IMPROVING MODERN CODE REVIEW LEVERAGING CONTEXTUAL AND STRUCTURAL INFORMATION FROM SOURCE CODE

by

Ohiduzzaman Shuvo

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

at

Dalhousie University
Halifax, Nova Scotia
August 2023

I dedicate this thesis to my mother Mrs. Rozina Begum, and my father Jahangir Fakir, whose inspirations help me to accomplish every step of my life.

Contents

List of Tables

List of Figures

Abstract

Review comments are a major building block of modern code reviews. Ensuring the quality of code review comments is essential, but manually writing high-quality review comments is technically challenging and time-consuming. Over the years, there have been numerous attempts to automatically assess and recommend code review comments, but they could be limited in several aspects. First, according to existing evidence, various development practices including code reviews could be drastically different between open and closed-source systems. However, only a little research has been done to better understand how existing techniques might perform differently when assessing the code reviews from open and closed-source systems. Second, existing techniques that recommend or generate code review comments often suffer from a lack of scalability (e.g., requirements of specialized hardware by Deep Learning models) and generalizability (e.g., use of only one programming language).

In this thesis, we (a) conduct an empirical study to better understand the challenges of existing techniques for code review assessment and (b) propose a novel, scalable technique for review comment recommendation. First, we empirically investigate how existing techniques perform in assessing code reviews from open-source and closed-source systems. We find that the performance of existing techniques significantly differs when assessing code reviews from these two types of systems. Our findings also suggest that less experienced developers submit more non-useful review comments to both systems, which warrants for automated support in writing code reviews. Second, to help developers write better review comments, we propose a novel technique – RevCom – that recommends relevant review comments by leveraging various code-level changes with structured information retrieval. Our technique outperforms both IR-based and DL-based baselines while being lightweight, scalable and has the potential to reduce the cognitive effort and time of the reviewers.

List of Abbreviations Used

BLEU Bilingual Evaluation Understudy.
BoW Bag of Words.
BPE Byte-Pair Encoding.

CBOW Continuous Bag of Words.
CRA Code Review Analytics.

DL Deep Learning.

GPM Gestalt Pattern Matching.

IR Information Retrieval.

LCS Longest Common Substring.
LR Logistic Regression.
LSTM Long Short-term Memory.

MCR Modern Code Review.

NB Naive Bayes.
NLP Natural Language Processing.
NMT Neural Machine Translation.

OOV Out-of-vocabulary.

PRs Pull Requests.

RF Random Forest.

SE Software Engineering.
SMOTE Synthetic Minority Oversampling Technique.
SRBD Samsung Research Bangladesh.
SVM Support Vector Machine.

T5 Text-To-Text Transfer Transformer.

TF-IDF Term Frequency - Inverse Document Frequency.

VSM Vector Space Model.

Acknowledgements

First, I thank the Almighty, the most gracious and the most merciful, who granted me the capability to carry out this work. Then I would like to express my heartiest gratitude to my supervisor Dr. Masudur Rahman for his constant guidance, advice, encouragement and extraordinary patience during this thesis work. Without his constant support, this work would have been impossible.

I would also like to express my sincere gratitude to Dr. Tushar Sharma and Dr. Evangelos E. Milios for their invaluable advisement and meticulous evaluation of my thesis. Their scholarly perspectives and thoughtful input have elevated the quality and rigor of my research, and I am truly appreciative of their time and expertise.

Thanks to all of the members of the Intelligent Automation in Software EngineeRing (RAISE) Lab, with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Parvez Mahbub, Sigma Jahan, Asif Sameer, Usmi Mukherjee and Mehil B Shah.

I am grateful to Dalhousie University and its Computer Science Department for their generous financial support through scholarships, awards and bursaries that helped me to concentrate more deeply on my thesis work. In particular, I would like to thank Dr. Michael McAllister and Megan Baker.

I would like to express my deepest gratitude to my parents, Jahangir Fakir and Rozina Begum, for their unwavering support and love throughout my academic journey. Their guidance, encouragement, and sacrifices have been instrumental in shaping the person I am today and achieving this significant milestone in my life.

I would also like to thank my younger brother – Sayful Islam, and younger sister – Jannatul Akter Labonno, who have always inspired me in every stage of my life.

Lastly, I extend my heartfelt thanks to all those who have supported me in various ways throughout this research endeavor. Your encouragement, advice, and assistance have been invaluable, and I am deeply grateful for your presence in my life.

Chapter 1

Introduction

1.1 Motivation

Peer code review is an engineering practice where software developers submit changed code to peers (a.k.a., reviewers). The reviewers then check if the code is suitable for integration into the main code base and recommend useful changes. Besides finding fine-grained defects (e.g., logical errors), code review helps improve the readability [13, 82, 93], maintainability [82], and design quality [63] of source code.

Recently, a lightweight review process, namely Modern Code Review (MCR), often assisted by specialized software tools [11], has gained significant popularity in both industry and open-source development. In MCR, change suggestions are made by reviewers in the form of *review comments*, one of the main building blocks of code reviews [15]. These comments are generally discussed by the developers and the reviewers before making any corrections to the code and then submitting the next version of the changed code. Hence, ensuring the *quality* of review comments is essential in code reviews. However, manually writing high-quality code review comments could be technically challenging and time-consuming, which warrants for automated tools and techniques.

Past studies [22, 15, 44, 76, 34, 51, 88] use automated tools and techniques to assess or recommend code review comments. Unfortunately, they could be limited in several aspects. First, according to Khanjani and Sulaiman [42], various development practices of open-source systems (e.g., code review, software design, documentation) could be drastically different from that of closed-source systems. However, to the best of our knowledge, there exists no study that investigates how existing tools and techniques differ when assessing the code reviews from open-source and closed-source systems. An investigation using both types of software systems is essential to comprehensively understand the behaviours of automated tools and techniques towards these systems. It not only can deliver actionable insights for stakeholders (e.g., developers,

reviewers) but also can reveal potential gaps in the literature. Second, recommended reviews can help a reviewer write better review comments with reduced effort and a code submitter by reducing the wait time [34]. However, existing techniques that recommend or generate code review comments suffer from a lack of scalability and generalizability. For instance, existing DL-based techniques require specialized computing resources and long training time, which could be costly. On the other hand, the IR-based technique [34] recommends code review comments only for method-level changes, which makes it unsuitable for code changes outside of a method body.

1.2 Problem Statement

Past studies [22, 15, 44, 76] use automated tools and techniques to analyze several quality aspects of code review comments such as *sentiments* or *usefulness*. El Asri et al. [22] detect sentiments in review comments and show that the reviewer's sentiments could affect a code submitter's perception about the reviews. However, they only use open-source subject systems for their analysis. On the other hand, Rahman et al. [76] and Hasan et al. [32] use several textual and historical properties of code reviews to classify the useful and non-useful code review comments where they use only closed-source software systems. Thus, existing studies use either open or closed-source subject systems to investigate the quality aspects of review comments. However, according to Khanjani and Sulaiman [42], various development practices including code reviews could be drastically different between open and closed-source systems. In other words, open-source and closed-source software systems could be different in terms of the sentiments and usefulness of their review comments. Such differences might influence the performance of automated tools and techniques assessing the code reviews. However, to the best of our knowledge, there exists no work that investigates how existing tools and techniques can differ when assessing review quality aspects – sentiments and usefulness – from both open and closed-source systems. Such a comparison not only can deliver actionable insights for stakeholders (e.g., developers and reviewers) but also can encourage better tool supports for modern code reviews.

Although several approaches exist to automatically assess different quality aspects of code review comments, manually writing high-quality review comments is significantly challenging. To address this challenge, several existing approaches [29, 88, 98,

50] recommend or generate code review comments using Deep Learning (DL) techniques. Earlier works [29, 88] use Long Short-term Memory (LSTM) networks with an attention mechanism [8] to recommend code review comments. Later approaches [98, 50] employ more sophisticated architecture such as Text-To-Text Transfer Transformer (T5) [101] to generate code review comments. However, they require specialized computing resources (e.g., 16 × 40GB GPU [101]), which could hurt their scalability. They also require long training time (e.g., 12 days [101]), which could be costly.

Recent studies [26, 33, 54, 60] suggest that simpler approaches, such as Information Retrieval (IR), can perform better than complex deep learning models with less computational time and resources. Hong et al. [34] propose an IR-based approach that leverages method-level similarity in recommending code review comments. Although their approach outperforms deep learning models, it could be limited in several aspects. First, they use the Bag of Words (BoW) model [72] that represents source code as token vectors ignoring the code structures and semantics. Source code contains both structured (e.g., methods, library information) and unstructured items (e.g., code comments). Second, they report their findings for only Java-based systems, which might not generalize to other programming languages (e.g., C++, Python). Finally, their approach to recommending review comments was evaluated only using method-level information in the source code. However, method bodies might not cover all the changes that require code reviews. Thus, the existing approaches might not perform well in recommending reviews for the code changes outside of a method body (see listing 4.1, 4.2). According to Li et al. [50], structured information such as *diff* contains all types of changes in the source code and thus can help better understand the semantics of any code changes. However, the work of Hong et al. [34] overlooks this structured information and recommends code reviews only for the changes in the method.

1.3 Our Contribution

In this thesis, we conduct two separate but complementary studies to address the above gaps from the literature as follows.

In the first study, we conduct an empirical investigation with seven automated tools and techniques and contrast their performance when assessing the quality of

2,183 code review comments from open and closed-source systems. First, we detect sentiments in the review comments using five sentiment detection approaches – Stanford Core NLP [57], SentiStrength [39], Senti4SD [16], SentiCR [3], and SentiMoji [19]. We then contrast the performance of these tools and techniques between open-source and closed-source systems. Second, we determine the usefulness of code review comments using two usefulness classification techniques – RevHelper [76] and CRA-model [32]. We then contrast the performance of these techniques between open and closed-source systems. Finally, we investigate the role of sentiments in classifying the useful and non-useful code review comments from two types of systems.

We evaluate the performance of existing tools and techniques in terms of accuracy, precision, recall, and F1-score and report several findings. First, the performance of automated tools and techniques can vary from 1.12% to 14.67% between open and closed-source systems when detecting sentiments in the review comments. Second, the performance of review usefulness classification techniques can vary from 13.33% to 18.42% between open and closed-source systems, which is significant. Finally, the sentiments can improve the usefulness classification of review comments by 2.94% for open and 1.67% for closed source systems. Our findings also suggest that less experienced developers submit more non-useful review comments in both open-source and closed-source systems, which warrants for automated support in writing code review comments.

To help developers write better review comments, in the second study, we propose a novel technique, namely – RevCom – that recommends relevant review comments leveraging various code-level changes captured with structured information retrieval. Our work is inspired by the work of Hong et al. [34] that relies on method-level similarity and the Bag of Words (BoW) to recommend code review comments. However, unlike their work, RevCom leverages the structured information from all types of code changes and thus can recommend code reviews for both method-level and non-method-level changes.

We evaluate our proposed approach with $\approx 56K$ changed code and comment pairs from eight projects (four Python + four Java). We use three different metrics – BLEU score [69], perfect prediction, and semantic similarity [31] to evaluate the performance of RevCom. We find that RevCom can recommend review comments with an average

BLEU score of ≈ 26.63%. According to Google's AutoML Translation documentation[1], such a BLEU score indicates that the review comments can capture the original intent of the reviewers with some grammatical errors. We also find that structured information plays a significant role in our approach. Furthermore, a comparison with two state-of-the-art techniques – CommentFinder [34] and CodeReviewer [50] show that RevCom outperforms them in all three metrics.

1.4 Related Publications

Several parts of this thesis have been submitted and accepted at different conferences. We provide the list of publications here. In each of these papers, I am the primary author, and all the studies were conducted by me under the supervision of Dr. Masud Rahman. While I wrote these papers, the co-authors took part in advising, editing, and reviewing the papers.

- *Ohiduzzaman Shuvo*, Parvez Mahbub, and M. Masudur Rahman. *Recommending Code Reviews Leveraging Code Changes with Structured Information Retrieval*. In Proceeding of The 39th IEEE/ACM International Conference on Software Maintenance and Evolution (ICSME 2023), pp.13, Bogota, Colombia, October 2023 (In press).

Apart from the aforementioned paper, our another paper is ready to be submitted to a major software engineering conference.

- *Ohiduzzaman Shuvo*, Parvez Mahbub, and M. Masudur Rahman. *Mind the Gap: Contrasting the Quality of Comments in Code Reviews between Open and Closed-Source Systems*. In Proceeding of The 31st IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2024), pp. 12, Rovaniemi, Finland, March 2024 (to be submitted).

Finally, the idea of using structural and contextual information from source code inspired other research in major software engineering conferences, where I am the second author.

[1] https://bit.ly/3wGpCIx

- Parvez Mahbub, *Ohiduzzaman Shuvo*, and M. Masudur Rahman. *Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation.* In Proceeding of The 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023), pp. 13, Melbourne, Australia, May 2023.

- Parvez Mahbub, *Ohiduzzaman Shuvo*, and M. Masudur Rahman. *Defectors: A Large, Diverse Python Dataset for Defect Prediction.* In Proceeding of The 20th International Conference on Mining Software Repositories (MSR 2023), pp. 5, Melbourne, Australia, May 2023.

1.5 Outline of the Thesis

The thesis contains five chapters in total. To assess and recommend code review comments, we conduct two independent but interrelated studies, and this section outlines different chapters of the thesis.

- Chapter 2 discusses several background concepts (e.g., modern code review, word embeddings) that will be required to follow the rest of the thesis.

- Chapter 3 discusses our first study, How do Automated Tools and Techniques Differ between Open and Closed Source Systems in Assessing their Code Review Quality? An Empirical Study.

- Chapter 4 discusses our second study that proposes *RevCom*, which recommends relevant review comments leveraging various code-level changes with structured information retrieval.

- Chapter 5 concludes the thesis with a list of directions for future works.

Chapter 2

Background

In this chapter, we introduce the required terminologies and concepts to follow the remainder of this thesis. Section 2.1 introduces the concept of modern code review. Section 2.2 discusses the sentiments in the code review comments. Section 2.3 discusses the usefulness of review comments and differentiates between useful and non-useful review comments using corresponding examples. Section 2.4 discusses the formulation of review recommendation as an information retrieval problem. Section 2.5 describes embedding – a process of translating high-dimensional numerical data into low-dimensional semantic representations. Finally, Section 2.6 summarizes this chapter.

2.1 Modern Code Review

In Modern Code Review (MCR), a code author submits a changed code to implement new features or fix bugs in the old version. Let us denote the original and updated codes as C_0 and C_1. Once the changed code ($D : C_0 \rightarrow C_1$) is ready for review, the author creates a pull request with the code review tool (e.g., GitHub) and invites peers (a.k.a., reviewers) for the review. Then, the reviewers inspect the changed code and provide feedback (i.e., review comments) on the specific parts of it. Based on these comments, the author submits a new version of the changed code C_2. Note that the review process is not finished yet. The reviewers can further provide feedback as the code review comments on the changed code C_2, and the authors might revise the code again. This process repeats until the submitted code C_n has sufficient quality to be integrated into the code repository. However, manually writing the review comments could require significant time and cognitive effort from a reviewer. Our work (Chapter 4) automatically recommends relevant code review comments, which could reduce the time and effort required for modern code review.

