically automated code review tools, has gained popularity[4]. Despite this trend, there is a notable absence of application tools offering automated code review recommendations and classifications to developers on GitHub.

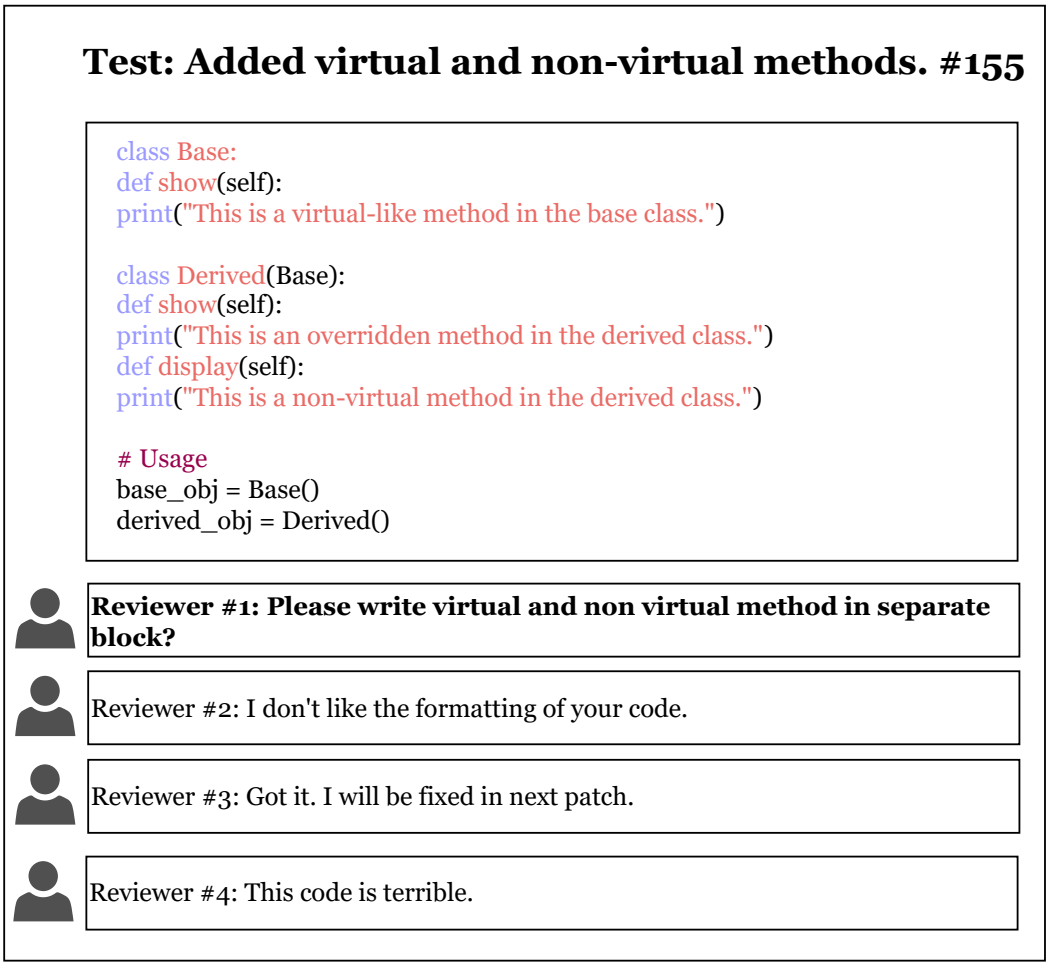

Figure 1.1: An example of classifying code reviews to solve the issue for specific code commit #155. The text in bold font identifies the actionable code review.

To address this gap, we focus on the following key tasks: (i) *Classifying code review*: where the objective is to categorize actionable review comments based on a new code commit and a set of code reviews provided by peers (See figure 1.1). (ii) *Suggesting similar reviews from GitHub and entity linking to recommend Stack*

Overflow posts: When actionable reviews are identified, the task is to provide similar code reviews that were previously resolved by other developers in the same organization. Additionally, it provides relevant information from Stack Overflow to help resolve the issue (See figure 4.3). (iii) *Providing solutions based on search queries*, wherein actionable reviews are treated as search queries, and the goal is to offer solutions based on the given query by leveraging large language models (See figure 4.5)

1.1 Motivation

Nowadays, many industries have increasingly adopted modern code review practices as a fundamental strategy to constantly monitor and improve the quality of code changes. This approach highlighted in research by Alomar et al. [2] and Cunha et al. [13], not only ensures code reliability but also fosters a culture of collaborative learning and continuous improvement among developers. Sadowski et al. [55] focused on Google's practice and analyzed its current state of code review. They point out in this analysis that tool-based code reviews have become the norm for a wide range of open-source and industrial systems. Bachelli et al. [4] focuses on the evolution of code review techniques in software development. Their study highlights the significance of the transition by emphasizing how modern methods are more adaptive and efficient in today's dynamic software development environment.

This evolution of code review practices reflects broader changes in software engineering aimed at optimizing the review process for better quality and productivity in software projects. In their research, they conducted a study on code review anticipations and found that automated code review tools can effectively identify certain types of problems and reduce the number of manual inspections required. Pascarella et al. [45] provided a taxonomy of information needs in a code review. According to this taxonomy, reviewers require information with regards to sustainability of an alliterative solution, correct understanding, rationale, code context, necessity, specialized expertise, and modularity of changes (splittable) for a successful review process. Nowadays, it has become a difficult task for developers to find important bug issues from code review comments. In that case, recommendation and classification systems have become the modern solution to the growing number of code review comments. Rahman et al. [57] introduced an application tool called ToxicCR, which detects toxic discussions in code review. They anticipate that this tool helps combat toxicity in the FOSS community. Manual code review is time-consuming, and automating it can increase efficiency. Existing techniques struggle to capture the subtle differences between the two code versions. As a response, Shi et al. [59] introduce DACE, a deep model that excels at learning correction features by contrasting code changes in the context of the source code. Their experiments on six open-source projects show that DACE outperforms other

automated code review methods.

Analogies and examples have been human's conventional way of explaining their cognition and decisions. In today's world, the use of examples is one of the methods for decision explanation and explainable AI. Adequately explaining machine-made decisions proved to be effective in convincing domain experts to follow the recommendation and accept the intelligence of the machines. In this context, and to address developers' requirements for rationale, code context, and a clear understanding of code reviews [4], we studied the use of analogies by providing context-specific examples along with code reviews. The use of examples to elucidate code review issues within software teams has been largely unexplored in existing literature. While developers traditionally engage in manual searches for examples, the automated retrieval of such instances from software repositories is not only feasible but also streamlines developers' activities.

1.2 Thesis Contributions

This thesis focuses on enhancing the efficiency and effectiveness of the code review process using natural language processing techniques. The key contributions are summarized below:

- For improving code quality, we developed a classifier to predict actionable

code review comments using feature engineering-based classifiers and transformer-based classifiers.

- We evaluated the classification methods for identifying actionable code reviews and presented a corpus of code reviews for this task. Through an empirical evaluation, we examined the effectiveness of different models in addressing this classification challenge.

- We introduced RefineCode, a system leveraging the aforementioned classifier to identify similar code reviews within a GitHub project. It also extracts relevant information from Stack Overflow based on the developer's code reviews using the entity linking process. Finally, RefineCode incorporates a chatbot using the pre-trained Stable Beluga (Llama2) [64] model, which aims to support developers by showing relevant example solutions to understand code problems mentioned in the code reviews.

Overall, this thesis addresses code review challenges by aiding in issue prioritization and promoting efficient problem-solving. The proposed techniques contribute to a more streamlined and effective code review process, ultimately enhancing software quality and fostering improved collaboration among developers.

1.3 Research Questions (RQs)

In this research, we have briefly analyzed the code review comments and performed a unique technique for classifying and recommending code reviews. Here, we developed RefineCode, a tool that uses feature engineering-based approaches and large language models to identify actionable review comments and provide relevant suggestions in code reviews. It utilizes entity linking and text similarity measures to fetch relevant information from Stack Overflow to help developers solve coding problems. In addition, we implemented the RefiCode chatbot to facilitate code review by showing developers clear examples to understand code review issues. In pursuit of our research objectives, we address two key research questions (**RQ**s):

RQ1: Which model is the most effective in identifying actionable reviews from GitHub projects?

Why and How: Identifying actionable reviews in GitHub projects is crucial for prioritizing and addressing critical issues. To address this research question, we collected data from five major projects, labeled 9,500 code reviews, and evaluated various feature engineering-based approaches and transformer-based models (BERT and DistilBERT) to determine the most effective one. Our findings provide valuable insights that can guide developers and project managers in prioritizing important code reviews and enhancing the overall code review processes.

RQ2: How can RefineCode assist developers in resolving coding issues based on code reviews?

Why and How: Developers often encounter coding issues and rely on platforms like Stack Overflow for solutions. However, finding relevant answers and understanding code examples can be challenging due to the vast amount of information. In our thesis, we find similar code reviews from GitHub projects by measuring textual similarity. We also help developers find potential solutions from Stack Overview using Named Entity Recognition (NER) and chunking processes. Finally, RefineCode Chatbot utilizes the Stable Beluga (Llama2) pre-trained model to potential solutions based on specific comments in code reviews.

1.4 Organization of the Thesis

Before moving to the next chapter, we give a brief description below regarding how the following chapters of this thesis are organized.

- We begin Chapter 2 with a brief introduction to machine learning and deep learning, followed by their applications in natural language processing that are relevant to our research problem. We then discuss various deep learning architectures followed by various word embedding and sentence embedding techniques that have been widely used in recent years in deep neural models

for a wide range of natural language understanding and generation problems. We then review the Transformer architecture and the various language models based on it. Finally, we reviewed the research questions (RQs) that we studied in this thesis.

- Then in Chapter 3, we first briefly discuss the background of the Code Review Classification that we study for improving the code quality in this thesis. In the same chapter, we describe our proposed approaches for this task along with the datasets that we used to evaluate the proposed models. Moreover, our experimental details as well as the results are also discussed in this chapter.

- In Chapter 4, we present the RefineCode application that assists the developer in solving the issue based on code reviews. We present the key features of the system and demonstrate the utility of the system with illustrative use case scenarios.

- Finally, the concluding remarks of this thesis as well as our plans for future work are discussed in Chapter 5.

2 Literature Review

In this section, we provide an overview of the recent work in identifying code review, recommendations, and entity linking with Stack Overflow. We start with a literature review on the code review analysis which we are motivated to use in our thesis. Then we describe recent state-of-the-art models based on machine learning and transformers. After a literature review on machine learning and transformer-based models, we describe the related work on research question tasks that we plan to solve: i) automating code review, ii) suggesting similar code review, and iii) entity linking with Stack Overflow based on review.