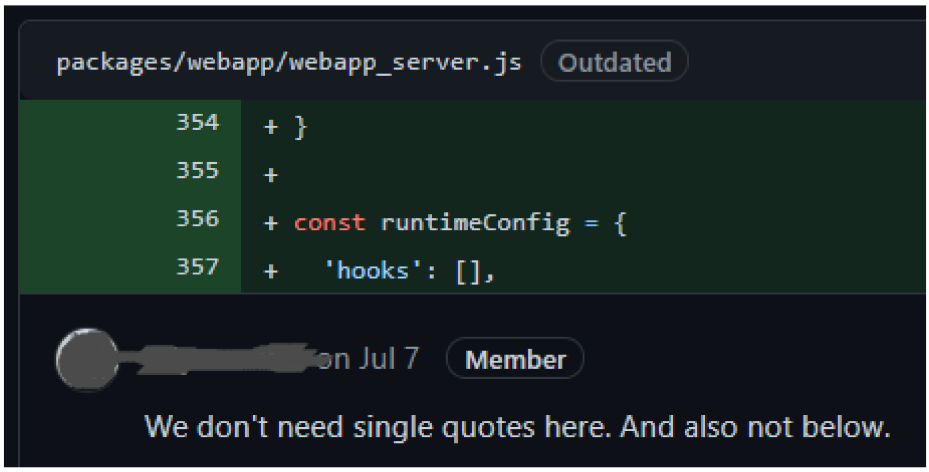
2.2 Sentiments in Review Comments

El Asri et al. [22] show that beyond technical information, code review comments could contain reviewers' sentiments, which could affect a code submitter's perception about the reviews. For example, negative sentiments in the review comments could delay the review acceptance by 1.32 days on average [22], which is a significant delay. In the same vein, Kononenko et al. [44] suggests that *well-done* code reviews warrant clear, constructive, and timely feedback from a domain expert who has strong interpersonal skills (e.g., encouraging, supportive, optimistic). Manually detecting sentiments in code review comments could be time-consuming and error-prone. Thus, recently, many tools and techniques (e.g., SentiCR [3], SentiMoji [19]) attempt to automatically detect the sentiments in code review comments. However, most of the existing works [3, 22, 19] investigate sentiments only in the open-source code reviews. According to the existing evidence [42], various aspects of software development, including code reviews could be drastically different between open-source and closed-source systems. We thus analyze the sentiments in the code review comments from both types of systems, which might provide useful insights for better tools support in modern code reviews.

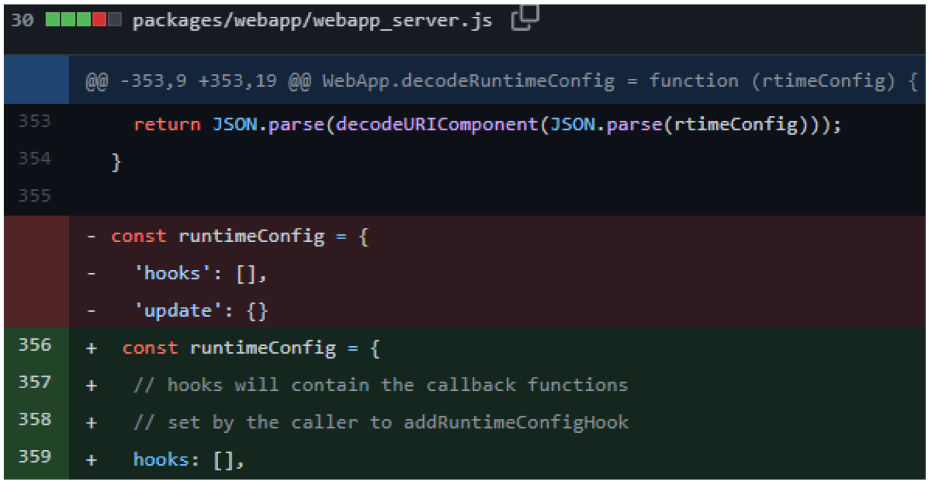
2.3 Usefulness of Review Comments

A review comment is considered to be *useful* by the developers if it triggers code-level changes within its vicinity (e.g., 1–10 lines of the comment location) [15]. For instance, the review comment[1] in Fig. 2.1(a) (bottom row) – *"We don't need single quotes here, And also not below"* – triggered a code change within its vicinity (i.e., Line-359, Fig. 2.1(b)). According to Bosu et al. [15], the comment was useful to the developers, as was explained by the changes made to the code. On the contrary, the review comment[2] in Fig. 2.2 (bottom row) – *"This is a breaking change, no?"*– did not trigger any code-level change, which indicates that this comment might be found *non-useful* by the developers. Although previous studies [76, 15, 32] investigate the usefulness of review comments, their investigation is limited to only closed-source

[1] http://bit.ly/412bXJ7
[2] http://bit.ly/3YSJJPg



(a) Useful code review comment



(b) Code change triggered by the useful review comment

Figure 2.1: Useful code review comment with its associated commit

subject systems. Hence, we analyze the usefulness of review comments from both open and closed-source systems, which might provide useful insights.

2.4 Review Recommendation using Information Retrieval

Information Retrieval (IR) is a popular method that facilitates access to vast repositories of information (e.g., world wide web). In traditional IR, there are two major components: query and corpus. The query consists of a few keywords, whereas the corpus represents a collection of searchable documents. To date, IR has been used to

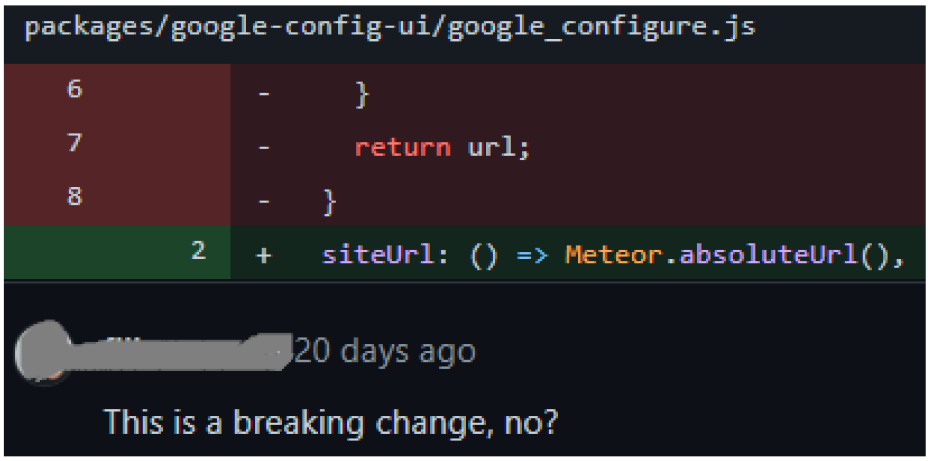


Figure 2.2: Non-useful code review comment

solve at least 20 software engineering problems [77]. An existing study [34] formulates code review recommendation as an Information Retrieval problem. Given an unseen code-level change, their technique searches for a similar code change in the corpus using Information Retrieval and recommends the corresponding comments as the review comments for the unseen change. Their underlying idea was that similar code is likely to receive similar code review comments. Inspired by their research, we also leverage Information Retrieval in code review recommendations (Chapter 4). In particular, we accept a code change (a.k.a., commit) as an input, and extract the changed code, library information and file path to formulate a query. Similarly, we extract the same three items from each of document (a.k.a., code changes) of our corpus. We then represent these structured items from both query and corpus as vectors. Then we execute the query against the corpus using the BM25 algorithm [81] and retrieve similar code changes. Finally, we capture the comments from similar retrieved changed codes and recommend them as code review comments for the query (i.e., unseen code change).

2.5 Word Embedding

Word embedding is a distributed representation of words in a Vector Space Model (VSM) where semantically similar words appear close to each other [64]. An embedding function $E : \mathcal{X} \rightarrow \mathbb{R}^d$ takes an input X in the domain $\mathcal{X}$ and produces its vector representation in a d-dimensional vector space [17]. The vector is distributed

in the sense that a single value in the vector does not convey any meaning; rather, the vector as a whole represents the semantics of the input word [56]. Word embedding overcomes many limitations of other Vector Space Model, such as the sparse representation problem of one-hot encoding or the vocabulary mismatch issue of TF-IDF. Several techniques employ neural networks to learn richer word representations, such as Word2Vec [62]. It uses fully connected layers to understand the context surrounding each word and generates a vector for each word. In our work, we train a Word2Vec model using GitHub CodeSearchNet [38] dataset to generate embeddings for our analysis.

2.6 Summary

In this chapter, we introduce different terminologies and background concepts that would help one to follow the remaining of the thesis. We discuss the modern code review process. We then discuss the sentiments and the usefulness of code review comments. We also discuss review recommendation using Information Retrieval (IR) approach. Finally, we discuss the word embedding which has been used by our proposed approach to extract the semantic information from code review comments.

Chapter 3

How do Automated Tools and Techniques Differ between Open and Closed Source Systems in Assessing their Code Review Quality? An Empirical Study

Review comments play a major role in modern code reviews. Ensuring the high quality of review comments is essential for effective code reviews. Existing work [42] suggests that various development practices including code reviews could be drastically different between open-source and closed-source systems. Such differences might impact the performance of automated tools and techniques in assessing the code reviews from both types of systems. However, to the best of our knowledge, there exist no studies that assess the quality of code reviews from both open-source and closed-source systems using automated tools and techniques. An investigation using both types of software systems is essential to comprehensively understand the behaviours of automated tools and techniques towards these systems. In this chapter, we discuss our first study that empirically investigates this gap in the literature and reports several meaningful insights.

The rest of this chapter is organized as follows. Section 3.1 introduces the study and highlights the novelty of our contribution. Section 3.2 presents our experimental design, datasets, and performance metrics. Section 3.3 discusses our experimental results. Section 3.4 discusses our key findings and the implications of our research. Section 3.5 identifies possible threats to the validity of our work. Section 3.6 discusses the existing studies related to our research. Finally, Section 3.7 summarizes this study.

3.1 Introduction

Peer code review is an engineering practice where developers submit changed code to peers (a.k.a. reviewers). The reviewers then check the eligibility of that code to be integrated into the main code base and recommend useful changes. Beside finding

code-level defects (e.g., logical errors), the review process ensures that the changed code does not degrade the quality of the overall project.

Recently, a lightweight review process, namely *Modern Code Review* (MCR), often assisted by specialized software tools [11], has gained significant popularity in both industry and open-source development. In MCR, change suggestions are made by reviewers in the form of *review comments*, one of the main building blocks of code reviews [15]. These comments are generally discussed by the changed code submitter and the reviewers before making any corrections to the code and then submitting the next version of the changed code. Hence, the *quality* of review comments plays a vital role in MCR. However, manually assessing the quality of code review comments could be both time-consuming and error-prone. Thus, past studies [22, 15, 44, 76] use automated tools and techniques to analyze several comment quality aspects such as *sentiments* or *usefulness* of code review comments.

El Asri et al. [22] detect sentiments in the review comments and show that the reviewer's sentiments could affect a code submitter's perception about the reviews. However, they only use open-source subject systems for their analysis. On the other hand, Rahman et al. [76] and Hasan et al. [32] use several textual and historical properties of code reviews to classify the useful and non-useful code review comments where they use only closed-source software systems. Thus, existing studies use either open or closed-source subject systems to investigate various quality aspects of review comments. According to Khanjani and Sulaiman [42], various practices (e.g., code review, software design, documentation) of open-source systems could be drastically different from that of closed-source systems. Thus, open-source and closed-source software systems could also be different in terms of the sentiments and usefulness of their review comments. Such differences might influence the performance of automated tools in assessing the quality aspects of code reviews from these systems. Thus, analyzing both types of systems is essential to comprehensively understand the behaviours of automated tools and techniques towards them. To the best of our knowledge, there exists no work that assesses the review quality aspects – sentiments and usefulness – from both open and closed-source systems using automated tools and then contrasts their performance between the two types of systems. Such a comparison not only can deliver actionable insights for stakeholders (e.g., developers,

reviewers) but also can encourage better tool supports for modern code reviews.

In this paper, we conduct an empirical study using 2,183 code review comments, assess their two quality aspects – *sentiment* and *usefulness* – using seven automated tools and techniques. We then contrast the performance of these automated tools and techniques between open and closed-source software systems. First, we detect sentiments in the review comments using five sentiment detection approaches – Stanford Core NLP [57], SentiStrength-SE [39], Senti4SD [16], SentiCR [3], and SentiMoji [19]. We then contrast the performance of these tools and techniques between open-source and closed-source systems. Second, we determine the usefulness of code review comments using two usefulness classification techniques – RevHelper [76] and CRA-model [32]. We then contrast the performance of these techniques between open and closed-source systems. Finally, we investigate the role of sentiments in classifying the useful and non-useful code review comments from two types of systems. We also provide a curated dataset[1] of 2,183 code review comments from six open-source and four closed-source software systems that could be useful for third-party replication and reuse. Thus, we answer three important research questions as follows.

(a) **RQ1: How do existing tools and techniques perform in detecting the sentiments in code review comments from open-source and closed-source systems?**

Existing studies [67, 66, 3] provide meaningful insights from their detection and analysis of sentiments in code review comments. For example, El Asri et al. [22] shows that negative sentiments in the review comments could delay the review acceptance by 1.32 days on average. However, their analyses were limited to mostly open-source systems, which might not generalize to closed-source systems. We thus contrast the performance of five existing sentiment detection tools and techniques [3, 16, 39, 57, 19] between six open-source and four closed-source software systems. We find that the performance of the automated approaches in sentiment detection can differ from 1.12% to 14.67% between open and closed-source systems. Although the performance can also differ among the projects from the same type of domain, the difference is significantly higher between the cross-domain. We conduct a follow-up manual analysis to understand the reason behind the varying detection performance across our subject systems.

[1] https://bit.ly/3n9sfRj

We find that reviewers from these two types of systems have varying levels of reviewing experience, which might have led to varying prevalence of sentiments in their code review comments. Since existing tools and techniques were trained on these comments, their sentiment detection performance thus might also have been affected. Thus, our findings provide actionable insights which could be useful for tool development in modern code reviews. For instance, the behaviour of existing sentiment detection tools and techniques could vary based on the development practice of the subject systems, which warrants better tool that takes development practices into consideration.

(b) **RQ2: How do existing techniques perform in classifying the usefulness of code review comments from open-source and closed-source systems?**

Rahman et al. [76] developed RevHelper [76] that uses textual and historical features from reviews to classify the useful and non-useful code review comments. They focus on classifying useful and non-useful review comments using the features that are available during the submission of a review comment. On the other hand, Hasan et al. [32] developed CRA-model [32] to determine the usefulness of a review comment using post-review attributes. However, both of them evaluate their technique using only closed-source subject systems, which might not generalize to open-source systems. We thus contrast the performance of these two techniques between six open and four closed-source subject systems. We find that the performance of the automated techniques in classifying useful and non-useful review comments could vary from 13.33% to 18.42% between open and closed-source systems, which is significant. Although the performance can also differ among the projects from the same type of domain, the difference is significantly higher for the cross-domain. Our follow-up manual investigation shows that open-source reviewers have more reviewing experience than closed-source reviewers, which might explain the presence of more useful review comments in open-source systems (Table 3.1). Since the number of useful review comments differs between open and closed-source systems, the performance of the automated tools could also be influenced by this factor. Furthermore, we find that the usefulness of review comments is positively correlated with the reviewer experience. That is, reviewers with more reviewing experience tend to submit more useful

review comments. Thus, our findings provide meaningful insight regarding the usefulness of review comments from both open and closed-source systems, which could be useful for improving the existing reviewer recommendation tools like CORRECT [75], RevRec [104], and RevFinder [94].

(c) **RQ3: How do sentiments influence the classification of useful and non-useful code reviews in open and closed source systems?**
Although previous studies [15, 32] consider sentiments to classify the useful and non-useful review comments, they only used closed-source systems in their experiment. We thus investigate the influence of sentiments in classifying the useful and non-useful review comments from both open and closed source systems using two existing techniques – RevHelper [76] and CRA-model [32]. We find that sentiments can improve the usefulness classification of review comments by 2.94% for open and 1.67% for closed source subject systems. Since the performance of the machine learning model can be influenced by the size of the training data, we further combine the review comments from both types of systems and evaluate the performance of the selected techniques. We also find that sentiments can improve the usefulness classification of review comments up to 2.94% for the combined dataset, which is promising. To classify the useful and non-useful review comments, existing techniques depend on both textual and historical features (e.g., developer's experience). However, calculating historical features requires analyzing the whole codebase or code change history, which could be both costly and infeasible. Thus, sentiments, a textual feature, could be an inexpensive and feasible alternative to them for improving the classification of review usefulness.

3.2 Study Design

Fig. 3.1 shows the schematic diagram of our conducted study in this paper. We first contrast the performance of five existing sentiment detection tools and techniques [3, 16, 39, 57, 19] between open and closed-source systems. Second, we contrast the performance of two existing comment usefulness classification techniques [76, 32] between open-source and closed-source systems. Finally, we investigate the influence of sentiments in classifying the review's usefulness. This section discusses the major steps of our study design as follows.

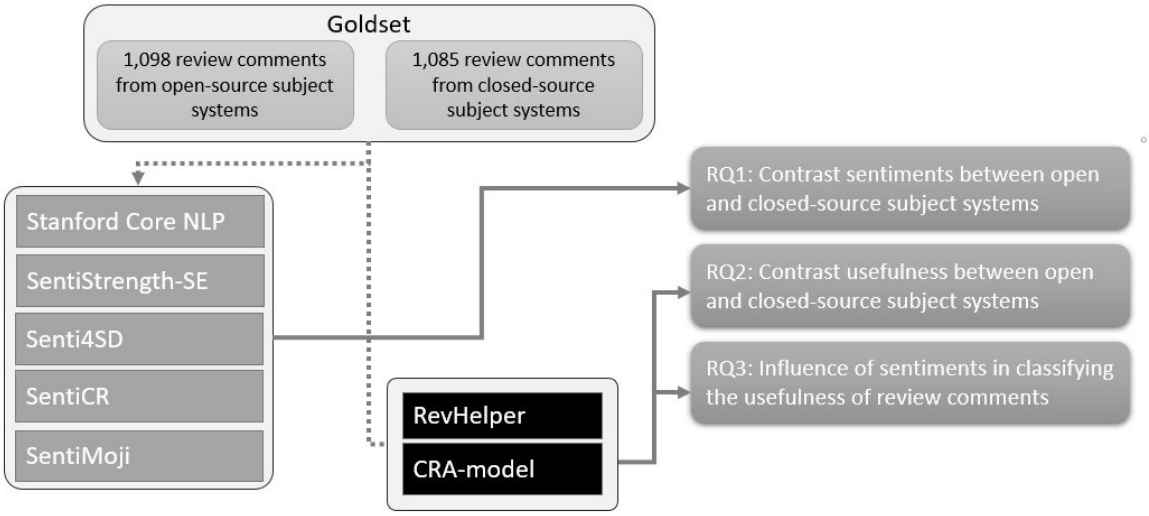


Figure 3.1: Overview of our empirical study

3.2.1 Construction of Review Usefulness Dataset

We conduct our empirical study using two different datasets based on open-source and closed-source subject systems. We construct our first dataset using 1,111 review comments from six popular open-source projects in three popular programming languages – *Python, Java, and JavaScript*. The second dataset (closed-source based) was constructed using 1,116 review comments from an existing benchmark by Rahman et al. [76].

3.2.1.1 Construction of Open-Source Dataset

We used two popular filters for collecting the top projects from GitHub. First, we collect the top three projects from each programming language based on their star counts at GitHub [5, 6], which results in nine projects. Second, we sort the collected projects by the number of their pull requests and take the top two projects from each programming language, which results in six projects for our dataset. These projects contain at least 1000 pull requests in each of them. In GitHub, pull requests contain code changes for reviews and represent the activity or relevance of a project. This filtration keeps only the active projects and removes the forked repositories, as the pull requests are not inherited [51]. To determine the feasibility of our idea, we then

collect the 180 (i.e., 6 X 30) most recent merged pull requests (hereby PR) using GitHub *GraphQL API* [28] from each project. We note that 16.67% of the PRs have reviewing history containing review comments. Being equipped with this evidence, we attempt to find only the eligible PRs. We consider a PR *eligible* for our analysis if it has at least one code review comment and a follow-up change commit.

After the feasibility analysis above, we collect 1800 (i.e., 6 X 300 PRs) most recently merged PRs from the six selected project repositories. Existing techniques [76, 32] extract various features from the changed source files to predict the usefulness of review comments. However, due to frequent changes in a large active project, old source code files might not always be available on GitHub. This might also happen due to any administrative tasks, such as quashing multiple commits from an old version into a new one. We thus collect only the most recent PRs (by PR number) to ensure that the necessary source code files are available at the correct version for our analysis. Rahman et al. [76] report that Pull Requests could be created for scaffolding rather than merging the code changes. Therefore, we collect only the PRs that are merged into the main code base. Such an approach helped us discard the noisy Pull Requests (i.e., pull requests created for scaffolding) from our dataset.

In GitHub, two types of review comments, *pull request comments* and *in-line comments* are submitted by the reviewers. Similar to a former study [76], we focus only on in-line review comments since they are directly connected to some parts of the source code. To collect code review comments, we first manually separate 357 eligible PRs out of 1,800 previously collected PRs. We then create a GitHub API crawler to collect all the review comments, source files, and other meta-information (e.g., pull request author and commit information) for our analysis. We run the crawler and collect 2,575 comments and other associated meta information to construct our initial dataset. Note that the comments we collect in this step include both reviewer's and the author's comments.

To further ensure the quality of our dataset, we perform two filtration steps. We find that $\approx$ 39% comments in our initial dataset are not code review comments (i.e., submitted by the pull request authors). Since our goal is to evaluate various facets of code review comments, we first carefully exclude the non-reviewer comments, which results in 1,578 review comments in our dataset. Similar to prior studies [97, 88],

Table 3.1: Study dataset

Systems	Projects	PLs	PRs	Useful Comments	Non-useful Comments	Total
OSS	Square	Java	58	166 (64.34%)	92 (35.65%)	258
	Elastic Search		32	62 (59.61%)	42(40.39%)	104
	Meteor	JavaScript	57	73 (50%)	73 (50%)	146
	Polymer		81	148 (69.48%)	65 (30.52%)	213
	Keras	Python	58	119 (70%)	50 (30%)	169
	Numpy		71	116 (52.48%)	105 (47.52%)	221
	Total		**357**	**684 (61.56%)**	**427 (38.44%)**	**1,111**
CSS	CS	Python	80	153 (59.77%)	103 (40.23%)	256
	SM		97	164 (58.36%)	117 (41.64%)	281
	MS		88	155 (58.32%)	133 (46.18%)	288
	SR		101	146 (50.17%)	145 (49.38%)	291
	Total		**366**	**618 (55.53%)**	**498 (44.47%)**	**1,116**

OSS = Open-source Systems, CSS = Closed-source systems, PLs = Programming Languages, PRs = Pull Request

we also filter out trivial or short comments (e.g., 'nice', 'thank you', 'LGTM'), which results in 1,111 review comments in our final dataset.

Then the first author of this article carefully annotates each review comment in our dataset either as *useful* or *non-useful* by applying the *change triggering heuristic* as described in Section 2.3. According to our annotation, 684 (61.56%) review comments are useful, and 427 (38.44%) are non-useful in the open-source dataset. Since we use a clearly-defined heuristic to annotate the review comments, the possibility of subjective bias in our dataset could be negligible. However, to analyze the quality of the annotated dataset, the second author also annotated a statistically significant subsample of 286 review comments (95% confidence level and a 5% error margin). We observe almost perfect agreement (0.95 kappa value) between the two annotators, which indicates the consistency of our annotation. The summary statistics of our dataset are shown in Table 3.1.

3.2.1.2 Collection of Closed-Source Dataset

We also collect 1,116 review comments from four closed-source, python-based subject systems provided by an existing benchmark of Rahman et al. [76]. It contains a total of 618 (55.53%) useful and 498 (44.47%) non-useful review comments.

3.2.2 Construction of Sentiment Polarity Goldset

To detect sentiments in review comments using existing sentiment detection tools, the ground truth sentiment label is necessary for each review comment. We thus annotate 1,111 review comments from open-source and 1,116 from closed-source systems involving four human annotators to construct the ground truth. Novielli et al. [66] suggest that the absence of clear guidelines in the annotation process could lead to noisy ground truth, which might negatively affect the performance of sentiment analysis tools. In particular, they show that SE-specific customization in sentiment detection tools for software engineering texts might not guarantee a reasonable accuracy if *ad hoc* annotation is followed. Here, *ad hoc* annotation refers to the assignment of sentiment polarity based only on the subjective perception of the annotator. Shaver et al. [86] proposed a theoretical framework that follows a tree-based structure for the hierarchical classification of emotions. Using this theoretical framework, Calefato et al. [16] designed a model-driven schema for annotating their sentiment classification dataset. A model-driven schema is a concrete approach that involves detailed guidelines and training of the annotator. We adopt this model-driven annotation schema to annotate the review comments in our datasets.

3.2.2.1 Participant Selection and Training

We employed four participants for the annotation task, where each participant was a graduate student and had previous experience in code review activities. First, we conduct a 30-minute session with the participants and then share a tutorial video[2] to explain the guidelines for the annotation task. We also conduct a pilot study and assign *ten* random review comments to each annotator to assess their expertise in annotating review comments. We then conducted a follow-up discussion to clarify any possible ambiguities in interpreting the annotation guidelines. As shown in Table 3.2, we organize four annotators into two groups – {A1, A2, A4} and {A1, A2, A3}, where each group has three members. Our goal was to annotate each comment by three annotators and then use a majority voting to determine the sentiment label of each comment. A similar approach of majority voting was used by the existing literature [16, 65]. It should be noted that annotating each review comment by four

[2]https://youtu.be/wHYHgt27Ae8

Table 3.2: Weighted Cohen's Kappa and observed agreement for the annotators from the annotation study

Dataset	**Pairs**	**Weighted Cohen's Kappa**	**Observed Agreement**
Open-source	A1 & A2	0.63	0.86
	A1 & A4	0.56	0.82
	A2 & A4	0.57	0.84
Closed-source	A1 & A2	0.68	0.89
	A1 & A3	0.64	0.85
	A2 & A3	0.60	0.85
	Average	**0.61**	**0.84**

participants would not have been helpful for the majority voting. The first group was assigned to open-source review comments, while the second one was assigned to annotate the closed-source review comments. Thus, each review comment was assigned to three participants to annotate its sentiment polarity.

3.2.2.2 Sentiment Polarity Annotation

We asked the annotators to indicate the sentiment polarity with one of these four possible values – {negative $\rightarrow$ -1, neutral $\rightarrow$ 0, positive $\rightarrow$1 and mixed $\rightarrow$ 2}. Similar to Calefato et al. [16], we annotate love or joy as positive; anger, sadness, or fear as negative; and mixed for the presence of both positive and negative sentiment. For surprises, we annotate based on the existence of sentiment polarity in the meaning of the comments. The deadline for the annotation task was set to two weeks. The study was finished following a plenary meeting after the deadline.

Once we had all the annotations, there were three types of situations for each review comment.

(a) *Where all three annotations are the same*: We just use this annotation as the gold sentiment label.

(b) *Where one of the three annotations was different*: We use majority voting to resolve the conflict following an earlier study [65].

(c) *Where all three annotations were different*: We resolved this disagreement through a discussion with the corresponding annotators in the plenary meeting after the annotation study.

The third case (all annotators annotated differently) is a tiny fraction of the whole dataset, which is 2.25% for open-source and 1.62% for closed-source review comments. Sentiment annotation is an inherently subjective task [67]. The label assigned to a given text could be influenced by the personality traits of the human annotators regardless of the presence of clear annotation guidelines [83]. In the third case, we observe that some annotators are conservative and provide a neutral label for mild expressions of sentiments, while others are liberal and provide either a positive or negative label for the same expression of sentiments. Such a phenomenon might contribute to the disagreement between annotators. We thus conduct a plenary meeting with all annotators and resolve this disagreement. Furthermore, we exclude around 2% from open-source and 3% review comments from closed-source that express the opposite sentiments simultaneously, i.e., the sentiment polarity is mixed (e.g., Thanks! Sorry it took so long to merge). Thus, our final dataset consists of 2,183 review comments, 1,098 (50.30%) from open-source and 1,085 (49.70%) from closed-source subject systems. Table 3.3 shows the statistics of review comments and their ground truth sentiment labels.

Table 3.3: Distribution of sentiments and usefulness of review comments in the ground truth dataset

Dataset	**Sentiment Polarity**			**Usefulness**		**Total**
	Positive	**Negative**	**Neutral**	**Useful**	**Non-usefule**	
Open-source	180 (16.27 %)	49 (4.43 %)	869 (78.50 %)	676 (61.56 %)	422 (38.44 %)	1098
Closed-source	103 (9.45 %)	120 (11.09 %)	862 (79.08 %)	612 (56.40 %)	473 (43.60 %)	1085
Total	283 (12.86 %)	169 (7.76 %)	1731 (78.79 %)	1288 (59.00 %)	895 (41.00 %)	2183

3.2.2.3 Inter-annotator Agreement Analysis

Although majority voting was used above, we further calculate weighted Cohen's Kappa [24] to determine the agreements between each pair of annotators. As done

by Jongeling et al. [40], we follow the interpretation of κ by Wilson [102]. According to the interpretation, the agreement is *random* if $\kappa \leq 0$, *slight* if $0.01 \leq \kappa \leq 0.20$, *fair* if $0.21 \leq \kappa \leq 0.40$, *moderate* if $0.41 \leq \kappa \leq 0.60$, *substantial* if $0.61 \leq \kappa \leq 0.80$ and *almost perfect* if $0.81 \leq \kappa \leq 1.0$. As shown in Table 3.2, we notice a substantial agreement (i.e., average Kappa value of 0.61) between the annotators. We also measure observed agreement, which is the percentage of agreement between each pair of annotators. As shown in Table 3.2, the average observed agreement is 0.84, which indicates almost perfect agreement.