2.1 Supervised Learning in Code Review

Code review is a complex software engineering practice that aims to improve code quality, identify bugs, and ensure maintainability. Traditional code review processes rely heavily on manual inspections by developers, which can be time-consuming and error-prone. With the advent of machine learning (ML) techniques, there

is increasing interest in automating and improving code review processes. This section explores the state of the art in applying machine learning to code review and highlights key research contributions to the field.

Fregnan et al. [20] aims to identify the immediate impact of the review on the codebase. Despite their contributions, these classifications were conducted manually, limiting scalability, and their practical implications for software developers were underappreciated. This work advances the area of research by exploring the potential of machine learning in modifying auto-classifying reviews and assessing its real-world relevance through developer interviews and qualitative analysis. Dorin et al. [18] explore the integration of machine learning and image recognition to identify pre-screen and immature code segments. Such an approach builds on previous efforts to automate code quality testing with the goal of increasing code review efficiency by focusing on relevant application details. Unlike their approach, our approach focuses on providing actionable feedback and providing developers with a more personalized and contextually relevant code review experience.

Staron et al. [63] addressed the challenge of automating the massive flow of code patches in a CI flow. They aimed to auto-extract coding guideline rules and pinpoint specific code fragments or lines that would benefit most from manual reviews, targeting the perspective of software designers. Lin et al. [35] builds on the existing literature by emphasizing the potential of Neural Machine Translation (NMT)

models that, especially when combined with code granularity, mark a step forward in the efficiency of automated code review. Ochodek et al.[42] introduced an automated approach to detect violations of companyspecific coding guidelines within large industry codebases. They developed a machine-learning tool capable of learning from a limited sample of non-consistent code lines and efficiently identifying similar instances across millions of lines of code. Lal et al. [29] introduced a novel machine learning technique designed to further optimize and accelerate the code review process, aiming to ensure a more efficient and accurate evaluation of submitted code. While their methodology is commendable, our work on RefineCode underscores the importance of clarity and context in code reviews, ensuring developers gain deeper insights from the feedback.

2.2 Deep Learning in Code Review

Traditional code reviews rely heavily on human expertise, leading to potential oversights and inconsistencies. Deep learning, a subset of machine learning, has demonstrated its ability to understand complex patterns in data, and its application to code review is beginning to transform this domain. Deep learning (DL) has witnessed rapid advances in natural language processing, with significant advances in language modeling, machine translation and paragraph understanding [30], [40], [43], [58].

Gupta et al. [24] introduced DeepCodeReviewer (DCR), an advanced system

powered by deep learning. DCR efficiently recommends relevant reviews for common code issues, using historical peer reviews from Microsoft's internal code repository for its training. Initial tests demonstrated the model's ability to determine the relevance of reviews related to specific code snippets Additionally, a user study and survey confirmed a high acceptance rate for DCR's suggestions, aligning with the goal of making code reviews more error-focused. Unlike their recommendation-focused approach, our work highlights a holistic review process that caters to both fresher and experienced developers.

Li [34] proposed model, CodeReviewer, uses a large dataset obtained from open-source projects and employs four unique pre-training tasks designed specifically for the code review context. This model has been evaluated across three essential tasks in code review, demonstrating a significant superiority over previous models. This superiority is attributed to the adaptive pre-training task and the multilingual dataset, which improves the understanding of code switching. Li et al. [32] presents AUGER, a tool that uses the text-to-text transfer transformer (T5) model to automatically generate review comments. Based on data from 11 Java projects, AUGER outperforms other methods in ROUGE-L by 37.38% and generates 29% valuable comments. While Li's model focuses on adaptability, our approach is prepared towards scalability, ensuring efficient code reviews for projects of all sizes.

Hong et al. [26] introduce CommentFinder, a more efficient method for recom-

mending code review comments. In tests of more than 151,000 modified methods, CommentFinder outperformed previous methods in accuracy by 32% and was 49 times faster, suggesting that it can provide faster and more accurate review support in real-world settings. The intersection of deep learning and code review has opened up exciting avenues of research. As models become more sophisticated and datasets more extensive, the potential for fully automated, highly accurate code review systems increases. Our work diverges from Hong's by incorporating real-time collaboration features, which allow developers to interact and discuss instantly generated code review comments.

2.3 Code Review Analysis

Recognizing the substantial time spent by software developers on code reviews, it is crucial to determine the factors that contribute to their effectiveness and enhance their overall code quality. Rahman et al.[50] compared useful and non-useful review comments, focusing on textual features and reviewer experience. Similarly, Bosu et al. [8] conducted a hybrid three-stage research, qualitatively examining factors that make code reviews valuable for developers. They developed a classification model to distinguish between useful and non-useful feedback. In a more recent study, Rahman et al. [52] focused on predicting the clarity of code review comments in GitHub projects.

In contrast to the aforementioned studies, our focus centers on predicting which code reviews necessitate actions, in terms of making specific changes to the code. By identifying actionable reviews, our objective is to streamline the process for developers, reducing the time and effort required to discern which reviews need immediate attention and resolution.

In 2017, Bachelli et al.[4] identified the information needs of developers during code review. Their research highlighted that automated code review tools can pinpoint specific problem types, reducing the need for manual checks. Moreover, many industries now use modern code review techniques as a crucial method for continuous monitoring and enhancement of code quality[2, 13, 51]. Our research expands on this theme by exploring how industries can seamlessly integrate traditional and modern review practices.

Anderson et al. [66], in their study involving 57,498 code changes across seven open-source projects, examined the effectiveness of six machine learning algorithms in predicting successful design modifications. They focused on how machine learning can aid in understanding the nuances of code review by analyzing the content of reviewers' discussions. In a separate study, Li et al. [33] conducted an analysis of three well-known open-source software projects hosted on GitHub. They developed a detailed classification system for review comments, dividing them into 11 distinct sub-categories. Furthermore, Li et al.[31] in a later 2019 study introduced a new

deep learning model called DeepReview, which is based on Convolutional Neural Networks (CNN). While they use deep learning for automated review, our work focuses on combining deep learning insight with human insight for a more balanced review.

Yang et al. proposed CodeHow, an approach that recommends relevant code snippets from existing code reviews to assist developers in understanding and fixing issues [37]. Silva et al. developed CROKAGE, a tool that improves code search by providing comprehensive solutions with code examples and explanations [61]. It outperforms baselines and the state-of-the-art, showcasing the potential of recommendation systems in supporting developers. Siow et al. [62] proposed CORE, an automated code review system based on code changes and reviews, which is evaluated for Java projects from GitHub. Balachandran et al. [5], introduced a tool called Review Bot for the integration of automated static analysis with the code review process. Review Bot uses the output of multiple static analysis tools to automatically publish reviews. Through a user study, they show that integrating static analysis tools into the code review process can improve code review quality. Code reviews become challenging when a changeset contains multiple separate code changes. Barnett et al. [6] presented CLUSTERCHANGES, an automated method for breaking changesets. They evaluate its effectiveness through quantitative analysis and a user study for a comprehensive evaluation. Although various studies have

contributed significantly to understanding and improving the code review process, our work proposes a holistic, integrated approach, which best bridges traditional and modern practices.

2.4 Transformer Models for Automating Code Reviews

Transformer models have catalyzed a transformative leap in the automation of code reviews. Their capability to understand intricate patterns in code and natural language has been harnessed by various studies. In this section, we delve into notable efforts in this space and contrast them with our unique approach. Tufano et al. [65], leverage a pre-trained Text-To-Text Transfer Transformer (T5) model to outperform previous DL models in automating code review tasks. Their experiments use a larger and more challenging dataset of code review activities, building on prior research. Unlike their approach which mainly relies on raw transformer power, our approach integrates domain-specific optimizations tailored for different code review contexts.

Zhang et al. [73] conducted a systematic evaluation of five existing SA4SE tools (Stanford CoreNLP, SentiStrength, SentiStrength-SE, SentiCR, and SentiSD) and fine-tuned four state-of-the-art pre-trained Transformer-based models (BERT, RoBERTa, XLNet, and ALBERT) on six SE datasets. This marks the first attempt to fine-tune these models for SA4SE tasks. While Zhang et al. Primarily

focused on sentiment analysis, our application tools provide a more holistic review solution by considering multifaceted factors beyond just sentiment in code review.

Wang et al. [68] introduced CodeT5, a unified pre-trained encoder-decoder Transformer model. They also present a novel identifier-aware pre-training task, allowing the model to distinguish code tokens that are identifiers and recover them when masked. Additionally, they proposed a bimodal dual-generation task that utilizes user-written code comments to enhance alignment between Natural Language (NL) and Programming Language (PL). Feng et al. [19] presented CodeBERT, a bimodal pre-trained model designed for both programming language (PL) and natural language (NL). For that, they fine-tune CodeBERT on two NL-PL applications and achieve state-of-the-art performance in natural language code search and code documentation generation.

Paul et al. [46] investigated the potential of large language models pre-trained in both Natural Language (NL) and Programming Language (PL) to improve automated program repair. They applied state-of-the-art models like PLBART and CodeT5 to natural language-based program repair datasets, which contained code review and subsequent code changes. They also explored the performance of code generative models like Codex and GPT-3.5-Turbo. Our research extends this by not only using dual-model training but also embedding real-time developer feedback to ensure alignment with actual coding objectives. In our application tools,

we applied pre-train models BERT and DistilBERT with fine tuning to identify the code reviews and provide code examples using code generative models Stable Beluga (Llama2) [64].

2.5 Recommendation in Code Review

There are four main recognized methods for clarifying decision-making processes to domain experts[1]. The recommendation is a widely acknowledged and effective method for elucidating decisions. The emergence of machine learning (ML) in automating decision-making has broadened the scope of research into ML explanation methods[23]. The advent of ML for automating decisions has expanded research in ML explanations. Domain experts tend to trust and follow machine-generated decisions if they align with logical reasoning.

Zanjani et al. [71] introduce cHRev, a method that recommends reviewers based on their historical contributions to previous reviews. They evaluate cHRev on open-source projects at Microsoft and a commercial codebase, showing significant improvements over existing approaches that rely on generic review information or source code repository data.

In our study, we build upon these previous works by proposing a novel technique that combines code review recommendations and entity linking with Stack Overflow solutions. By integrating these approaches, we aim to provide developers

with personalized and contextually relevant recommendations, assisting them in understanding and resolving coding issues effectively.

3 Actionable Code Review Classification

In this chapter, we answer the **RQ1**, i.e., which model is the most effective in identifying actionable reviews from GitHub projects? In particular, we provide an overview of the techniques for classifying actionable code reviews including Feature Engineering with Machine Learning (ML) algorithms and transformer-based models to classify code reviews. We then compare the performance of different models to understand their effectiveness. The ultimate goal is to develop an application system capable of handling Review Extraction, text preprocessing, text mining, model training and evaluation, and classification of review text using Natural Language Processing (NLP) techniques.

Figure 3.1 shows an overall overview of the research workflow. First, we extracted code reviews from Github Projects using the GitHub REST API. Then, We filter out the ambiguous and unclear reviews. After that, we applied classifiers to identify actionable code reviews. After identifying actionable reviews, we developed RefineCode which utilizes text similarity measures and entity-linking

techniques to identify potential solutions from StackOverflow. In this Chapter, we will describe the code review classification techniques. Then, we will discuss the RefineCode application in the next Chapter.

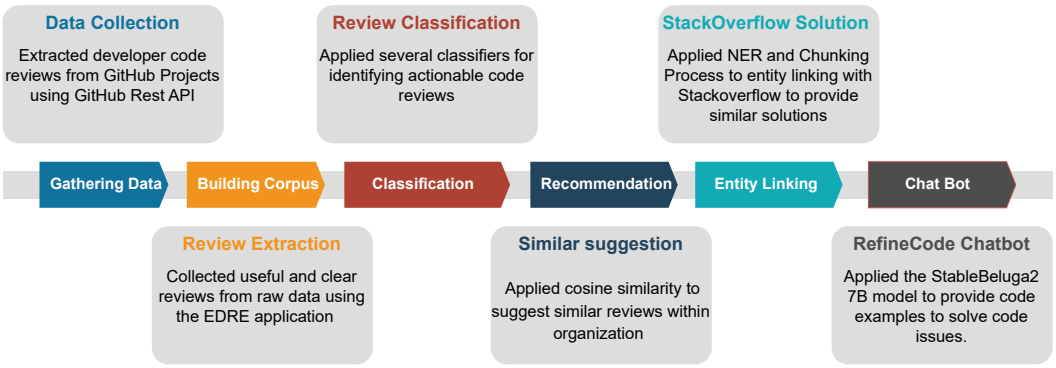

Figure 3.1: The overall overview of the research workflow

3.1 Code Reviews Extraction

At first, we created a corpus of code reviews from GitHub projects, which consists of a total of 9,500 reviews (the data collection process is described in detail in Section 3.5). However, some code reviews exhibit ambiguity or lack meaningful content, therefore it is important to filter them out before identifying actionable code reviews. To address this challenge, we employ a text classifier designed to identify clear code reviews, as detailed in Rahman et al. [52]. Specifically, we opted for the Support Vector Machine Classifier utilizing TF-IDF word vectors as features. This choice resulted in a high F-score of 92% in categorizing the code reviews.

3.2 Text Prepossessing

We apply a standard text preprocessing pipeline to our code reviews, which includes punctuation removal, tokenization, noise removal, POS tagging, lemmatization, and extraction of n-grams. We remove unnecessary punctuation from our sentences. Then, we apply the tokenization process to split words within each sentence. We also remove stop words from the text using NLTK Library [7]. We then use the lemmatization process to map words to their base or root form. Next, we generate a set of sequential word combinations from the input sentences using n-gram models (bi-grams and tri-grams).

3.2.1 Tokenization Process

In this part, we created a bag of words in our text documents and divided them into punctuation marks. However, we made sure that short words like “don’t”, “I’ll”, “I’d” remain as one word. Splitting a given text into smaller parts is called a token. We used the NLTK word tokenizer to parse the input text into a list of words.

3.2.2 Stop Words Removal

To remove the stop words, we used the NLTK library which erases the common and dispensable words from our text sentences. Stop words are common words in a language like "an", "almost", "the", "is", "in". These words have no important meaning and we usually remove the words from our corpus data.

3.2.3 Part of Speech Tagging(POS)

After Text Tokenization, we used the POS tagger function from NLTK to level each word in a sentence into nouns(NN), verbs(VB), adverbs(RB), and adjectives(JJ) etc. For example in the sentence "I should understand how to inject code" we have collected words like 'should' (VB), 'understand' (VB), 'how' (RB), 'inject'(VB), and 'code' (NN).

3.2.4 Lemmatization Process

We used the Lemmatization Process of modifying words to their word Lemma, base, or root form. The goal of lemmatization is to reduce the reflective form of a word and sometimes derive similar forms to a common base form (e.g., worked–work, removed–remove, changes-change, etc.).

Table 3.1: Process of an N-gram

2-gram(bigram)	3-gram(trigram)
"Please enter"	"Please enter your"
"Enter your"	"Enter Your code"
"Your code"	"Your code review"
"code review"	——

3.2.5 N-Gram Process

An N-gram is a contiguous sequence of N words used for predicting the next item in such a sequence. In the context of N-grams, $p(w \mid h)$ denotes the probability of a word w occurring, given the history or context denoted by h, which is the sequence of words preceding w [10]. We used N-gram models to analyze the developer code reviews. This approach allowed us to examine patterns and sequences of language used within the review. By breaking the text into sequences of N words, we gain insight into general phrase structure and word associations. For example, 2-grams and 3-grams of "Please enter your code review" are shown in Table 3.1.

3.3 Feature Extraction Approaches

In this work, we explore feature extraction approaches for classification at two levels: Word-level and sentence-level. By considering different levels of embeddings, we aim to capture a more comprehensive understanding of the data and compare which embedding performed better for our training models.

3.3.1 Word Embedding

For word embedding, we experiment with Term Frequency-Inverse Document Frequency (TF-IDF) for word weighting. TF-IDF calculates the importance of words based on their term frequency and inverse document frequency, providing a suitable text representation for our classifiers. We used TF-IDF to select features and train our classifier, exploiting its strengths in text representation. TF-IDF vectorization assigns a higher weight to words that are more important in a document and less frequent in the corpus. This means that common words such as "the" and "a" are assigned a lower weight while rare words are given higher importance. Apart from that, we also used the N-gram process for sequences between words.

3.3.2 Sentence Embedding

Sentence embedding is a technique in natural language processing where sentences are mapped to vectors of real numbers, representing the sentences in a high-dimensional space. For this process, we utilize models such as Sentence-BERT [17], Universal Sentence Encoder (USE) [11], and Mirror-BERT [36] to train our models.

3.3.2.1 Sentence-BERT

Bidirectional Encoder Representations from Transformers (BERT)[17] is a language model based on the transformer architecture. While we used TF-IDF to extract the term frequency (which works at the word level) we used the BERT model to consider the bidirectional representation of words at the sentence level. Specifically, we adopt Sentence-BERT [53], which generates sentence embeddings by utilizing a modified version of the pretrained BERT network. The embeddings derived from Sentence-BERT serve as features for our classifications. This choice is motivated by the model's ability to yield rich sentence-level embeddings, effectively capturing the contextual nuances of words within each code review.

3.3.2.2 Universal Sentence Encoder

Universal Sentence Encoder(USE) is another sentence embedding technique from Google Research[11]. USE has two separate models of encoding sentences. The first one constructs sentence embeddings using the encoding sub-graph of the transformer architecture. The subgraph utilizes attention to capture the context as well as the sequence of words to generate high-quality embeddings. In this method, the sentence embeddings are generated from word embeddings by simply computing an element-wise sum of the individual word representations. The second variant of USE is called Deep Averaging Network (DAN) which computes the average of input embeddings for words and bi-grams together and then passes the combined representation through a feedforward neural network to generate sentence embeddings. In our thesis, we used the transformer architecture for sentence embedding.

3.3.2.3 Mirror-BERT

Mirror-BERT [36] is a contrastive learning technique that converts pretrained language models like BERT into universal text encoders without any supervision. In this thesis, we used Mirror-BERT for sentence embedding because it is known to be an efficient technique and it does not require any supervised or manually annotated data.

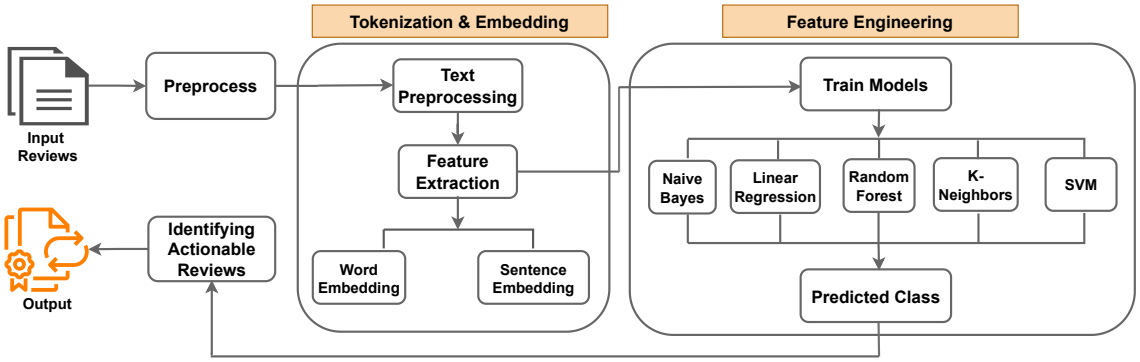

Figure 3.2: Overview of processes used to extract actionable issues from code reviews using supervised classification methods.

3.4 Code Review Classification

In our Section, we describe how we classify code reviews into actionable vs. non-actionable reviews based on feature engineering-based and transformer-based models.

3.4.1 Feature Engineering Based Classification

We utilize the following five supervised learning classification methods which are known to be effective for various text classification tasks [70, 74]. Figure 3.2 shows the overall overview of the code review classification process using these approaches.

- **Logistic Regression:** Logistic regression[27] is a predictive analysis algorithm primarily used for binary classification tasks. It works by estimating

the probability that a given input point belongs to a certain class, making it particularly useful in our context, where the task is a binary classification ("Actionable Review" or "Non-Actionable Review)".

- **Naive Bayes:** Naive Bayes [39] is a simple and efficient classification algorithm based on the Bayes Theorem, widely used in machine learning for tasks such as text classification and sentiment analysis. We used multinomial naive Bayes [28] for code review classification, which is particularly well-suited for code review classification due to its proficiency in handling text data. This model efficiently manages the high-dimensional nature of text by leveraging word frequencies, making it ideal for identifying key patterns in code review comments.

- **Support Vector Machine(SVM):** SVM [12] is a robust and versatile supervised machine learning algorithm which widely used for various classification problems. Its primary objective is to find a hyperplane in a multi-dimensional space that best separates different classes of data points. We chose the SVM for classifying code review comments due to its effectiveness in high-dimensional spaces.

- **Random Forest:** Random Forest [9] is an ensemble machine learning algorithm renowned for its versatility and accuracy in both classification and

regression tasks. We used Random Forest to classify code reviews because of its robustness and accuracy in handling complex, high-dimensional data, which is common in textual analysis. Additionally, its ability to handle unbalanced datasets is critical, given the often skewed nature of code review comments.

- **K-Nearest Neighbors (KNN):** KNN [47] is another supervised classification technique that identifies the k closest data points in the feature space to a given query point and makes predictions based on these neighbors. KNN is distinctive for its lack of an explicit training phase, instead saving the entire training dataset for use during estimation. While this simplicity is advantageous, it comes at the cost of increased computational load, particularly with large datasets. We used KNN for classifying code reviews due to its simplicity and effectiveness, especially in handling the diverse and complex patterns often found in textual data.