3.2.3 Selection of Sentiment Detection Tools

Although sentiments are subjective perceptions of the texts, manually annotating them could be costly and time-consuming, as such, not always feasible. Therefore researchers proposed several tools [57, 39, 16, 3, 19] to automatically detect sentiments in the text. Sentiment detection tools can be categorized into three types based on their adopted methodologies as follows.

- *Lexicon-based*: It exploits a lexicon dictionary to detect sentiments in a text.
- *Machine learning based*: It uses several textual features and trains machine learning algorithm to detect sentiment in the text.
- *Deep learning based*: It uses deep learning techniques to detect sentiments in the text.

Despite the popularity of these tools, researchers suggested that the sentiment detection performance of these tools could vary due to semantic shifts, domain-specific jargon, communication style or quality of the dataset [67, 40, 52, 55]. Therefore, using one tool to contrast the performance of detecting review sentiments between open and closed-source systems might not be sufficient. We thus apply two criteria for selecting the sentiment detection tools for our analysis. First, we select the representative tools from the aforementioned categories. Second, we also select such tools that are open-source and have replication packages. Following these criteria, we chose five different sentiment detection tools – Stanford Core NLP [57], SentiStrengthSE [39], SentiCR [3], Senti4SD [16] and SentiMoji [19] – that adopt three different approaches

for detecting sentiments. Except for Stanford Core NLP, these sentiment detection tools are specially designed for sentiment analysis in software engineering texts.

Stanford Core NLP [57] classifies sentiments in a single sentence, and it returns a sentiment value and corresponding polarity. The tool is designed as a Recursive Neural Tensor Network and is trained on the Stanford Sentiment Treebank [89].

SentiStrength, proposed by Thelwall et al. [92], is a lexicon based approach for sentiment classification. It contains several dictionaries that consist of both formal and informal words (e.g., emoji, slang). In these dictionaries, each term has its own sentiment strength. Along with this sentiment information, SentiStrength outputs two integers for a sentence – one for positive emotion and one for negative emotion. Later, Islam and Zibran [39] adopted it for the software engineering domain by adding a domain-specific dictionary. They named it **SentiStrength-SE**.

SentiCR, proposed by Ahmed et al. [3], is a sentiment classification tool especially designed for code review comments. To adapt to the nature of code review comments, SentiCR involves several preprocessing stages. It uses TF-IDF [4] and Bag of Words as the feature set. It also applies SMOTE [18] for over-sampling to minimize the class-imbalance problem.

Senti4SD, proposed by Calefato et al. [16], is a supervised machine learning based sentiment classification tool. It uses three different types of features – (i) generic sentiment lexicons from SentiStrength, (ii) different keyword features including occurrences of uni-grams, bi-grams, upper case words, and slang expressions (iii) word embedding trained on Stack Overflow data using Continuous Bag of Words (CBOW) algorithm [61]. They preprocessed the text by removing URLs, HTML elements and code snippets and finally classified the sentiment polarity using Support Vector Machine (SVM).

SentiMoji is a customized sentiment analysis tool that is trained on a small amount of labelled software engineering data and a large-scale emoji labelled data from Twitter and GitHub. It reuses the architecture of the DeepMoji model [23] that follows a two-stage approach. First, to incorporate technical knowledge, it fine-tunes the DeepMoji architecture using emoji labels. Second, it uses this fine-tuned model to generate vector representation of sentiment labelled text and then train the sentiment classifier based on these vectors.

3.2.4 Selection of Techniques for Classifying Review Usefulness

Review comments could be either useful or non-useful (see Section 2.3 for examples). However, their manual classification is time-consuming and might not always be feasible. Therefore, we select existing techniques trained on our dataset to determine the usefulness of review comments from open and closed-source subject systems. Previously, Bosu et al. [15] proposed an automated classifier to classify useful and non-useful review comments. However, they used only closed-source systems, and many of their features might not be available during the submission of a review comment. On the other hand, Rahman et al. [76] conducted an empirical study to compare different attributes of useful and non-useful review comments. They also propose a model – RevHelper [76] that uses three different machine learning algorithms – Naive Bayes (NB), Logistic Regression (LR), and Random Forest (RF) – to classify the useful and non-useful review comments. They use seven textual (e.g., code element ratio, conceptual similarity, stop word ratio) and eight historical features (e.g., developer experience) to classify the review comments [76]. Recently, Hasan et al. [32] conduct an empirical study at Samsung Research Bangladesh (SRBD) and developed a framework namely Code Review Analytics (CRA), to improve the review effectiveness. They also propose an automated model (hereby, CRA-model) that uses 26 different textual and historical features to classify the review comment. Rahman et al. [76] focus on classifying the usefulness of review comments using the features that are available during the submission of a review (e.g., sentiments, code elements). On the other hand, Hasan et al. [32] focus on classifying the usefulness of review comments using post-review attributes (e.g., number of patches, review thread length).

To build knowledge through a family of experiments, replications are crucial in the software engineering domain [10]. According to Shull et al. [87], software engineering replication can be categorized into two types: 1) *exact replications* and 2) *conceptual replication*. In *exact replication*, the study and procedures are closely followed, whereas, in conceptual replication, different methodologies are used to study the same set of questions. Furthermore, exact replication can be divided into subcategories: i) *dependent*, in which all the variables and conditions are kept as close to the original study as possible; ii) *independent*, in which some aspects of the original study are modified to fit a new context. Since the existing usefulness classification techniques

were developed for specific industries (e.g., RevHelper for an anonymous company and CRA-model for SRBD), some of the features can not be computed in this study. Therefore, we decide to conduct an *exact independent* replication [87].

To replicate the RevHelper, we consider all the 15 features used by Rahman et al. [76]. We also include the majority of the features used by Hasan et al. [32] to replicate the CRA-model. In particular, we use 20 out of 26 features for this replication. First, we exclude two sentiment-related features (comment sentiments and reply sentiments) as we plan to separately investigate the influence of sentiment in our third research question. Second, we also exclude change trigger and status features since we followed the change triggering heuristics to determine the usefulness of review comments. Finally, line change and thread length could not be computed as the corresponding artefacts were not published by Hasan et al. [32], and the required details were missing in the description of their study. For each of the features in our replication, Table 3.4 provides a brief description and indicates whether the feature was adopted either by Rahman et al. [76] or by Hasan et al. [32].

3.2.5 Model Training

We contrast the performance of existing tools and techniques in detecting two review quality aspects (e.g., sentiment and usefulness) between open-source and closed-source systems. We use five existing tools and techniques for sentiment detection from three different aspects – lexicon-based, machine learning, and deep learning for our experiment. A recent study by Novielli et al. [68] suggests that domain-specific tuning or retraining is necessary since jargon from different domains might hurt the performance of a sentiment detection tool. Therefore, we retrained two machine learning-based techniques (Senti4SD and SentiCR) and fine-tuned the deep learning-based technique (SentiMoji) using our training set. To replicate these tools, we use the replication packages provided by the original authors [16, 3, 19]. Stanford Core NLP [57] and Sentistrength-SE [39] leverage a lexicon-based approach where retraining is not necessary. To ensure a fair comparison, we report the average performance based on ten-fold cross-validation settings, as was suggested by the existing literature [66, 3]. We randomly divided our dataset into ten groups, and each of the ten groups was used as the test data once, while the remaining nine groups were used as the training data.

Table 3.4: Features used in our replication to predict code review usefulness

Feature	**Description**	**CRA-model**	**RevHelper**
comment_message	The message text is turned into a fixed dimensional vector using TF-IDF.	Yes	No
question_ratio	Ratio of interrogative sentences in a comment.	Yes	Yes
code_element_number	Number of source code tokens in a comment.	Yes	Yes
code_element_ratio	The ratio of source code tokens in the comment.	Yes	Yes
similarity	Cosine similarity between the source code and review comment.	Yes	Yes
readability	Flesch-Kincaid readability score.	Yes	Yes
word_count	Number of words in a comment.	Yes	Yes
stop_word_ratio	The ratio of stop words in the comment.	Yes	Yes
author_responded	If the author responded to the review comment.	Yes	No
review_interval	The time interval between the code upload and comment submission.	Yes	No
num_patches	Total number of patchsets for this code review.	Yes	No
confirmatory_response	If the code author responds with “Done”, “Fixed”.	Yes	No
gratitude	If the code author responds with “Thank you”.	Yes	No
is_last_patch	If the patch associated with the comment is the last patchset for the review.	Yes	No
patch_id	The patch number of the source code where the comment is submitted.	Yes	No
num_participant	Number of participants in the comment thread.	Yes	No
code_ownership	The number of code changes the reviewer has committed for the current file.	Yes	Yes
code_reviewership	The number of prior code changes of the current file the reviewer has reviewed before.	Yes	Yes
reviewing_experience	The number of code changes the reviewer has reviewed for current project.	Yes	Yes
developer_experience	The number of code changes the code author has committed for current project	Yes	Yes
committed_twice	Whether the developer committed twice	No	Yes
reviewing_twice	Whether the developer reviews the PR twice	No	Yes
reviewed_PRs	Total number of PRs reviewed by the devloper	No	Yes
ext_lib_similarity	Developer’s working experience with the dependencies of a target file	No	Yes

To determine the usefulness of code review comments from open and closed-source systems, we use RevHelper and CRA-model. We use the replication package provided by the original authors [76, 32]. Furthermore, we report the average performance based on ten-fold cross-validation settings following the existing literature [76, 32].

3.2.6 Performance Metrics

We evaluate the performance of existing tools and techniques in terms of accuracy, precision, recall, and F1-score. This choice is in line with the previous research and standard methodology adopted for review quality aspects analysis [52, 66, 19]. We define these metrics as follows.

3.2.6.1 Accuracy

Accuracy is the most commonly used performance metric for classification tasks [27]. It refers to the ratio between the number of correctly classified samples and the total number of samples. Mathematically, it can be defined as follows.

$$Accuracy = \frac{n}{N} \tag{3.1}$$

where n is the number of correctly classified samples, and N is the total number of samples.

Despite being the most popular performance metric for classification tasks, accuracy is often biased towards an imbalance dataset [1]. To mitigate such issues, we also use precision, recall, and F1-score.

3.2.6.2 Precision

Precision measures how precisely a model can identify the classes of the instances. Formally, for any class C, precision is the ratio of the number of correctly classified instances and the number of instances that are classified to the class C [2]. Mathematically,

$$Precision(C) = \frac{TP}{TP + FP} \tag{3.2}$$

where TP is the number of items correctly classified as an instance of class C, and FP is the number of items wrongly identified as an instance of class C.

3.2.6.3 Recall

Recall measures the ability of a model to correctly identify all positive instances out of all actual positive instances. Formally, for any class C, recall is the ratio between the total number of correctly classified instances and the total number of instances of class C [2]. Mathematically,

$$Recall(C) = \frac{TP}{TP + FN} \tag{3.3}$$

where TP is the number of items correctly classified as an instance of class C, and FN is the number of instances of class C that the model could not identify.

3.2.6.4 F1-score

While high precision pushes a model to make fewer mistakes, high recall pushes a model to identify all instances. These requirements are often contradictory. When a model tries to identify all instances of class C (i.e., high recall), it might identify some instances as class C with low confidence. Such identification might, in turn, cause more mistakes and low precision. To balance between these two conflicting requirements, the F-1 score is used. Formally, the F-1 score is the weighted harmonic mean of precision and recall. Mathematically,

$$F1(C) = \frac{2 \cdot Precision(C) \cdot Recall(C)}{Precision(C) + Recall(C)} \tag{3.4}$$

Precision, recall, and F1-score are defined only for binary classification. Since we have three classes for sentiment classification, we combine these measures using both macro and micro-averaging. Macro-averaging calculates the average of metric scores for each class and then computes the arithmetic mean of these scores. On the other hand, micro-averaging computes the measurements for all data points in all classes [106]. The formula for calculating macro and micro-average F1-score (F) is shown below:

$$F_{macro} = \frac{\sum_{i=1}^{k} F_i}{k} \tag{3.5}$$

$$F_{micro} = \frac{\sum_{i=1}^{k} TP_i}{\sum_{i=1}^{k} TP_i + \sum_{i=1}^{k} FP_i} \tag{3.6}$$

F_{macro} and F_{micro} represent the macro and micro-averaged F1-score respectively. F_i, TP_i, and FP_i represent the F1-score, number of true positives, and number of false

positives for the ith class, respectively, where k denotes the number of sentiment polarity classes. We can calculate the macro and micro-averaged precision and recall similarly. We consider that a model performs better than another only when it achieves higher values for both $F1_{macro}$ and $F1_{micro}$.

Table 3.5: Comparing the performance of existing approaches in detecting sentiments in code review comments from open and closed-source subject systems

Approach	**Dataset**	**Positive**			**Neutral**			**Negative**			**Macro-avg**			**Micro-avg**		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
CoreNLP	OSS	0.55	0.49	0.52	0.86	0.43	0.57	0.07	0.85	0.14	0.49	0.59	**0.41**	0.45	0.45	**0.45**
	CSS	0.44	0.56	0.49	0.91	0.49	0.64	0.19	0.83	0.31	0.51	0.63	**0.58**	0.53	0.53	**0.53**
	Merged	0.43	0.36	0.39	0.83	0.45	0.58	0.12	0.71	0.20	0.46	0.50	**0.39**	0.46	0.46	**0.46**
Senti-SE*	OSS	0.71	0.73	0.72	0.93	0.86	0.90	0.14	0.36	0.20	0.59	0.65	**0.61**	0.83	0.83	**0.83**
	CSS	0.64	0.72	0.68	0.91	0.92	0.92	0.48	0.37	0.42	0.68	0.67	**0.67**	0.85	0.85	**0.85**
	Merged	0.75	0.65	0.70	0.90	0.88	0.89	0.29	0.39	0.33	0.64	0.64	**0.64**	0.82	0.82	**0.82**
Senti4SD	OSS	0.74	0.47	0.58	0.88	0.97	0.92	0.0	0.0	0.0	0.54	0.48	**0.50**	0.86	0.86	**0.86**
	CSS	0.76	0.52	0.62	0.86	0.95	0.91	0.33	0.14	0.20	0.65	0.54	**0.58**	0.83	0.83	**0.83**
	Merged	0.84	0.52	0.64	0.86	0.90	0.92	0.56	0.10	0.17	0.75	0.53	**0.58**	0.80	0.80	**0.80**
SentiCR	OSS	0.73	0.67	0.70	0.90	0.93	0.91	0.14	0.09	0.11	0.59	0.56	**0.58**	0.86	0.86	**0.86**
	CSS	0.31	0.52	0.39	0.87	0.85	0.86	0.37	0.25	0.30	0.52	0.54	**0.39**	0.75	0.75	**0.75**
	Merged	0.64	0.49	0.55	0.87	0.90	0.88	0.35	0.37	0.36	0.62	0.59	**0.60**	0.80	0.80	**0.80**
SentiMoji	OSS	0.89	0.55	0.68	0.88	0.99	0.93	0.82	0.28	0.42	0.86	0.61	**0.68**	0.88	0.88	**0.88**
	CSS	0.90	0.69	0.78	0.90	0.98	0.94	0.50	0.21	0.30	0.77	0.63	**0.67**	0.89	0.89	**0.89**
	Merged	0.90	0.64	0.75	0.89	0.98	0.93	0.68	0.25	0.37	0.82	0.63	**0.68**	0.88	0.88	**0.88**

OSS= Open-source system, CSS= Closed-source system,
Senti-SE*=SentiStrength-SE, P = Precision, R = Recall, F1 = F1-score

3.3 Study Result

3.3.1 Answering RQ1: How do existing sentiment detectors perform in detecting sentiments of code review comments from open-source and closed-source systems?