Cross-validation: For Cross-validation, we use Scikitlearn's SearchGrid function to tune our cross-validation parameter. After building the grid, we execute our GridSearchCV model passing `classifiers()` for finding the best estimator parameter (`.best_params_`) and n-fold value. As the result of this search, we use $n = 10$ for the cross-validation. So, among the total of 9,500 sentences, we train

our models using 80% of the corpus.

3.4.2 Transformer-based Classification

In this work, we utilize two transformer-based language models, namely BERT [16] and DistilBERT [56], for the classification of reviews into actionable and non-actionable categories. Figure 3.3 provides an overview of the code review classification process employing transformer-based models which are fine-tuned on our labeled data. We describe the two models below.

- **BERT:** BERT [16] leverage the transformer architecture for the deep bidirectional understanding of textual contexts. Unlike traditional models that process text in a linear sequence, BERT analyzes text in both directions simultaneously, allowing for a more nuanced and comprehensive understanding of language. It goes through a two-stage process: extensive pre-training on a large body of text using tasks such as masked language modeling and next-sentence prediction, followed by fine-tuning for specific applications such as sentiment analysis, question answering, and language inference. This approach enables BERT to achieve state-of-the-art results on various NLP tasks, although its large size and complexity require considerable computational resources. In this work, we chose to fine-tune the BERT model on the code review text data given its effectiveness in understanding the textual contexts.

- **Distil-BERT:** DistilBERT [56] is a distilled version of the BERT model, designed to be smaller and more computationally efficient while retaining much of the original model's performance. The model's efficiency makes it a suitable choice for our code review classification tasks.

Fine-tining: For fine-tuning BERT (`bert-base-uncased`) and DistilBERT (`distilbert-base-uncased`) models, we utilize our labeled GitHub project dataset. During the fine-tuning process, we adjust the model's parameters, including hidden_layer="gelu", hidden_size=768, max_position_embeddings=512, and num hidden layers=12, to align with our specific task requirements. Then, we optimize the hyper-parameter using Adam optimizer(`Adam`) for the training model. The BERT and DistilBERT models are trained with an initial learning rate of (2e-5) which determines the step size for adjusting the weights of the model during training. We use a batch size of 32, which determines the number of samples used in each training iteration, and conduct training for 5 epochs.

For the classification task, we initially employ the SparseCategoricalCrossentropy loss function. However, we consider modifications to the loss function to enhance performance based on our dataset characteristics. One such modification is the weighted cross-entropy loss, where we assign higher weights to the minority class to address class imbalance:

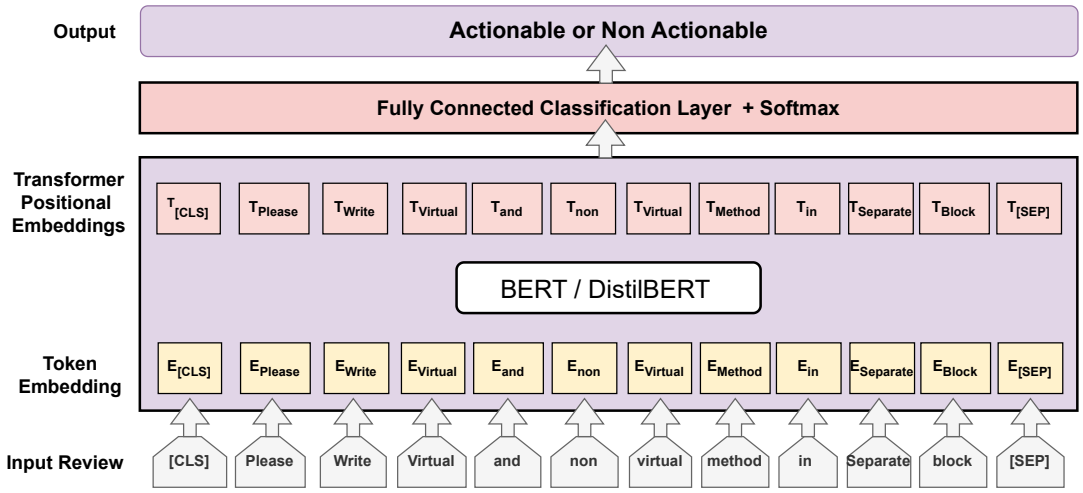

Figure 3.3: Overview of processes used to extract actionable issues from code reviews using transformer-based techniques.

$$Loss = -\sum_{i=1}^{N}\sum_{j=1}^{C} w_j \cdot y_{ij} \log(p_{ij})$$

Here, N represents the total number of instances in the dataset, while C denotes the number of classes. The term w_j is the assigned weight for class j, implemented to address class imbalance. The variable y_{ij} is a binary indicator, set to 1 if class j is the correct classification for instance i, and 0 otherwise. Lastly, p_{ij} is the predicted probability that instance i belongs to class j.

Moreover, we incorporate L2 regularization into the loss function to mitigate overfitting and promote generalizability:

$$Loss = -\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(p_{ij}) + \lambda\sum_{k=1}^{K}\|W_k\|$$

In the equation, λ represents the strength of the regularization parameter, K denotes the number of weight matrices in the model, and $\|W_k\|$ signifies the L2 norm of the kth weight matrix, which is a measure of the magnitude of the weights, contributing to the regularization term in the loss function

By integrating these modifications into the training process, we aim to optimize the performance of our BERT and DistilBERT models. Through experimentation, we fine-tune the weights, regularization parameters, and other adjustments to tailor the loss function according to our specific requirements and challenges.

Then, we use cross-validation to evaluate our model performance. Specifically, we employ 10-fold cross-validation to evaluate the performance of our classifier using the `StratifiedKFold` function of `Scikit-learn` library. In each round of cross-validation, we train our models using 80% of the data and use the remaining 20% of the data for testing.

3.5 Experimental Setup

In this section, we describe the dataset collection and data labeling process that we used to evaluate the performance of our proposed method. We first describe the

datasets that we used in our thesis and then follow the evaluation results.

3.5.1 Empirical Data

We analyzed the availability software of an industry partner through collaborative projects that mainly focus on developing mobile apps and are practicing a rigorous code review process while hosting their code on GitHub repositories. We gathered information on GitHub projects using GitHub Rest API to mine the code review comments. Figure 3.4 shows the process of data extraction from GitHub projects.

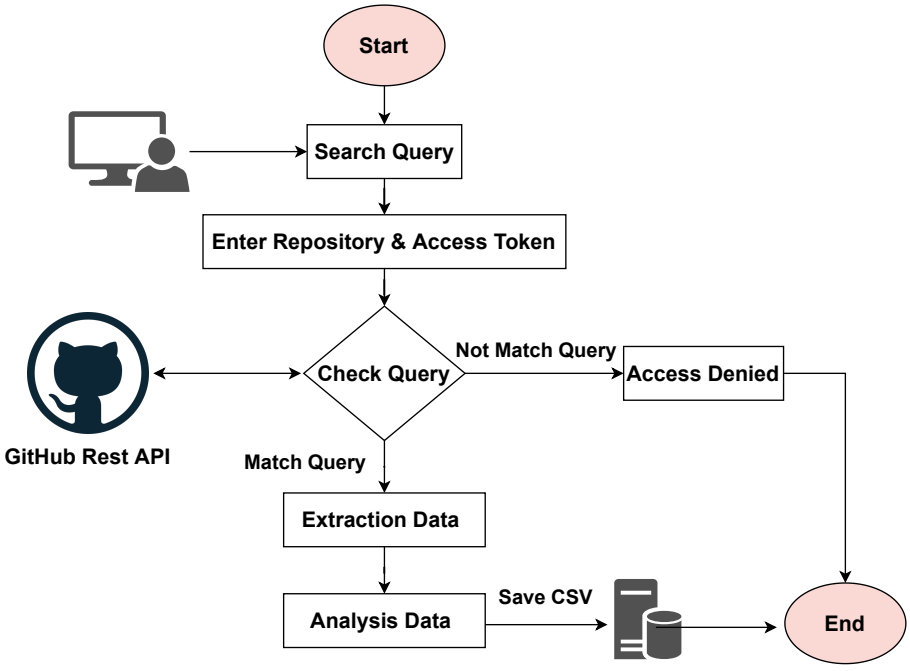

Figure 3.4: Process Flow of gathering data from GitHub Projects

3.5.1.1 Data Gathering

For this thesis, we created a corpus by collecting code reviews from GitHub projects, which consists of a total of 9,500 reviews and 11 columns: Repo Name, Branch Name, Project Name, Commit ID, User Info, Comment, Line Number, File Name, Revision ID, Creation Date, and Last Update. We collected data from five major projects using the GitHub Rest API and mined clean code reviews from raw data using a classifier [52]. These projects are all management tools for mobile devices. Project Alpha began in January 2019 and included 1,929 code reviews. Project Beta was launched in January 2019 and included 2,167 reviews, and Project Gamma has been in development since 2019 and has included 1,126 reviews. The Delta Project has been developed since 2019 and has involved 2,152 reviews. Project Sigma was started in January 2019 and included 2,126 reviews, for all these projects, we collected historical data from the GitHub repository from January 2019 to November 2019, resulting in 11 months of data overall. This includes 9,500 reviews overall and across the five projects.

3.5.1.2 Labelling Code Reviews

As the first step to answer our research questions, we labeled the code reviews. A total of 44 Developers have worked directly on these five projects. We launched a

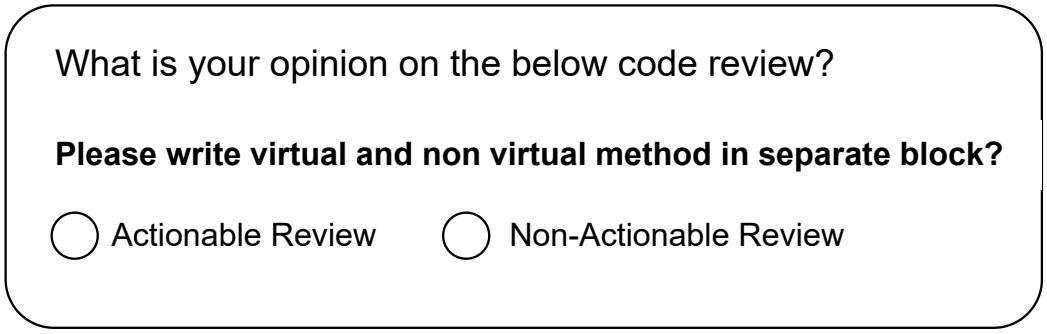

Figure 3.5: A sample of labeling task assigned to the developers.

survey on the team management channel of the team and requested all 44 Developers to participate in labeling the data. Eight developers agreed to participate in labeling the data (response rate of 18%).

We asked developers to classify code reviews into actionable and non-actionable. Figure 3.5 shows an example of the labeling task. We explain the two categories of code reviews below.

Actionable Code Review: This type of code review suggests specific implementable changes or highlights issues needing attention. For example, the code review "Please write virtual and nonvirtual methods in separate blocks" suggests a specific change to the source code that requires developers' attention. In table 3.2 shows several sample actionable code reviews in our corpus.

Non-actionable Code Review: This category encompasses code reviews that do not propose specific actions for developers. For instance, a comment such as

"Got it. It will be fixed in the next patch" merely acknowledges an issue without an immediate call to action. Similarly, a comment like "Actually, I have no idea how to provide attributes for lines" suggests a lack of direction rather than providing a clear step for code revision, placing it in the "non-actionable" classification. These comments primarily focus on understanding the code rather than explicitly prompting changes. Sample non-actionable code reviews from our corpus are illustrated in Table 3.2.

We completed the labeling of 9,500 (actionable 4,313 and non-actionable 5,187) reviews with the help of eight developers working on these projects. In Table 3.3, we summarized the statistics of the three projects dataset.

Table 3.2: The sample labeled review comments of our corpus dataset

Commit ID	**Code Review Comments**	**Labels**
c8108e70fd	Please write virtual and non virtual methods in separate block.	Actionable
468c5b40b4	Please include Debug file of Settings project which is StDebug instead of WDebugBase file.	Actionable
119ae63011	Instead of calling redundant remove observer method, it is better to add logic at the end of this method. if self.suotaBinaryCopyStatus == false {// Remove observer}	Actionable
0eb16a1e91	remove the save method, data will be store with the device info dictionary. Here keep only getSKUcodeValue like func getSKUcodeValue() String? Note: add SKU_code value into getGearInfoDictionary method	Actionable
b122675352	I don't like the formatting of your code.	Non Actionable
b122675434	Got it. I will be fixed in next patch.	Non Actionable
b122675443	change has been applied with new patch	Non Actionable
b122675468	Actually, now, I have no idea how to provide attributes for lines.	Non Actionable

Table 3.3: Overview of Statistics for the 9,500 labeled reviews.

Labels	Alpha	Beta	Gamma	Delta	Sigma
Actionable	29%	30%	11%	15%	15%
Non-Actionable	25%	19%	14%	30%	12%

3.6 Results and Discussions

In this section, we describe the performance of various feature engineering-based and transformer-based models for code review classification. Table 3.4 shows the results of feature engineering-based approaches. We observe that SVM and Random Forest achieved the highest precision (0.93 and 0.94, respectively) and F1 score of 0.95, while Sentence-BERT exhibited excellent recall of 0.98. In contrast, K-Neighbor and Naive Bayes did not achieve good performance.

Table 3.4: The results of feature engineering based approaches

MODEL	**TF-IDF**			**Sentence-BERT**			**USE**			**Mirror-BERT**		
Classifiers	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
SVM	0.93	0.96	0.94	0.93	0.90	0.91	0.93	0.94	0.93	0.94	0.91	0.92
Random Forest	0.94	0.97	**0.95***	0.87	0.98	0.93	0.90	0.97	0.93	0.93	0.96	0.93
Logistic Regression	0.91	0.98	**0.95***	0.94	0.93	0.93	0.89	0.94	0.92	0.94	0.95	0.94
K-neighbour	0.75	0.64	0.69	0.94	0.92	0.93	0.95	0.85	0.90	0.96	0.84	0.90
Naive Bayes	0.93	0.96	0.94	0.89	0.79	0.81	0.88	0.80	0.80	0.85	0.83	0.84

Table 3.5 presents the results of the transformer-based model, where both

Table 3.5: The result of Transformer Based Models

MODEL	**Precision**	**Recall**	**F1 Score**
BERT	0.96	0.97	**0.96***
DistilBERT	0.95	0.96	0.95

BERT and DistilBERT showed impressive precision (0.96 and 0.95, respectively), recall (0.97 and 0.96), and F1 scores (0.96 and 0.95). Our findings suggest that transformer-based models, especially BERT, outperform traditional feature engineering-based supervised learning classifiers on various measures.

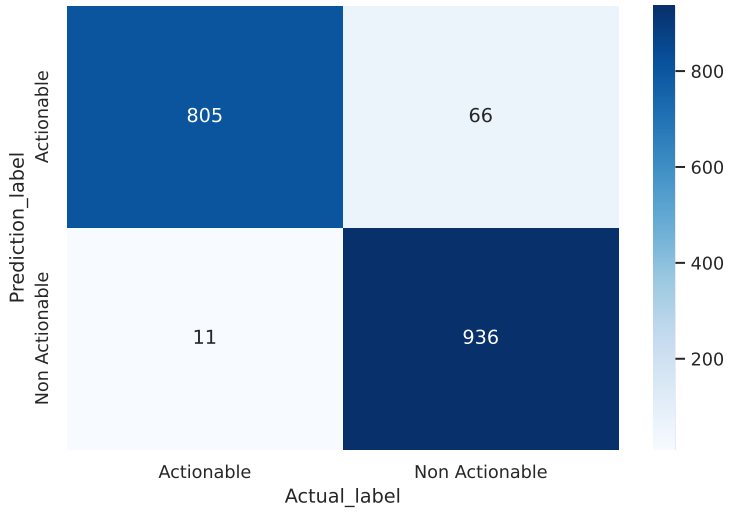

Figure 3.6: The Confusion Matrix to measure classifier performance

We further analyze the errors made by classification models. Figure 3.6 shows

the confusion matrix for the BERT classifier which achieved the best performance in terms of the F1 score. These results indicate a strong predictive performance by the BERT classifier, with a high true positive rate and a low false negative rate for both classes, demonstrating the model's ability to effectively distinguish between 'actionable' and 'non-actionable' categories. This level of performance underscores the robustness of the BERT model in handling classification tasks where contextual nuances of text are crucial for accurate categorization.

> The BERT model was trained using our labeled dataset and demonstrated high accuracy in identifying actionable reviews, achieving an outstanding F-score of 0.96. This level of accuracy emphasizes the model's effectiveness in this particular task.

3.7 Threats to Validity

Although our code review classification approach makes important contributions to the literature, it is essential to acknowledge certain threats to validity. We aim to address these limitations in the future to refine our approach and enhance its broader applicability.

- **Limited Data Size:** Our dataset, consisting of 9,500 code reviews from five private GitHub projects, is relatively small, which raises concerns about

higher F1 scores and may restrict the generalizability of our results to larger datasets.

- **Code Contexts:** While we collected the review comments, we could not have associated code snippets or source code files to them. Due to privacy and proprietary concerns in collecting data from private industry projects, we were unable to provide additional contextual data. Including code contexts could potentially enhance the classification of code reviews.

- **Limited Timeframe:** Our study's temporal scope, spanning from January 2019 to November 2019, might not fully capture the dynamic evolution of code review practices and tool advancements over time.

- **Industry-Specific Focus:** Our dataset focusing on a specific industry partner primarily involved in mobile app development may limit the applicability of our findings to a broader spectrum of software development domains.

- **Biases in Labelling Process:** The labeling process, carried out by a small number of developers (8 out of 44), may introduce potential biases, and the binary classification of code reviews as actionable or non-actionable might oversimplify the intricate nature of such feedback.

3.8 Summary

This Chapter presents several techniques for classifying code reviews to identify the ones that require immediate actions from developers. Through an empirical evaluation in an industrial setting, we analyzed 9,500 code reviews across five distinct projects. We experimented with both traditional supervised classification methods as well as transformer-based models. We noticed that the BERT classifier for code review exhibited superior performance, achieving an F-score of 0.96. In the next Chapter, we will discuss how we can integrate the code review classification to improve software development practices by supporting the developers in addressing the issues raised in the actionable code reviews.

4 Improving Code Quality with RefineCode

In this Chapter, we explore **RQ2**, focusing on how RefineCode can assist developers in addressing coding issues identified in code reviews. Specifically, we examine how the RefineCode application tool utilizes code review classification to support developers in resolving these issues. RefineCode encompasses three primary features as shown in Figure 3.1. Firstly, it recommends similar code reviews from GitHub Projects within an organization, aiding developers in addressing specific issues. Secondly, it analyzes code reviews to automatically extract relevant information from Stack Overflow, providing valuable insights for issue resolution. Thirdly, RefineCode integrates a ChatBot feature, offering potential solutions tailored to a given code review.

In the remainder of the Chapter, we first provide motivation for the adoption of the aforementioned features in RefineCode. Subsequently, We explain the implementation details of these features. Then, we demonstrate the Application system using real-world use case scenarios. Finally, we delve into a discussion on various

potential threats to the validity of the RefineCode system.

4.1 Motivation

Based on our analysis of the code reviews and relevant work in the literature, we identified the required features for RefineCode. In the following, we elucidate the rationale behind incorporating the three key features of RefineCode.

Why similar code reviews from GitHub projects? Siam et al. underscore the imperative for automatic code review recommendations in project development and maintenance, particularly in large companies, as pertinent and useful reviews can significantly alleviate the workload of developers [62]. When developers encounter issues, their initial recourse often involves checking if the team has previously resolved analogous problems. his process is crucial, allowing developers to learn from past solutions and apply insights to their ongoing work. Examining how similar issues were addressed in prior code reviews enables developers to swiftly identify effective strategies. Furthermore, leveraging past reviews not only enhances efficiency but also fosters a deeper understanding of common challenges and solutions within the project, contributing to a more collaborative and knowledge-rich development environment.

Why recommendation and entity Linking with Stack Overflow issues? Software developers frequently use technical question-and-answer platforms like

Stack Overflow to adopt potential solutions or to get some clues on how to resolve the issues pertinent to a code review. However, identifying posts from Stack Overflow that are related to the entities mentioned in a code review can be time-consuming for a developer. A previous study suggests that developers may struggle to find suitable code from Stack Overflow and integrate it into different contexts [69]. To address this challenge, RefineCode aims to reduce developers' efforts by linking entities of a code review to relevant Stack Overflow issues so that developers quickly find solutions tailored to their specific problems.

Why the chatbot for providing potential solutions? Recent advancements in automatic code generation from natural language have shown promise [41]. Large language models (LLMs) have demonstrated their ability to assist developers in code revision for resolving code quality issues [67]. In this context, we implement the chatbot feature with a prompt engineering approach, leveraging LLMs to automatically generate candidate code revision solutions. The RefineCode Chatbot offers an example-driven solution to enhance code quality and assist developers in programming tasks for specific feature implementations. By providing real-time code samples and suggestions based on specific developer actionable reviews and selected prompts, the chatbot empowers developers to learn and apply best practices in coding, ultimately contributing to more efficient and error-free software development.

4.2 RefineCode Implementation

We now provide the implementation details of the three key features of RefineCode.

4.2.1 Retrieving Similar Code Reviews

For retrieving similar code reviews from GitHub projects, we utilize cosine similarity among code reviews. Specifically, we measure the cosine similarity between the feature representations of two sentences. The features are computed based on TF-IDF and N-gram, where TF-IDF measures the importance of words within each review, adjusting for their frequency across all reviews, while N-gram analysis captures context by considering sequences of N words together.