We contrast the performance of five existing tools and techniques in detecting sentiments of review comments between six open and four closed-source subject systems.

Existing tools for sentiment detection can be divided into three different categories – lexicon-based, machine learning-based, and deep learning-based. Table 3.5 shows

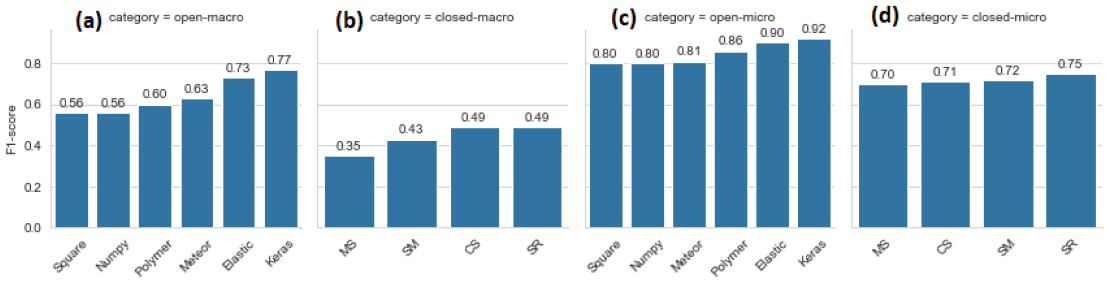
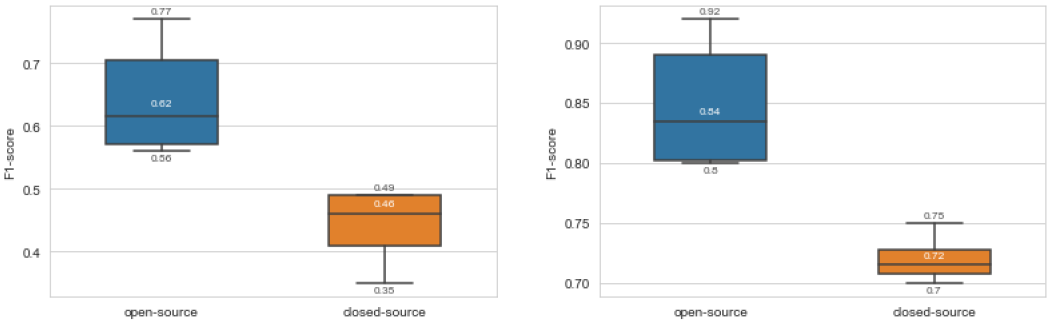


Figure 3.2: Macro-averaged F1-scores from **(a)** open-source and **(b)** closed-source projects and micro-averaged F1-scores from **(c)** open-source and **(d)** closed-source projects

the precision, recall, and F1-score of five sentiment detection tools from these three categories, which are evaluated with both open-source and closed-source systems. From Table 3.5, we see that the machine learning-based tool SentiCR outperforms Senti4SD with a higher macro-average F1-score (0.60). SentiCR was designed to detect sentiments on code review comments, whereas Senti4SD was proposed for sentiment detection on the StackOverflow dataset. Thus, the performance difference between these tools might be expected. However, SentiCR shows a difference of 14.67% in micro-averaged F1-score between open and closed-source systems when sentiments are detected from their code reviews. On the other hand, in the lexicon-based category, SentStrength-SE outperforms Stanford CoreNLP and delivers a macro-averaged F1-score of 0.64 and a micro-averaged F1-score of 0.82. Nevertheless, it also shows a difference of 2.40% in micro-averaged F1-score when sentiments are detected from open and closed-source code reviews. The deep learning-based tool SentiMoji reduces this difference to 1.12%. We thus observe that the performance of existing tools and techniques could differ from 1.12% to 14.67% between open and closed-source systems when sentiments are detected from their review comments.

According to our above analysis, the performance of SentiCR varies the most among all sentiment detection tools. We thus select SentiCR to further compare its performance between two types of systems (e.g., open-source versus closed-source) and within the same type of system. For both open-source and closed-source systems, we keep one project for testing and the remaining projects from the same type for training. From Fig 3.2 (a) and Fig 3.2 (b), we see that the macro-averaged F1-score of SentiCR ranges from 0.58 to 0.77 across the projects from the open-source domain



(a) Macro-average (b) Micro-average

Figure 3.3: Performance comparison of SentiCR in the cross-systems setting.

and ranges from 0.35 to 0.49 across the projects from the closed-source domain. That is, the macro-averaged F1-score of SentiCR varies by 24.67% across the projects in the open-source domain and varies by 28.57% across the projects in the closed-source domain. However, from Fig 3.3(a), we see that this metric ranges from 0.35 to 0.77, which is $\approx$ 55% difference between the two types of systems (e.g., open-source and closed-source). From Fig 3.2 (c) and Fig 3.2 (d), we see that the micro-averaged F1-score of SentiCR varies by 13.04% across the projects in the open-source domain and varies by 6.67% across the projects in the closed-source domain. However, from Fig 3.3(b), we see that this metric score ranges from 0.70 to 0.92, which is $\approx$ 24% difference between the two types of systems (i.e., open-source and closed-source). Thus, the performance of the existing tool in detecting sentiments can differ among the projects of the same domain. However, the difference is much higher between the two types of domains (e.g., open-source and closed-source).

Besides comparing the performance of the automated tools, we further contrast the prevalence of sentiments in the review comments between open and closed-source systems using the ground truth information. From Table 3.3, we see that the closed-source subject systems have $\approx$ 7% more negative review comments. In contrast, open-source subject systems have $\approx$ 7% more positive review comments. Such an imbalance might have contributed to the differences in the performance of automated tools when detecting sentiments from open and closed-source code reviews.

We also perform manual analysis to understand the difference in sentiments between open and closed-source systems. El Asri et al. [22] reported that the peripheral developers (i.e., newcomers) frequently express both positive and negative sentiments,

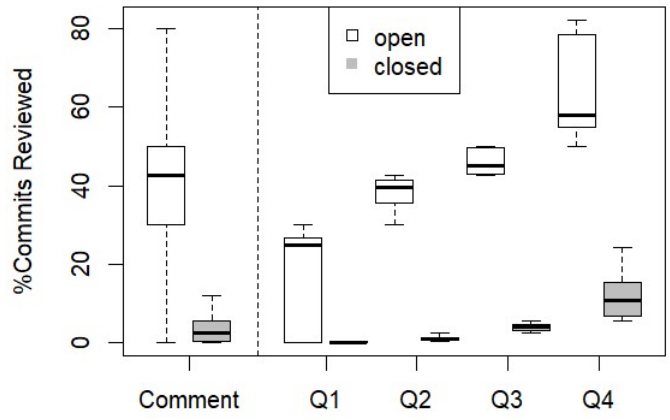


Figure 3.4: Comparison of commit-level reviewing experience between open and closed-source systems.

whereas the core developers (a.k.a., experienced developers) remain neutral when submitting their review comments. Thus, we investigate the reviewing experience of each reviewer, which might have contributed to the differences in sentiment prevalence and detection performance above. In particular, we investigate the reviewers' commit and document reviewing experience as a proxy of their experience in the context of our selected subject systems. We determine the commit reviewing experience by calculating the total number of commits reviewed by a developer from a subject system. Similarly, we determine the document reviewing experience by calculating the total number of source file reviewed by a developer from a subject system. Since both commit and document reviewing experience can be affected by project size, we further determine the percentages of all commits and all documents from a system that are reviewed by a developer. Fig. 3.4 contrasts the reviewers' commit reviewing experience between open-source and closed-source systems. We perform Mann-Whitney Wilcoxon tests following the existing literature [76] to determine the statistical significance of reviewing experience.

From Fig. 3.4, we see that the distribution of commit-level reviewing experience is significantly different (i.e., $p = 2.2e{-}16 < 0.05$, Cliff's $\delta = 0.83$) between open and closed-source systems. Specifically, the median measure of commit-level reviewing experience is $\approx 42\%$ for open-source and $\approx 2.5\%$ for closed-source subject systems. That

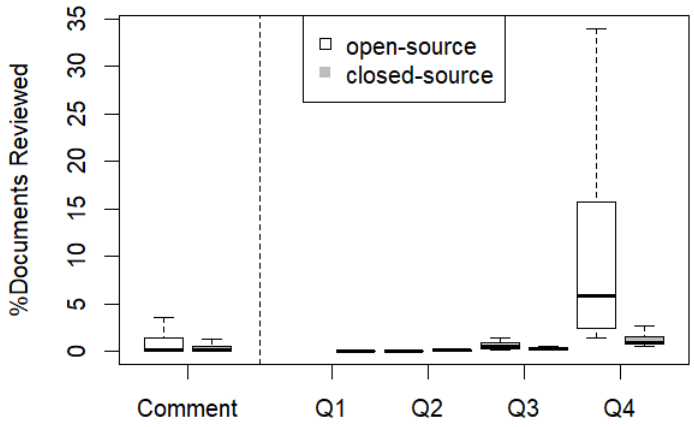
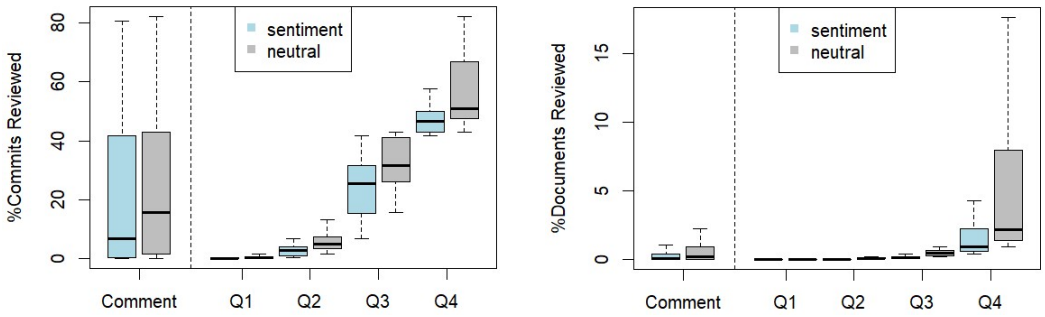


Figure 3.5: Comparison of document-level reviewing experience between open and closed-source systems.

is, open-source reviewers have ≈ 17 times more experience than closed-source reviewers, regardless of the project size. Moreover, we also observe that open-source reviewers have significantly higher reviewing experience from their quartile-based analysis.

Since the above analysis suggest that the closed-source reviewers have less commit-level experience than open-source reviewers, for more granular insights, we further compare the document-level reviewing experience between the two types of systems. Fig. 3.5 contrast the reviewers' document reviewing experience between open-source and closed-source systems. Similarly, we perform Mann-Whitney Wilcoxon tests following the existing literature [76] to determine the statistical significance of reviewing experience.

From Fig. 3.5, we see that the distribution of document-level reviewing experience is not statistically significant (i.e., $p = 0.2 > 0.05$, Cliff's $\delta = 0.26$). For more granular insight, we compare the distribution of each quartile of document-level reviewing experience between open and closed-source reviewers. From Fig. 3.5, we see that the median document-level reviewing experience is ≈ 0 for the first two quartiles, Q1 and Q2, in open and closed-source systems. However, in the fourth quartile, open-source reviewers have a higher median for document-level reviewing experience. Furthermore, this difference in distributions of reviewing experience is statistically significant ($p = 2.2e{-}16 < 0.05$ and Cliff's $\delta = 0.9$). Thus, we see that reviewers from the closed-source systems have less reviewing experience than the reviewers from the open-source systems.



(a) Reviewers' sentiments based on commit-review experience

(b) Reviewers' sentiments based on document-review experience

Figure 3.6: Comparison of reviewer's sentiments based on reviewing experience

Since the above analysis suggests that open-source systems have more experienced reviewers than closed-source ones, we further investigate how sentiments in the review comments could be affected by the experience of corresponding reviewers. In particular, we compare the experiences of code reviewers expressing strong sentiments (i.e., positive, negative) with that of reviewers expressing no sentiments (i.e., neutral). From Fig. 3.6(a), we see that the distribution of commit reviewing experience is significantly different(i.e., $p = 3.2e{-}06 < 0.05$, Cliff's $\delta = 0.14$) between the reviewers expressing strong sentiments and neutral sentiments. More specifically, the reviewers who frequently express strong sentiments have a median measure of $\approx 7\%$ for commit-level reviewing experience. In contrast, the median measure of commit-level reviewing experience is $\approx 15\%$ who remain neutral. Thus, reviewers who frequently express positive or negative sentiments have ≈ 2 times less commit reviewing experience. Furthermore, in Fig. 3.6(b), the distribution of document-level reviewing experience is also significantly different (i.e., $p = 6.28e{-}16 < 0.05$, Cliff's $\delta = 0.25$) between reviewers expressing strong sentiments and neutral sentiments. Specifically, the reviewers who frequently express sentiments have a median measure of $\approx 0.06\%$ for document-level reviewing experience. In contrast, the median measure of document-level reviewing experience is $\approx 0.23\%$ who remain neutral. Thus, reviewers who frequently express strong sentiments have ≈ 4 times less document-level reviewing experience.

Based on the above analysis, we further investigate the correlation between the reviewer's sentiments and reviewing experience (i.e., commit-level and document-level) using the Point-Biserial correlation coefficient [45]. This correlation test measures

the relationship between the dichotomous nominal variable (i.e., binary variable) and numeric variable. Since we have sentiments as the dichotomous nominal variable (e.g., sentiments, neutral) and review experience as the numeric variable, the Point-Biserial correlation measure is well suited for our purpose. According to this correlation analysis, we find that the sentiments are negatively correlated with the review experience. That is, reviewers with less reviewing experience frequently express strong sentiments in their review comments. Furthermore, the correlation between reviewing experience and sentiments was found to be statistically significant. For commit-level experience, correlation coefficient $r_{\text{b}} = -0.18$ and $p = 1.60e{-}17 < 0.05$. For document-level experience, correlation coefficient $r_{\text{b}} = -0.08$ and $p = 0.00 < 0.05$. That is, although the correlation was not strong, it was still statistically significant.

Therefore, our analysis suggests that reviewers with less experience frequently express both positive and negative sentiments in closed-source systems. Our findings from this comprehensive analysis generalize the findings of El Asri et al. [22] for open-source systems as well.

Summary of $\mathbf{RQ_1}$: The performance of the automated tools in detecting sentiments in the code review comments could differ from 1.12% to 14.67% between open-source and closed-source subject systems. We also observe that reviewers from these two types of systems have varying levels of reviewing experience, which might have led to varying prevalence of sentiments in their code review comments. Since existing tools and techniques were trained on these comments, their sentiment detection performance thus might also have been affected.

3.3.2 Answering RQ2: How do existing techniques perform in classifying the usefulness of code review comments from open-source and closed-source systems?

We contrast the performance of two existing techniques for comment usefulness classification between six open-source and four closed-source subject systems. Table 3.6 shows the accuracy, precision, recall, and F1-score of two techniques in classifying review usefulness. From there, we see that CRA-model [32] outperforms RevHelper [76] with an accuracy of 0.76 for open-source, 0.66 for closed source, and 0.71 for the merged dataset. However, the accuracy metrics could be biased towards the majority

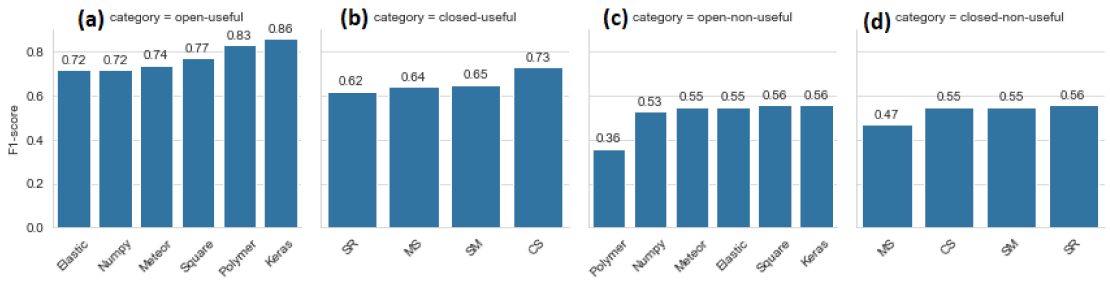


Figure 3.7: F1-score among **(a)** open-source and **(b)** closed-source for useful classification and among **(c)** open-source and **(d)** closed-source for non-useful classification

class of our dataset. We thus further compare the techniques in terms of the F1-score. From Table 3.6, we see that CRA-model also outperforms RevHelper, with F1-scores of 0.83 and 0.64 when detecting useful comments from open-source and closed-source code reviews, respectively. Revhelper uses comment attributes that are available during the submission of a review, whereas CRA-model additionally uses post-review attributes to classify the usefulness of review comments. Thus, the performance difference between these techniques might be explainable.

From Table 3.6, we also note that Revhelper has a difference of 13.33% in accuracy when classifying review usefulness from open-source and closed-source systems. On the other hand, CRA-model differs by 18.42% in the same context, which is noticeable. We also notice a similar difference in F1-score for both techniques. Specifically, for CRA-model, the F1-score can differ from 10.91% to 13.25% between open and closed-source systems when classifying non-useful and useful review comments. For Revhelper, the F1-score can differ from 11.11% to 13.41% in the same context. We thus observe that the performance of existing techniques could differ up to $\approx$ 18% between open and closed-source systems when classifying review usefulness from these systems.

According to our above analysis, the performance of CRA-model varies the most in classifying the usefulness of review comments. We thus select CRA-model to further compare its performance within the same type and between the two types of systems (i.e., open-source versus closed-source). For both open-source and closed-source systems, we keep one project for testing and the remaining projects from the same type for training. From Fig 3.7 (a) and Fig 3.7 (b), we see that the F1-score in classifying useful comments ranges from 0.72 to 0.86 across the projects in the

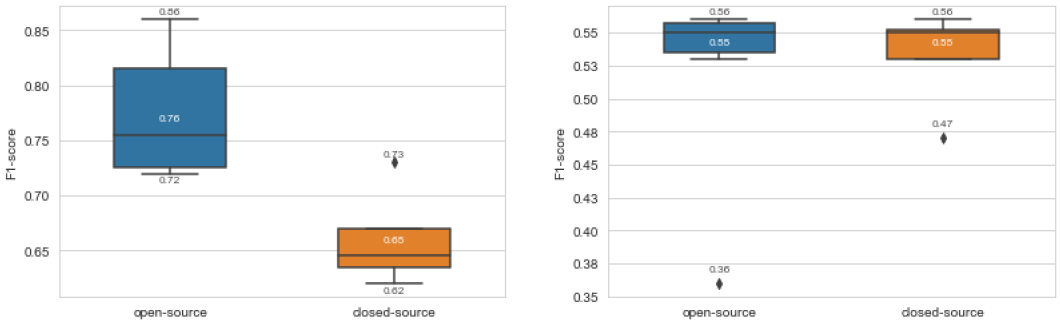
Table 3.6: Performance comparison of RevHelper and CRA-model with different features and datasets

Approach	Features	Dataset	A	Useful			Non-useful		
				P	R	F1	P	R	F1
CRA-model	Textual+Experience+Context*	Open-source	**0.76**	0.76	0.90	0.83	0.79	0.54	0.64
		Closed-source	**0.66**	0.68	0.78	0.72	0.65	0.52	0.57
		Merged	**0.71**	0.71	0.85	0.78	0.71	0.50	0.58
	Textual+Experience+Context*+Sentiment	Open-source	**0.77**	0.77	0.91	0.83	0.80	0.55	0.64
		Closed-source	**0.68**	0.69	0.79	0.73	0.66	0.53	0.58
		Merged	**0.72**	0.72.	0.87	0.79	0.74	0.51	0.60
	Sentiment	Open-source	**0.69**	0.68	0.93	0.79	0.74	0.31	0.44
		Closed-source	**0.61**	0.61	0.87	0.72	0.63	0.28	0.38
		Merged	**0.64**	0.63	0.92	0.75	0.69	0.24	0.35
RevHelper	Textual+Experience	Open-source	**0.75**	0.76	0.88	0.82	0.75	0.56	0.63
		Closed-source	**0.65**	0.69	0.76	0.71	0.62	0.51	0.56
		Merged	**0.70**	0.71	0.84	0.77	0.69	0.50	0.58
	Textual+Experience+Sentiment	Open-source	**0.76**	0.77	0.88	0.82	0.75	0.57	0.64
		Closed-source	**0.66**	0.68	0.77	0.72	0.64	0.53	0.58
		Merged	**0.72**	0.72	0.85	0.78	0.71	0.53	0.60

Context*=Review Context, A = Accuracy, P = Precision, R = Recall, F1 = F1-score

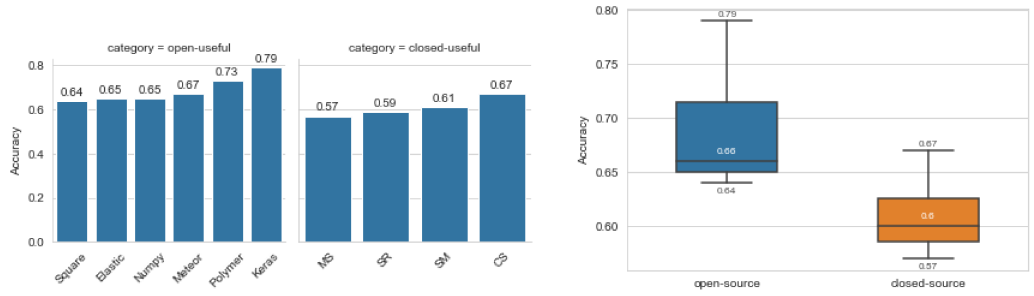
open-source domain and ranges from 0.62 to 0.73 across the projects in the closed-source domain. That is, the F1-score can differ by 16.27% across the projects in the open-source domain and by 15.06% across the projects in the closed-source domain. However, from Fig 3.8(a), we see that this metric score ranges from 0.62 to 0.86, which is $\approx$ 28% difference between the two types of systems. From Fig 3.7 (c) and Fig 3.7 (d), we see that the F1-score in classifying non-useful comments ranges from 0.36 to 0.56 across the projects in the open-source domain and ranges from 0.47 to 0.56 across the projects in the closed-source domain. That is, the F1-score score can differ by 35.71% across the projects in the open-source domain and by 16% across the projects in the closed-source domain. However, From Fig 3.8(b), we see that the metric ranges from 0.47 to 0.56, which is $\approx$ 16% difference between the two types of systems.

We also compare the accuracy of classifying both useful and non-useful review comments. From Fig 3.9(a), we see that the accuracy of the CRA-model ranges from 0.64 to 0.79 across the projects in the open-source domain and ranges from



(a) Useful (b) Non-useful

Figure 3.8: F1-score of CRA-model in cross-projects of open-source and closed-source systems



(a) Within systems (b) Cross systems

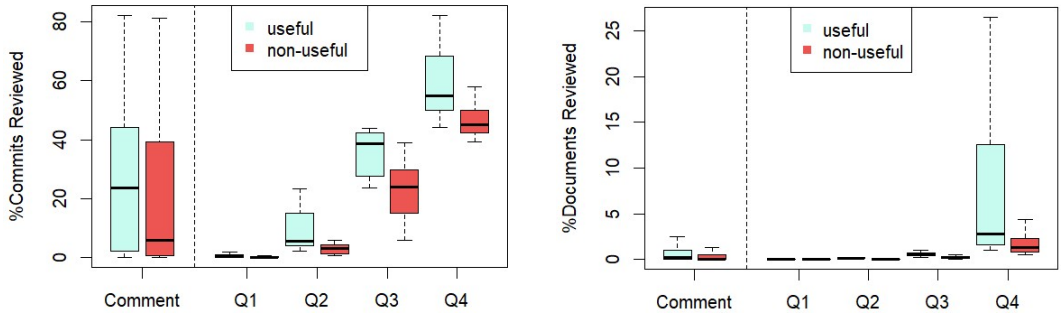
Figure 3.9: Accuracy comparison between open-source and closed-source projects

0.57 to 0.67 across the projects in the closed-source domain. That is, the accuracy can differ by 18.98% across the projects in the open-source domain and by 14.92% across the projects in closed-source domain. However, from Fig 3.9(b), we see that the same metric ranges from 0.57 to 0.79, which is $\approx 28\%$ difference between the two types of systems. Thus, the performance of the existing tool can differ in classifying the usefulness of review comments across the same type of systems. However, the performance difference is much higher between open-source and closed-source systems.

The performance of existing techniques in classifying review usefulness differs between open and closed-source systems. To gain better insights regarding the difference, we thus further compare the prevalence of their useful review comments using ground truth information. From Table 3.3, we see that open-source subject systems have 61.56% useful review comments, whereas the ratio is 56.40% for closed-source subject systems. Thus, closed-source subject systems have $\approx 8\%$ less useful review comments. Such a difference in the prevalence of useful review comments might have contributed to the performance difference in the existing techniques between open and closed-source systems.

According to Rahman et al. [76], the developer's experience is more effective than textual content in classifying the usefulness of code review comments. In RQ1, we observe that review experience can be different between open-source and closed-source systems. We found that open-source reviewers were more experienced in terms of code reviews (See Fig. 3.5). We thus investigate how reviewers' experience could affect the usefulness of their review comments. We compare the experience of code reviewers submitting useful and non-useful review comments.

From Fig. 3.10(a), we see that useful and non-useful review comments are significantly different (i.e., $p = 2.46e{-}14 < 0.05$ and Cliff's $\delta = 0.20$) in terms of the distribution of their reviewer's reviewing experience. More specifically, the reviewers who submit useful review comments have a median measure of $\approx 23.58\%$ for commit-level review experience. In contrast, the median measure of commit-level review experience is $\approx 5.87\%$ for the reviewers who submit non-useful review comments. Thus reviewers who submit useful review comments have ≈ 4 times more commit-level review experience. Furthermore, in Fig. 3.10(b), the distribution of document-level review experience is also significantly different (i.e., $p = 2.2e{-}16 < 0.05$ and Cliff's



(a) Comment usefulness based on commit-review experience

(b) Comment usefulness based on document-review experience

Figure 3.10: Comparison of useful and non-useful review comments based on reviewing experience

$\delta = 0.25$) between reviewers who submit useful and non-useful review comments. Specifically, reviewers who submit useful review comments have a median measure of $\approx 0.27\%$ for document-level review experience. On the other hand, the median measure of document-level review experience is $\approx 0.06\%$ for reviewers who submit non-useful review comments. Thus, reviewers who submit useful review comments have ≈ 5 times more document-level review experience. This finding might be explainable since previous study [76] suggests that review experience is one of the crucial factors for identifying useful review comments.

Similar to RQ_1, we also investigate the correlation between reviewing experience and comment usefulness. According to the Point-Biserial correlation measure, the correlation coefficient $r_\text{b} = 0.16$ for commit-level review experience and $r_\text{b} = 0.14$ for document-level review experience. That is, both reviewing experience exhibits a marginal positive correlation with the review comment usefulness. Interestingly, the correlation between reviewing experience and comment usefulness is also statistically significant. For commit-level experience, $p = 2.92e\text{–}13 < 0.05$ and document-level experience, $p = 1.74e\text{–}11 < 0.05$. Thus reviewers with more reviewing experience tend to submit more useful review comments.

Thus, our analysis suggests that the reviewers who submit useful comments have more reviewing experience than reviewers who submit non-useful code review comments. Since existing techniques leverage the reviewer's experience to classify useful review comments, the difference in the reviewer's experience might contribute to their

varying performance between open and closed-source systems.

Summary of RQ_2: The performance of automated techniques in classifying the usefulness of review comments could differ up to ≈ 18% between open and closed-source systems in our constructed dataset. We also observe that open-source reviewers have more reviewing experience, and consequently, they submit more useful review comments, which might contribute to the varying performance levels of existing techniques between the two types of systems.

3.3.3 Answering RQ3: How do sentiments influence the usefulness of code reviews in open and closed source systems?

Although previous studies [32, 15] use sentiments to classify the useful and non-useful review comments, their investigation was limited to only closed source systems. According to Khanjani and Sulaiman [42], various factors (e.g., code reviews) in open-source software systems could be significantly different from those of closed-source systems. Thus, we investigate the influence of sentiments in classifying the usefulness of review comments from both open and closed-source systems. We use comment sentiments with other textual, historical, and review features and evaluate the performance of RevHelper and CRA-model on two different types of systems. Table 3.6 shows the performance of RevHelper and CRA-model in classifying review usefulness where review sentiment was used as a feature. From there, we see that the sentiments can improve the accuracy of both techniques marginally. We found that sentiments improve the accuracy of Revhelper from 0.75 to 0.76 for open-source and from 0.65 to 0.66 for closed-source, which is 1.67% improvement for both types of systems. On the other hand, sentiments improve the accuracy of CRA-model from 0.76 to 0.77 for open-source and from 0.66 to 0.68 for closed-source, which is 1.67% improvement for open-source systems and 2.94% improvement for closed-source systems, respectively. Nevertheless, the performance of a machine learning model can be affected by the size of the training data. Hence, we combine review comments from both open and closed-source systems and assess the performance of RevHelper and CRA-model. From Table 3.6, we see that the performance improvement for both techniques is still marginal (e.g., 1.67% –2.94%). In our RQ1, we observe that reviewers with less reviewing experience frequently express strong sentiments. In our RQ2, we also see

that reviewing experience exhibits a weak but positive correlation with the review comment usefulness. Since sentiments, reviewing experience and usefulness are correlated to each other, the performance improvements of usefulness classification using sentiments might be explainable. Interestingly, these improvements are also statistically significant according to Wilcoxon Signed Rank test [71], for RevHelper p is $0.000 < 0.05$, and Cliff's δ is 0.88 and for CRA-model, p is $0.008 < 0.05$ and Cliff's δ is 0.88. Thus, we observe from the above analysis that the sentiments can improve the classification of comment usefulness marginally (e.g., $\approx 3\%$), but the improvement is statistically significant.