4.2.2 Recommendation and Entity Linking with Stack Overflow

For recommending Stack Overflow discussions related to a code review, we apply an entity linking process that involves chunking and Named Entity Recognition (NER) in NLTK.

Chunking Process: Chunking is the process of segmenting a text into short phrases or word groups that are more meaningful than individual words but less detailed than full sentences. Chunking enables the extraction of key parts of speech (verbs, adverbs, adjectives, and nouns) through regular expression patterns. When

we apply a chunking process to the code review comment "Please write virtual and non-virtual methods in separate blocks", and focus only on verbs, adverbs, adjectives, and nouns, then we identify and group the relevant words as "Virtual methods", "non-virtual methods".

Named Entity Recognition (NER): NER [38] identifies and classifies named entities in text into specific groups such as names of people, organizations, locations, and more. When we applied NER to pick out key terms for code review 'Please write virtual and non-virtual methods in separate blocks', 'Virtual methods' and 'non-virtual methods' are identified as entities, indicating that these terms provide useful information as a category of programming concept.

After applying the chunking and NER, we prepare the search parameters containing the candidate embeddings (e.g., ["Virtual methods", "non-virtual methods"]) and then send them to the Stack Overflow API using the GET request. The API then returns the relevant information from Stack Overflow[1], [2]. Figure 4.1 shows the enhanced code review process with recommendations and entity linking.

[1]Title: C++ Calling pure virtual function from non virtual function in base class

[2]Link: https://stackoverflow.com/questions/49095676/c-calling-pure-virtual-function-from-non-virtual-function-in-base-class

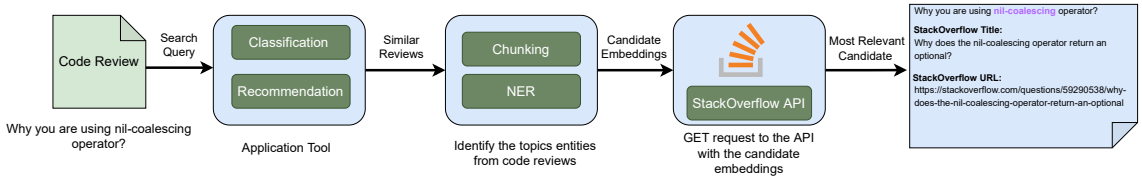

Figure 4.1: Enhanced Code Review Process with Recommendation and Entity Linking.

4.2.3 Example Driven Solutions By RefineCode Chatbot

For implementing the RefineCode chatbot, we used the StableBeluga-7B [64], which is a state-of-the-art LLM designed for generating text-based responses. In this chatbot interface, developers can provide actionable code reviews as input. In addition, they have the flexibility to specify the programming language relevant to their review such as C, C++, Java, Python, etc. The chatbot also allows developers to select specific scenarios or features related to their review comments such as 'API integration', 'User authentication', 'User interface enhancement', etc within the prompt. This feature is particularly useful as it tailors the chatbot's response to the specific context of the developer's current project.

4.3 System Demonstration

In this section, we demonstrate the utility of the three key features of the RefineCode system with illustrative examples.

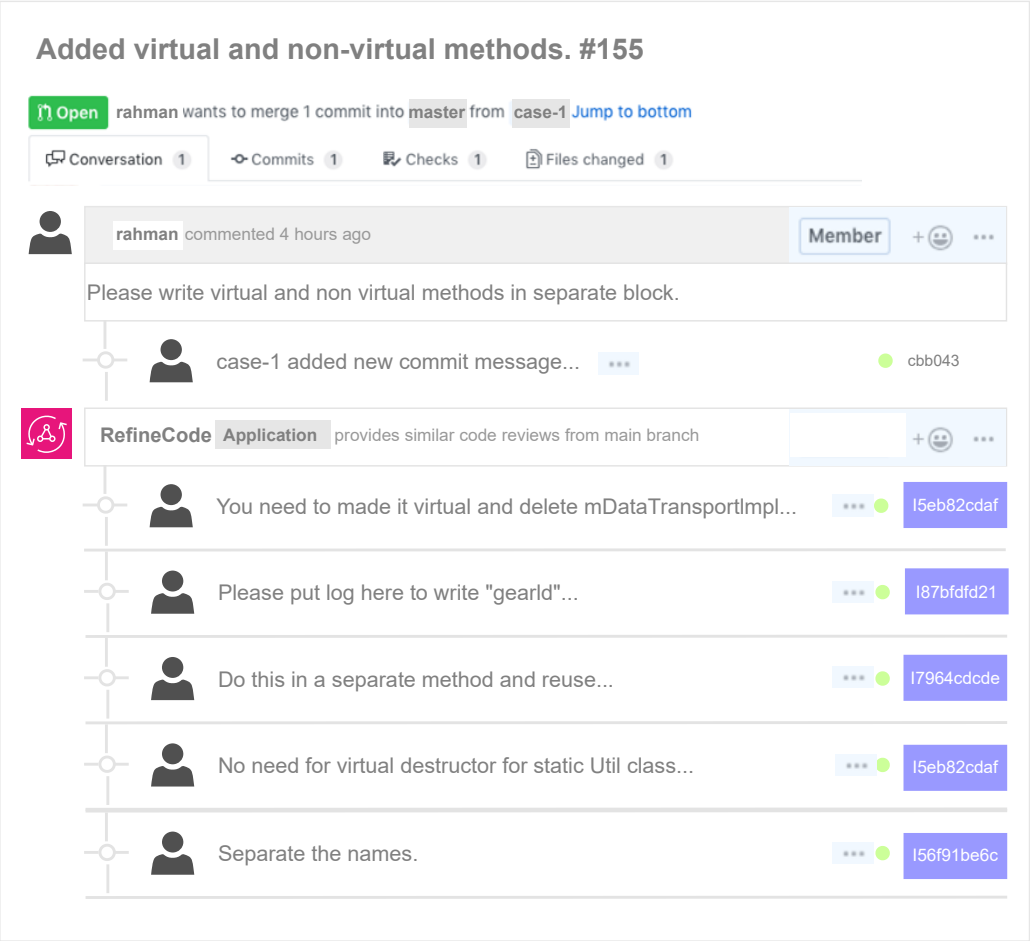

Figure 4.2: An example of providing similar code reviews to solve the specific actionable code review

Code Review Recommendation: Given an actionable code review, RefineCode provides similar code reviews from the GitHub code repository. Figure 4.2 shows an example scenario, where given the code review "Please write virtual and non-virtual methods in separate block.", the system identifies it as an actionable

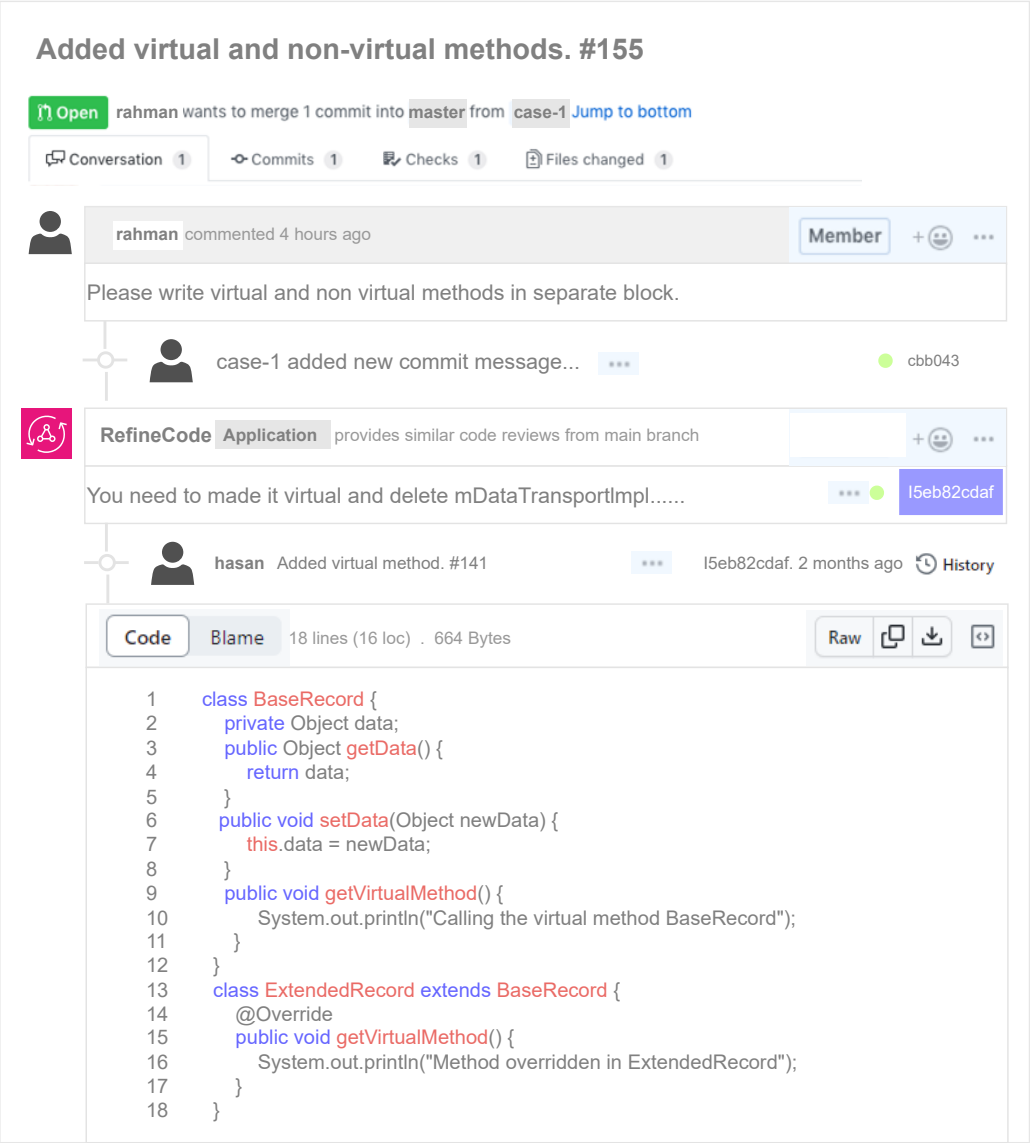

Figure 4.3: Example of providing code review solution commit based on previous similar code review

review and therefore it suggests similar five code reviews which are similar to this review. Then, a developer can open any of these similar code reviews to possibly

get ideas on how to fix this issue. Figure 4.3 shows an example where the developer opens a similar code review to get possible ideas to resolve the issues mentioned in the specific actionable code review.

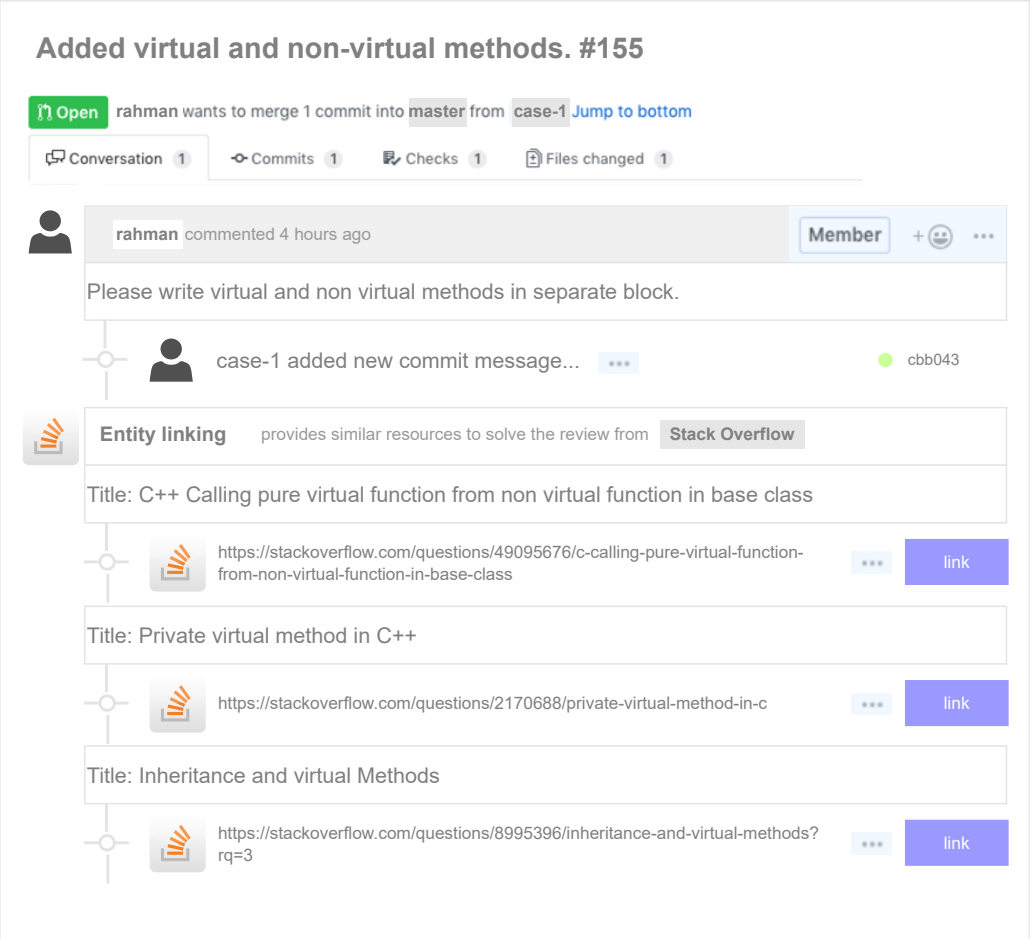

Figure 4.4: An example of providing relevant information from Stack Overflow to solve the specific actionable code review

Entity Linking with Stack Overflow: Similar code reviews from an organi-

zation's GitHub project may not provide specific solutions and developers may look to Stack Overflow to understand their code reviews. Figure 4.4 shows an example of recommending related question-answer posts retrieved from Stack Overflow, given the code review "Please write virtual and non-virtual methods in separate block."

RefineCode Chatbot: The RefineCode chatbot is designed to solve the issues in actionable code reviews for specific problem scenarios. For example, when a developer needs to understand how to implement a new user authentication feature *(Scenario 1)* or integrate a third-party RESTful API for payment processing *(Scenario 2)*, RefineCode can provide specific code snippets and best practice suggestions. This direct and context-sensitive support accelerates the problem-solving process, enabling developers to quickly resolve review comments and increase the quality of code. Figure 4.5 shows a test example provided by a developer request to distinguish between virtual and non-virtual methods in Python with a specific scenario.

> " Scenario 1: Provide a sample code for implementing a user authentication feature using OAuth 2.0 in a React Native mobile app, considering best practices for security and user experience."
>
> — User Authentication

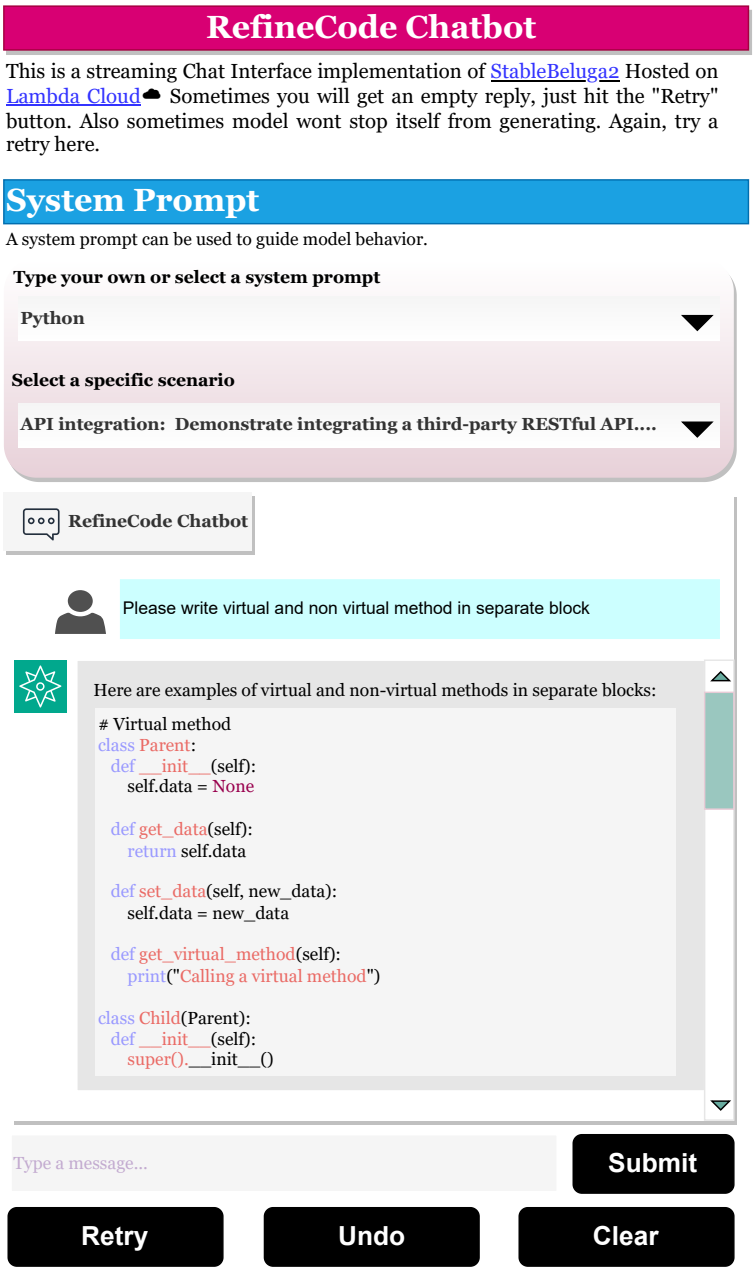

Figure 4.5: Example-driven solution to improve code quality by RefineCode chatbot.

> " Scenario 2: Demonstrate integrating a third-party RESTful API for payment processing in an e-commerce mobile app, focusing on secure transactions and error handling."
>
> — API Integration

4.4 Threats to Validity

Although our study makes valuable contributions to the field, it is essential to acknowledge certain threats to validity. Addressing these limitations will be paramount in future work to refine our approach and enhance its broader applicability.

- **Code review recommendation technique:** We utilized cosine similarity among vector representations of code reviews to recommend GitHub code reviews. However, employing more advanced similarity measures could potentially enhance the relevance and diversity of search results. Additionally, the current focus on code review texts as input may be expanded to include relevant code snippets, further improving the algorithm.

- **User study:** Our preliminary evaluation of the RefineCode application involved system demonstrations, providing anecdotal evidence of the efficacy of its key features. Nevertheless, we recognize the need for more comprehensive quantitative and qualitative evaluations. Specifically, assessing the usefulness

of recommended code reviews and GitHub posts and retrieved posts from Stack Overflow in addressing review comments requires further investigation. Conducting user studies can offer deeper insights into how RefineCode supports developers in real-world settings. In the future, it would be useful to run longitudinal studies to understand how these features from RefineCode may support developers in large industry projects.

4.5 Summary

This chapter presents RefineCode, an application that incorporates code review classification to facilitate developers in refining the code. We present the key features of the system and demonstrate the utility of the system with illustrative use case scenarios. Overall, this work presents the potential of combining machine learning techniques and external resources to support developers in their coding tasks and foster collaboration within software development teams. In the next chapter, we will revisit the key contributions of this thesis along with an overview of future work.

5 Conclusions and Future Work

In this concluding chapter, we first reflect on our main findings and insights and then outline our perspectives and directions for further research.

5.1 Conclusion

In this thesis, we utilize feature engineering-based and transformer-based model architecture to address the two types of research challenging tasks: (i) improve code quality and (ii) Assist developers in solving issues.

The rapidly evolving landscape of software development, characterized by collaboration and communication, highlights the absolute importance of code review. The main challenges of the code review process, mainly stemming from the ambiguity of feedback and communication barriers, demand a sophisticated solution.

For improving code quality, we propose two main approaches. First, we propose a new binary classification task that categorizes the code reviews into either actionable or non-actionable reviews. Second, we introduced the RefineCode ap-

plication tool that utilizes the actionable reviews to find similar code reviews from the GitHub projects within an organization as well as external solutions from Stack Overflow and LLM-driven chatbot. Our empirical evaluation demonstrates the high accuracy of the transformer-based classification on a dataset comprising 9,500 code reviews from five private projects in an industrial setting. The system demonstration, with illustrative examples, also suggests that RefineCode system may help developers in facilitating refining their code.

5.2 Future Work

In future work, we aim to expand our research by incorporating larger and more complex datasets, providing a greater challenge to our models. Additionally, we plan to expand the scope of our dataset beyond mobile app development, enhance the granularity of our labeling process, and explore more advanced recommendation algorithms beyond cosine similarity. We will also emphasize experimenting with more advanced techniques such as Code Llama [54], a state-of-the-art large language model for coding. Then, we will prioritize the ongoing training of our models to keep pace with modern software methodologies. Our focus will be squarely on enhancing the user experience, analyzing broader code contexts for more pertinent feedback, facilitating collaborative code reviews, and advancing the capabilities of the Stable Beluga (Llama2) chatbot. we will integrate the GPT-4 model into our

RefineCode chatbot to yield improved responses.