Since we notice that sentiments have significant impacts in detecting the usefulness of review comments, we further assess the performance of the best classification model (i.e., CRA-model) using only the sentiments feature. Interestingly, sentiment alone can classify useful and non-useful review comments with up to 0.69 accuracy (Table 3.6). Our follow-up investigation suggests that $\approx 64\%$ of neutral comments are useful, whereas $\approx 64\%$ comments with positive/negative sentiments are non-useful. Such a finding indicates a direct relationship between the sentiments and the usefulness of review comments. Furthermore, we also perform the Chi-square test [25] of independence to investigate the dependency between the sentiments and usefulness of review comments. Our null hypothesis is sentiments and usefulness of review comments are independent. According to this statistical test, we found that the p-value is $0.0002 < 0.005$, and the effect size is 0.03. Although the effect size is small, we can still reject the null hypothesis according to our p-value and accept the alternative hypothesis. That is, sentiments and usefulness are dependent on each other.

Thus, our experiment above suggests that sentiment can significantly influence the usefulness classification of code review comments. Our statistical analysis also suggests the dependence between the usefulness and sentiments of review comments, which is actionable insight for improving automated tools and techniques. For instance, the usefulness classification technique like RevHelper uses textual and historical features to classify useful and non-useful review comments. Since sentiment is a strong factor for detecting the review usefulness (see Table 3.6), it could be a feasible alternative to various costly metrics (e.g., authoring experience, reviewing experience) for the usefulness classification of review comments. Sentiment-based classifiers can

also be used to prevent non-useful code reviews as a lightweight solution.

Summary of RQ_3: Sentiments can improve the classification accuracy of review usefulness by 3%, which is promising. It also could be a feasible alternative to various costly metrics (e.g., authoring experience, reviewing experience) that are often used to classify useful and non-useful review comments.

3.4 Key Findings and Implications

3.4.1 Experienced reviewers provide more useful review comments.

In our dataset (open and closed-source), in terms of document-level review, the writers of non-useful comments are $\approx$ 77% less experienced than the writers of useful comments. We also notice similar findings in our second research question (Section 3.3.2). Thus, our study complements the finding of Rahman et al. [76] about the importance of experience in writing useful comments. This finding can not only benefit the review usefulness classification but also can help improve the existing reviewer recommendation tools like CORRECT [75], RevRec [104], and RevFinder [94].

3.4.2 Less experienced developers express stronger sentiments in their review comments.

We find that developers who express sentiments in their review comments have $\approx$ 70% less reviewing experience than the developers who do not express strong sentiments (i.e. neutral sentiment) in review comments. Thus, our study complements the findings of El Asri et al. [22] with similar findings in closed-source systems. We further find that in *open-source systems*, reviewers who express sentiments have a median of $\approx$ 0.0% reviewed document, where for *closed-source systems*, it is 0.16%. This indicates that in open-source systems, only the absolute newcomers express sentiments in their review comments. Since sentiments play an important role in the usefulness classification of review comments (Section 3.3.3), our findings could be useful for recommending reviewers capable of writing useful review comments containing neutral sentiments.

3.4.3 Use of sentiments improves the automatic classification of review usefulness.

We show that the use of sentiment label in usefulness classification can improve the accuracy by $\approx 3\%$. Interestingly, among all the feature set for classifying the usefulness of review comments, sentiment can be easily computed from the review comments text. All other historical and review features require searching the full codebase or even the whole code-change history. Therefore, an inexpensive feature like sentiment could be a feasible choice for improving the usefulness classification of review comments.

3.4.4 Open-source code reviews are more sparse and diverse.

While, in general, reviewers of open-source systems are more experienced, counter-intuitively, a significant portion of them are totally inexperienced. For instance, 8.38% of our open-source review comments have reviewers who have zero commit reviewing experience, which is 0.65% for closed-source. Furthermore, a significant amount of feature values in open-source reviews are zero (e.g., 72.04% external library experience), making the dataset sparse. These characteristics suggest that automatic tool support for code review in open-source systems should employ appropriate techniques that are robust to data sparsity.

In addition, open-source projects have more diverse metric values than closed-source projects based on standard deviation. For example, open-source systems have a standard deviation of 17532 for commit reviewing experience, while closed-source systems have only 713 for the same metric. Likewise, open-source systems have a standard deviation of 58.48 for source file reviewing experience, while closed-source systems have only 19.91 for the same metric. These findings suggest that code review metrics vary more in open-source than in closed-source projects. Interestingly, these variations are more noticeable among open-source projects with different programming languages. For instance, two Java-based projects have a standard deviation difference of 6.38 for source file reviewing experience. However, this difference increases to 54.77 between Java and JavaScript-based projects and to 62.90 between Java-based and Python-based projects. Therefore, special attention should be paid to data-diversity when developing cross-project tool support for different programming

language domains.

3.5 Threats To Validity

The threats to *internal validity* relate to experimental errors and biases [95]. Our dataset construction involves manual analysis where we followed the *change triggering heuristics* of Bosu et al. [15]. While human errors are unavoidable during analysis, the first and second authors performed the analysis in parallel and cross-checked with each other under the constant supervision of the third author to mitigate errors. We also manually cross-check for false positives using random samples. Finally, we construct our open-source dataset with 1,111 review comments from six open-source subject systems spending 80 person-hours. However, there might still be a little risk that the overall characteristics of open and closed-source systems might deviate from our selected systems due to confounding factors such as language domains and code review practice at different companies.

Furthermore, we construct our sentiment benchmark through manual annotation. Sentiment annotation is a subjective process since individual perception can be influenced by personality traits [83]. To mitigate this threat, we provide clear guidelines to the annotators based on the theoretical framework of Shaver et al. [86] and three annotators annotated each review comment. In addition, we exclude around 2% and 3% review comments from open and closed-source systems, respectively, that contain mixed and complex sentiments. The inter-annotator agreement (average weighted Cohen's Kappa, $\kappa = 0.61$) and observed agreement (percentage of agreement between a pair of annotators = 0.84) confirm the substantial reliability of our ground truth sentiment benchmark [40]. Finally, we construct our sentiment benchmark with 2,183 review comments from both open and closed-source systems spending 60 person-hours.

Threats to *external validity* relate to the generalization of a study [74]. We use six open-source and four closed-source subject systems for our analysis, while the choice of project selection can introduce potential bias. To mitigate this threat, we choose our subject systems based on a feasibility analysis where we consider the project's popularity using their star counts on GitHub. Furthermore, we also filter out inactive projects based on the number of pull requests. However, the characteristics of

our selected subject systems might not be an absolute representation of open-source subject systems.

Finally, the threat to *construct validity* relates to the suitability of the experimental design. As we mentioned in Section 3.2.1, we only consider eligible pull requests. However, this might introduce a bias towards the useful review comments. Furthermore, we select seven automated tools and techniques from the literature and evaluate their performance in assessing code review comments. However, in accordance with the previous work [19, 66, 3, 39, 16, 57], we use several appropriate metrics such as precision, recall, micro and macro-averaged F1-score, and accuracy to evaluate those tools and techniques, which indicates a little to no threat to the construct validity of our findings.

3.6 Related Work

Review comments are one of the main building blocks of modern code review since reviewers submit the change suggestions in the form of comments. Hence, the quality of the comments plays a significant role in ensuring an effective code review process. Previous studies [3, 15, 12, 76] analyze several quality aspects of review comments such as sentiments and usefulness. Ahmed et al. [3] analyzed the sentiments in the code review comments and proposed an automated sentiment classification technique – SentiCR. On the other hand, Efstathiou and Spinellis [21] studied the language of code review comments and reported that language does matter. Furthermore, Paul et al. [70] conduct an empirical study using six open-source projects to contrast the sentiments of review comments between male and female developers. They observe that male developers express more sentiments in the review comments than females. El Asri et al. [22] showed that peripheral developers (i.e., newcomers) express more sentiment than the core developers (a.k.a., experienced developers) during code reviews. At the same time, Huq et al. [37] reported that fix-inducing changes (changes that introduce bugs in the system) are preceded by positive review comments, which can help anticipate the bug in the codebase. Although these studies provide useful insight into the sentiments and their impact on the code review process, their analysis is limited to open-source subject systems.

Apart from sentiment detection, a few studies were conducted to analyze the

usefulness of code review comments using textual characteristics and reviewer experience [15, 76, 49]. Baysal et al. [12] studied the impact of various technical and non-technical factors that affect the duration of the code review process. They observed that reviewer experience could significantly affect the outcome of code reviews. Similarly, Kononenko et al. [43] conducted an empirical study using a large open-source system Mozilla to investigate code review quality. They observed that personal metrics such as reviewer experience are associated with the quality of the code review process. Moreover, existing studies also reported that the reviewer's expertise has the strongest relationship with the incidence of software post-release defects [46, 59]. Later, Rahman et al. [76] investigated various textual properties and reviewer experiences to determine the usefulness of review comments using four closed-source systems. They also propose a classification model that uses 15 different textual and historical attributes of reviews to classify useful and non-useful comments. Recently, Hasan et al. [32] proposed another classification model that uses 26 different textual, historical, and review attributes to classify useful and non-useful comments. While Rahman et al. [76] focuses on identifying the usefulness of a comment during the submission of a review, Hasan et al. [32] focus on identifying the useful comment after the submission of a review. Nevertheless, these studies provide useful insight into several comment attributes, However, their analysis is limited to only closed-source systems, which might not generalize to open-source systems.

Thus, to summarize, unlike the relevant literature, we analyze two review quality aspects (e.g., sentiments, usefulness) from both open and closed-source software systems using automated tools and techniques. We also investigate the interplay between sentiments and usefulness, and report that sentiment is an important factor for classifying the usefulness of review comments. Finally, we also provide a comprehensive benchmark dataset based on open and closed-source subject systems. To the best of our knowledge, there exists no work that comprehensively assess the performance of automated tools and techniques in measuring the quality of review comments and then compare their performance between open-source and closed-source software systems, which makes our work novel.

3.7 Summary

To summarize, we perform an empirical study using code review comments from six open-source and four closed-source subject systems. Our study aims to assess the performance of automated tools and techniques in measuring two quality aspects – *sentiments* and *usefulness* – of review comments and then contrast their performance between open and closed-source subject systems. We found that the performance of the automated tools significantly differs between open and closed-source subject systems. Moreover, we also observe that sentiment can improve the automatic classification of review usefulness up to $\approx 3\%$, which is promising. Our manual analysis also reveals that experienced reviewers submit more useful review comments than less experienced (a.k.a novice developers) reviewers. Given the significance of code review experience in writing useful review comments, the review process can be further enhanced with automated support in review comments recommendations for the novice developer. In Chapter 4, we thus propose a novel technique for relevant review comments recommendation – RevCom – that leverages various code-level changes using structured information retrieval. In particular, our technique can automatically recommend the review comments, which could reduce the cognitive effort and time required to write the review comments by the developer.

Chapter 4

RevCom – Recommending Code Reviews Leveraging Code Changes with Structured Information Retrieval

Our first study in Chapter 3 empirically assesses two review quality aspects – sentiments and usefulness – of code reviews from both open-source and closed-source systems using automated tools and techniques. It also shows how these quality aspects could vary between open-source and closed-source systems when assessing them. This study also reveals that less experienced developers submit more non-useful review comments to both types of systems, which warrants automatic support in writing code review comments. In this chapter, we discuss our second study that proposes a novel approach to help developers write better code review comments.

The rest of this chapter is organized as follows. Section 4.1 introduces the study and highlights the novelty of our contribution. Section 4.2 illustrates the usefulness of our technique with motivating examples. Section 4.3 presents our proposed technique for recommending code review comments using structured information retrieval. Section 4.4 discusses our experimental design, datasets, and evaluation metrics. Section 4.5 discusses the evaluation result of our proposed technique. Section 4.6 discusses the existing studies related to our research. Section 4.7 identifies possible threats to the validity of our work. Finally, Section 4.8 summarizes this study.

4.1 Introduction

Modern code review (MCR) has been reported as one of the popular quality assurance practices in software development and maintenance [9, 30]. In MCR, change suggestions are made by reviewers in the form of review comments, one of the main building blocks of code reviews. However, manually writing code review comments is still a challenging task, which requires significant time and cognitive effort. MCR involves examining the source code from different aspects such as logic, functionality, complexity, code style, and documentation [50]. Due to the size and complexity of

modern software projects, the number of review requests is also high [105, 80]. A code submitter might need to wait for 15 – 64 hours before receiving any code review [35], which could hurt their productivity. Thus, an automated recommendation of meaningful code review comments could benefit both a code submitter and a code reviewer. Recommended reviews can help the reviewer write better review comments with reduced effort while shortening the wait time for the code submitter [34].

Several existing approaches [29, 88, 98, 50] recommend or generate code review comments using Deep Learning (DL) networks. Earlier works [29, 88] use Long Short-term Memory (LSTM) networks with an attention mechanism [8] to recommend code review comments. Later approaches [98, 50] employ more sophisticated architecture such as Text-To-Text Transfer Transformer (T5) [101] to generate code review comments. However, they require specialized computing resources (e.g., 16 × 40GB GPU [101]), which could hurt their scalability. They also might require long training time (e.g., 12 days [101]).

Recent studies [26, 33, 54, 60] suggest that simpler approaches, such as Information Retrieval (IR), can perform better than complex Deep Learning models with less computational time and resources. Hong et al. [34] propose an IR-based approach that leverages method-level similarity in recommending code review comments. Although their approach outperforms Deep Learning models, it could be limited in several aspects. First, they use the Bag of Words (BoW) model [72] that represents source code as token vectors ignoring code structures and semantics. Source code contains both structured (e.g., methods, library information) and unstructured items (e.g., code comments). Second, they report their findings for only Java-based projects, which might not generalize to other programming languages. Finally, their approach to recommending review comments was evaluated only using method-level information in the source code. However, method bodies might not cover all the changes that require code reviews. Thus, the existing approaches might not perform well in recommending reviews for the code changes outside of a method body (see listing 4.1, 4.2). According to Li et al. [50], structured information such as *diff* contains all types of changes in the source code and thus can help better understand the semantics of any code changes. However, the work of Hong et al. [34] overlooks this structured information and recommends reviews only for the changes in the method.

In this study, we propose a novel technique for relevant review comments recommendation – RevCom – that leverages various code-level changes using structured information retrieval. RevCom uses different structured items from source code and can recommend relevant reviews for all types of changes (e.g., method-level and non-method-level). Our evaluation using three performance metrics show that RevCom outperforms both IR-based and DL-based baselines by up to 49.45% and 23.57% higher BLEU score in recommending review comments. We find that RevCom can recommend review comments with an average BLEU score of $\approx 26.63\%$. According to Google's AutoML Translation documentation[1], such a BLEU score indicates that the review comments can capture the original intent of the reviewers. All these findings suggest that RevCom can recommend relevant code reviews and has the potential to reduce the cognitive effort of human code reviewers.