Furthermore, we plan to fine-tune pre-trained models like Stable Beluga (Llama2) and GPT-4 on code review data, tailoring them to deliver precise solutions for actionable code reviews. This customization will enable the models to generate code that aligns closely with a developer's style, as they will learn from an organization's specific code repositories. This approach promises to provide developers with contextually relevant and organization-specific coding solutions. Additionally, we will utilize various new transformer-based models [60], [14], [22], [21], [25], [15], [49] and Machine Learning [48], [44], [72] , [3] to investigate their effectiveness in identifying reviews. Ultimately, our overarching goal is to develop a system that not only addresses the current needs of developers in resolving concerns expressed in code reviews but also promotes teamwork and enhances the overall quality of software development processes.

Bibliography

[1] A Ahmad, MT Alshurideh, and BH Al Kurdi. The four streams of decision making approaches: Brief summary and discussion. In *International Conference on Advanced Machine Learning Technologies and Applications*, pages 570–580. Springer, 2021.

[2] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[3] Zahra Ashktorab, Benjamin Hoover, Mayank Agarwal, Casey Dugan, Werner Geyer, Hao Bang Yang, and Mikhail Yurochkin. Fairness evaluation in text classification: Machine learning practitioner perspectives of individual and group fairness. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–20, 2023.

[4] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

[5] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 931–940. IEEE, 2013.

[6] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 134–144. IEEE, 2015.

[7] S Bird, E Loper, and E Klein. Natural language processing with python. oreilly media inc., sebastopol, usa, 2009.

[8] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 146–156. IEEE, 2015.

[9] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[10] Peter F Brown, Vincent J Della Pietra, Peter V Desouza, Jennifer C Lai, and Robert L Mercer. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–480, 1992.

[11] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.

[12] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20:273–297, 1995.

[13] Atacílio Cunha, Tayana Conte, and Bruno Gadelha. Code review is just reviewing code? a qualitative study with practitioners in industry. In *Brazilian Symposium on Software Engineering*, pages 269–274, 2021.

[14] Washington Cunha, Celso França, Guilherme Fonseca, Leonardo Rocha, and Marcos André Gonçalves. An effective, efficient, and scalable confidence-based instance selection framework for transformer-based text classification. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 665–674, 2023.

[15] Xiang Dai, Ilias Chalkidis, Sune Darkner, and Desmond Elliott. Revisiting transformer-based models for long document classification. *arXiv preprint arXiv:2204.06683*, 2022.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL `http://arxiv.org/abs/1810.04805`.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186, 2019.

[18] Michael Dorin, Trang Le, Rajkumar Kolakaluri, and Sergio Montenegro. Using machine learning image recognition for code reviews.

[19] 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. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[20] Enrico Fregnan, Fernando Petrulio, Linda Di Geronimo, and Alberto Bacchelli. What happens in my code reviews? an investigation on automatically classifying review changes. *Empirical Software Engineering*, 27(4):89, 2022.

[21] Prakhar Ganesh, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Hassan Sajjad, Preslav Nakov, Deming Chen, and Marianne Winslett. Compressing large-scale transformer-based models: A case study on bert. *Transactions of the Association for Computational Linguistics*, 9:1061–1080, 2021.

[22] Anthony Gillioz, Jacky Casas, Elena Mugellini, and Omar Abou Khaled. Overview of the transformer-based models for nlp tasks. In *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*, pages 179–183. IEEE, 2020.

[23] 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.

[24] Anshul Gupta and Neel Sundaresan. Intelligent code reviews using deep learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*, 2018.

[25] Chadi Helwe, Chloé Clavel, and Fabian Suchanek. Reasoning with transformer-based models: Deep learning, but shallow reasoning. In *International Conference on Automated Knowledge Base Construction (AKBC)*, 2021.

[26] Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Aldeida Aleti. Commentfinder: a simpler, faster, more accurate code review comments recommendation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 507–519, 2022.

[27] David W Hosmer, Stanley Lemeshow, and E Cook. Applied logistic regression 2nd edition. *New York: Jhon Wiley and Sons Inc*, 2000.

[28] Ashraf M Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. Multinomial naive bayes for text categorization revisited. In *AI 2004: Advances in Artificial Intelligence: 17th Australian Joint Conference on Artificial Intelligence, Cairns, Australia, December 4-6, 2004. Proceedings 17*, pages 488–499. Springer, 2005.

[29] Harsh Lal and Gaurav Pahwa. Code review analysis of software system using machine learning techniques. In *2017 11th International Conference on Intelligent Systems and Control (ISCO)*, pages 8–13. IEEE, 2017.

[30] Triet HM Le, Hao Chen, and Muhammad Ali Babar. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)*, 53(3):1–38, 2020.

[31] Heng-Yi Li, Shu-Ting Shi, Ferdian Thung, Xuan Huo, Bowen Xu, Ming Li, and David Lo. Deepreview: automatic code review using deep multi-instance learning. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 318–330. Springer, 2019.

[32] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. Auger: automatically generating review comments with pre-training models. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1009–1021, 2022.

[33] Zhixing Li, Yue Yu, Gang Yin, Tao Wang, Qiang Fan, and Huaimin Wang. Automatic classification of review comments in pull-based development model. In *SEKE*, pages 572–577, 2017.

[34] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1035–1047, 2022.

[35] Hong Yi Lin and Patanamon Thongtanunam. Towards automated code reviews: Does learning code structure help? In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 703–707. IEEE, 2023.

[36] Fangyu Liu, Ivan Vulić, Anna Korhonen, and Nigel Collier. Fast, effective, and self-supervised: Transforming masked language models into universal lexical and sentence encoders. *arXiv preprint arXiv:2104.08027*, 2021.

[37] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270. IEEE, 2015.

[38] Alireza Mansouri, Lilly Suriani Affendey, and Ali Mamat. Named entity recognition approaches. *International Journal of Computer Science and Network Security*, 8(2):339–344, 2008.

[39] Andrew McCallum, Kamal Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Madison, WI, 1998.

[40] Stuart Millar, Denis Podgurskii, Dan Kuykendall, Jesús Martínez del Rincón, and Paul Miller. Optimising vulnerability triage in dast with deep learning. In *Proceedings of the 15th ACM Workshop on Artificial Intelligence and Security*, pages 137–147, 2022.

[41] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR, 2023.

[42] Miroslaw Ochodek, Regina Hebig, Wilhelm Meding, Gert Frost, and Miroslaw Staron. Chapter 8 recognizing lines of code violating company-specific coding guidelines using machine learning. In *Accelerating Digital Transformation: 10 Years of Software Center*, pages 211–251. Springer, 2022.

[43] Ipek Ozkaya. The next frontier in software development: Ai-augmented software development processes. *IEEE Software*, 40(4):4–9, 2023.

[44] Ashokkumar Palanivinayagam, Claude Ziad El-Bayeh, and Robertas Damaševičius. Twenty years of machine-learning-based text classification: A systematic review. *Algorithms*, 16(5):236, 2023.

[45] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–27, 2018.

[46] Rishov Paul, Md Mohib Hossain, Masum Hasan, and Anindya Iqbal. Automated program repair based on code review: How do pre-trained transformer models perform? *arXiv preprint arXiv:2304.07840*, 2023.

[47] Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[48] Miftahul Qorib, Timothy Oladunni, Max Denis, Esther Ososanya, and Paul Cotae. Covid-19 vaccine hesitancy: Text mining, sentiment analysis and machine learning on covid-19 vaccination twitter dataset. *Expert Systems with Applications*, 212:118715, 2023.

[49] Abir Rahali and Moulay A Akhloufi. End-to-end transformer-based models in textual-based nlp. *AI*, 4(1):54–110, 2023.

[50] Mohammad Masudur Rahman, Chanchal K Roy, and Raula G Kula. Predicting usefulness of code review comments using textual features and developer experience. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 215–226. IEEE, 2017.

[51] Shadikur Rahman, Umme Ayman Koana, Hasibul Karim Shanto, Mahmuda Akter, Chitra Roy, and Aras M Ismael. Measuring the effectiveness of code review comments in github repositories: A machine learning approach. In *Machine Learning and Data Mining in Pattern Recognition*, volume 16, pages 35–48. ibai-publishing, 2020.

[52] Shadikur Rahman, Umme Ayman Koana, and Maleknaz Nayebi. Example driven code review explanation. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 307–312, 2022.

[53] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[54] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[55] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190, 2018.

[56] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108, 2019.

[57] Jaydeb Sarker, Asif Kamal Turzo, Ming Dong, and Amiangshu Bosu. Automated identification of toxic code reviews using toxicr. *ACM Transactions on Software Engineering and Methodology*, 2023.

[58] Oussama Ben Sghaier and Houari Sahraoui. A multi-step learning approach to assist code review. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 450–460. IEEE, 2023.

[59] Shu-Ting Shi, Ming Li, David Lo, Ferdian Thung, and Xuan Huo. Automatic code review by learning the revision of source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4910–4917, 2019.

[60] Yu Shi, Xi Zhang, and Ning Yu. Pl-transformer: a pos-aware and layer ensemble transformer for text classification. *Neural Computing and Applications*, 35 (2):1971–1982, 2023.

[61] Rodrigo FG Silva, Chanchal K Roy, Mohammad Masudur Rahman, Kevin A Schneider, Klerisson Paixao, and Marcelo de Almeida Maia. Recommending comprehensive solutions for programming tasks by mining crowd knowledge. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 358–368. IEEE, 2019.

[62] Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. Core: Automating review recommendation for code changes. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 284–295. IEEE, 2020.

[63] Miroslaw Staron, Miroslaw Ochodek, Wilhelm Meding, Ola Söder, and Emil Rosenberg. Machine learning to support code reviews in continuous integration. *Artificial Intelligence Methods For Software Engineering*, pages 141–167, 2021.

[64] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor

Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[65] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2291–2302, 2022.

[66] Anderson Uchôa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley KG Assunçao, Silvia Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and Alessandro Garcia. Predicting design impactful changes in modern code review: A large-scale empirical study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 471–482. IEEE, 2021.

[67] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. Frustrated with code quality issues? llms can help! September 2023.

[68] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021.

[69] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. How do developers utilize source code from stack overflow? *Empirical Software Engineering*, 24:637–673, 2019.

[70] Kareshna Zamani, Didar Zowghi, and Chetan Arora. Machine learning in requirements engineering: A mapping study. In *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*, pages 116–125. IEEE, 2021.

[71] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, 42(6):530–543, 2015.

[72] Dell Zhang, Murat Sensoy, Masoud Makrehchi, Bilyana Taneva-Popova, Lin Gui, and Yulan He. Uncertainty quantification for text classification. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3426–3429, 2023.

[73] Ting Zhang, Bowen Xu, Ferdian Thung, Stefanus Agus Haryono, David Lo, and Lingxiao Jiang. Sentiment analysis for software engineering: How far can pre-trained transformer models go? In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 70–80. IEEE, 2020.

[74] Liping Zhao, Waad Alhoshan, Alessio Ferrari, Keletso J Letsholo, Muideen A Ajagbe, Erol-Valeriu Chioasca, and Riza T Batista-Navarro. Natural language processing for requirements engineering: A systematic mapping study. *ACM Computing Surveys (CSUR)*, 54(3):1–41, 2021.