4.2 Motivating Example

To demonstrate the capability of our approach – RevCom, let us consider the example in Listing 4.1. The code snippet is taken from the *elastic/elasticsearch* Java repository[2]. The example shows a class-level change. According to the review comment in line 8, the reviewer suggests changing the privacy of variable *key* (see line 6) from *public* to *private*. We see that RevCom recommends exactly the same comment that the reviewer suggests (a.k.a. ground truth). RevCom retrieves the suggested review comment from a similar pull request from the same repository. Reviewers often provide similar types of review comments for similar code changes [34]. Our approach can exploit structured information from those changes and can recommend exact review comments occasionally.

Listing 4.2 shows another example containing a library-level change. The code snippet is from the *ansible/ansible* Python repository[3]. We see that even though the recommended Comment from RevCom does not exactly match the ground truth, both of them express the same semantic information.

[1]https://bit.ly/3wGpCIx
[2]https://bit.ly/3H7jvCt
[3]https://bit.ly/3QTO6H4

```
Code Change:
@@ -50,7 +52,7 @@
public static class Bucket extends InternalMultiBucketAggregation.
    InternalBucket implements Histogram.Bucket {

-         final long key;
+         public final long key;
-----------------------------------------
Ground Truth: "Could you explain why this needs to be public now? I
    think we should try to keep this package private if possible".

Recommended Comment: "Could you explain why this needs to be public
    now? I think we should try to keep this package private if
    possible."
```

Listing 4.1: Example of class-level code change

Unfortunately, the state-of-the-art IR-based technique – CommentFinder [34] could be limited for these change scenarios. First, in Listing 4.1, the change is related to a class-level variable which is declared outside of a method. Second, in Listing 4.2, this change is related to library information which is also not a part of any method. On the other hand, since our approach captures various code-level changes, it can recommend code reviews for both method-level and non-method-level changes.

```
Code Change:
@@ -0,0 +1,125 @@
+#!/usr/bin/python
+from __future__ import absolute_import, division, print_function
+from ansible.module_utils.aws.core import AnsibleAWSModule
+ from ansible.module_utils.ec2 import ( camel_dict_to_snake_dict,
     ec2_argument_spec)
-----------------------------------------------
Ground Truth: "These imports aren't needed but you will need ‘
    camel_dict_to_snake_dict‘ from ‘ansible.module_utils.ec2‘"

Recommended Comment: "Sorry, use ‘camel_dict_to_snake_dict‘ from ‘
    ansible.module_utils.ec2‘"
```

Listing 4.2: Example of library-level code change

4.3 Approach

In this section, we present our proposed technique – *RevCom* – that recommends relevant review comments by leveraging structured information from the source code. Figure 4.1 shows an overview of our technique, which consists of four steps. We describe the details of each step below.

4.3.1 Structured Information Extraction from Query and Corpus

RevCom uses a structured IR-based approach to recommend relevant code review comments. Since IR-based techniques do not require any training phase, it significantly reduces computational time compared to DL-based alternatives [54, 34]. RevCom takes a *diff* and corresponding source code document as input (Step a, Figure 4.1). A *diff* hunk is a sequence of code that represents the code changes between two versions of the same source file [88]. It follows a structured format containing the number of changed lines (denoted by @...@), added lines (denoted by +), deleted lines (denoted by −), and other contextual information (e.g., surrounding lines of added and deleted lines) from the source code (see Listing 4.1, 4.2). We extract added lines, deleted lines and contextual information from the *diff* as changed code fragment for the corresponding review comments. In our dataset, the changed code has a median of 14 lines.

Rahman et al. [76] suggest that experience with structured information from source code (e.g., library information) could help code reviewers write better review comments. Although a diff could contain the changed library information, unchanged libraries from the source code can provide additional contexts (e.g., existing dependencies), which could be valuable for code reviews. We thus extract the changed and unchanged library information (e.g., import statement or package name) from the diff and source code, respectively.

Li et al. [48] uses file path similarity to determine a reviewer's expertise and then recommend the relevant code reviewers. On the other hand, Hong et al. [34] suggests that similar code segments (e.g., method bodies) are likely to receive similar code review comments. Inspired by these works, we hypothesize that similar source files are likely to receive similar review comments. Thus RevCom also uses file path

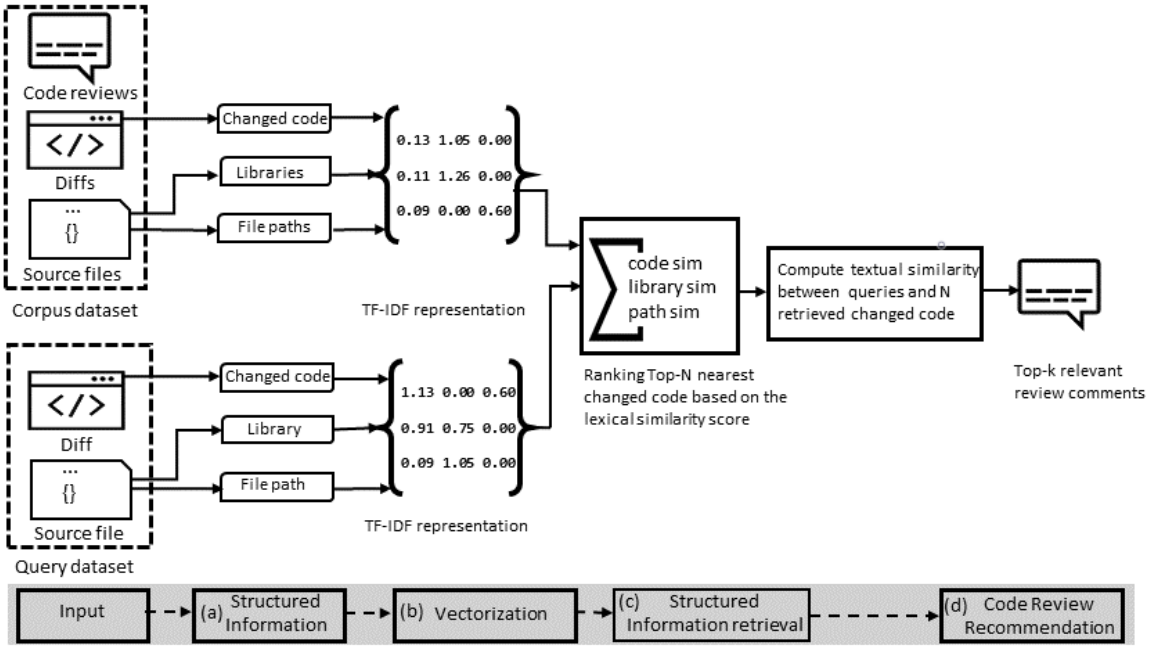


Figure 4.1: An overview of our proposed approach– RevCom

information to recommend the relevant review comments.

4.3.2 Vectorization of Query and Corpus

To facilitate similarity calculation, we represent all changed code fragments, library information, and file paths in TF-IDF vector space. To do so, we perform a code tokenization to break each changed code fragment, import statement, and file path into a sequence of code tokens. As suggested by Rahman et al. [78], we remove punctuation characters (except '+' and '−') to ensure that the code tokens are not artificially repetitive. After that, we convert the sequence of tokens into the frequency vector of code tokens using the *TfidfVectorizer* function of *scikit-learn* library (Step b, Figure 4.1). Since our studied programming languages are case-sensitive, we neither convert them to lowercase nor use any normalization technique (e.g., lemmatization and stemming) to reduce the inflectional forms.

4.3.3 Structured Information Retrieval

Source code contains both structured (e.g., methods, library information) and unstructured (e.g., code comments) items. Our approach extracts three structured items from three different sections of the source file – changed code, file path, and library information. To leverage all three structured items, we perform separate searches for each based on their lexical similarity score. For each single instance (e.g., code change), we first formulate three different queries using the structured items. Then we search for similar vectors in the corpus based on their lexical similarity. To calculate the lexical similarity between the query and corpus, we use the BM25 similarity score. BM25 is a probabilistic framework which overcomes several limitations of TF-IDF similarity, such as term saturation and document length issue [81]. We perform a weighted sum of the similarity scores for all three structured items using Equation 4.1 (Step c, Figure 4.1).

$$LexSim(Q, T) = \sum_{q \epsilon Q} w_q \cdot BM25(q, T_q) \tag{4.1}$$

Here, Q is the set of query instances, T is a single instance from the corpus, T_q is the same structured item as q from T, $BM25(q, T_q)$ is the BM25 similarity between q and T_q, and w_q is the optimized weight for query term q. To generate an optimized weight for each structured item, we follow an existing algorithm by Tian et al. [95]. The minimum and maximum values for the weight are 1 and 3. This weighted sum approach prioritizes a query term over others, even if the term is small in size (e.g. library information). Since context makes certain query terms more important than others, the weighted sum approach performs better than a simple arithmetic sum.

Based on this combined similarity score between *query* and *corpus*, we rank top-N *changed code fragments* from the corpus, which have the highest similarities with the query. Following relevant studies from the literature [34, 88], we use $N = 10$ in our experiment. We thus retrieve the top-10 similar *changed code fragments* from the corpus.

4.3.4 Review Comment Recommendation

Review comments associated with the retrieved *changed code fragments* might be relevant for a given changed code (e.g., query instance). However, the lexical similarity calculated above (Section 4.3.3) does not consider the actual order of code tokens. The

order of code tokens in retrieved *changed code fragments* could be different from the query *changed code fragment*, which could render the similarity measure spurious. For example, in a function, the order of parameters such as a, b and b, a are not the same. Therefore, it is important to consider the order of the tokens to compute the similarity.

We thus use Gestalt Pattern Matching (GPM) to calculate the textual similarity between the top-N retrieved *changed code fragments* and the query *changed code fragment* following an existing study [34]. This similarity measure calculates the textual similarity between the two documents while preserving the order of the characters. Given a query *changed code fragment* and top-N retrieved *changed code fragments*, GPM first searches for the Longest Common Substring (LCS) between the two *changed code fragments*. Then it uses the following equation to calculate their similarity.

$$GMP(diff_Q,\ diff_R) = \frac{2 \times N_C}{N_Q + N_R} \tag{4.2}$$

Here, N_Q is the number of characters in the query *changed code fragment*, N_R is the number of characters in the retrieved *changed code fragment*, and N_C is the number of characters in the longest common substring. Based on this textual similarity, we again rank the retrieved top-10 *changed code fragments* (step-c) against the query instance. Then, we collect the corresponding comments from these *changed code fragments* and recommend them as code review comments for a given changed code. (a.k.a. query instance).

4.4 Experimental Setup

We curate a dataset of $\approx 56K$ *diff* and review comment pairs from eight (four Python and four Java-based) popular projects. We evaluate the performance of RevCom using three appropriate metrics from relevant literature – BLEU score [69], perfect prediction, and semantic similarity [31]. We also compare the performance of RevCom with two state-of-the-art baselines [34, 50]. In our experiments, we thus answer the four research questions as follows.

- **RQ_1:** How does RevCom perform in recommending review comments in terms of different evaluation metrics?

- **RQ_2:** How do different structured information influence the performance of RevCom?

- **RQ_3:** How do different vectorization techniques influence the performance of RevCom?

- **RQ_4:** Can RevCom outperform the state-of-the-art IR-based and DL-based techniques?

4.4.1 Dataset Construction

To conduct our experiments, we curate a dataset of ≈ 56K *diff* and review comment pair from GitHub[4] using its REST API. We first collect the top 20 Java and the top 20 Python repositories based on their star count from GitHub [5, 6]. In order to ensure the quality of our dataset, we then sort the projects by the number of pull requests and filter out projects with less than 1500 pull requests. As discussed in section 2.1, pull requests contain the code change for the review and represent the activity or relevance of a project. This filtration keeps only the active projects and removes forked repositories, as the pull requests are not inherited [50]. After this filtration process, we find four Java and four Python repositories with more than 1500 pull requests. For each selected project, we create a GitHub API crawler to collect all the *diff*, source code, and corresponding reviews. We run the crawler and collect a total of ≈ 297K comments and other associated meta-information (e.g., pull request and commit information) to construct the *initial* dataset. Note that the comments we collect in this step include both reviewers' and the authors' comments.

To further ensure the quality of our dataset, we perform three filtration steps. We find that ≈ 64% comments in our initial dataset are not code review comments (e.g., submitted by the pull request authors). Since our goal is to recommend code review comments, we first carefully exclude the non-reviewer comments, which results in ≈ 189K review comments in our dataset. Similar to prior studies [97, 88], we also filter out trivial or short comments (e.g., "nice", "thank you", "LGTM"), which results in ≈ 122K review comments. Since RevCom leverages various structured information to recommend relevant review comments, it requires the file path and

[4]Accessed: October 12, 2022

Table 4.1: Statistics of the experimental dataset

PL	Repository	#PR	#Total Comments	#Reviewer Comments	#Filtered Reviewer Comments	#Review Comments for Python / Java Files
Python	Ansible	48371	59142	43080	31095	16608
	Keras	5777	6164	4347	3025	1729
	Django	16366	48280	35871	24644	13021
	Youtube-dl	4788	8723	6325	4514	4182
Java	Springboot	5500	3979	2598	1786	934
	Elasticsearch	61060	69303	47109	22930	9682
	Kafka	13062	97402	47109	32864	9437
	RxJava	3744	4689	3146	2000	475
	Total		**297,612**	**189,585**	**122,858**	**56,068**

library information from the source code. We thus eliminate the reviews that are related to documentation or other kinds of source files (e.g., .md or .rst files). In this step, we also carefully investigated a statistically significant sample of 383 comments (95% confidence level, 5% error margin) and found that none of the bot comments (41 in total) were associated with any source code. Since each valid comment should be connected to the source code, we removed all comments lacking an association to code, which essentially discarded all bot comments from our final dataset. Finally, our dataset contains $\approx$ 56K review comments, their associated *diff*, and source files. The summary statistics of our dataset are shown in Table 4.1.

Once we complete the filtration and have a refined dataset, we split it into corpus and query. Similar to earlier work [34], we keep 70% of the instances for corpus and the remaining as the query. If multiple comments are made against the same changed code of a diff, we consider these comments as separate instances. However, to handle the overlapped between the query and corpus set, we carefully keep the query instances different from the corpus instances during the dataset split.

4.4.2 Embedding Generation

In this work, we adopt Word2Vec [62] – a popular algorithm to generate word embedding for our experiment. There are quite a few pre-trained Word2Vec-based word embeddings available for reuse. However, these word embeddings are trained on natural language, which might not be able to capture semantics in the source code [73]. Therefore, we train a Word2Vec model using GitHub CodeSearchNet [38] dataset and use the word embedding for our analysis. CodeSearchNet contains $\approx$ 6M methods written in popular programming languages accompanied by natural language documentation.

The Out-of-vocabulary (OOV) issue is common in code-related task [47, 103, 90, 36] as source code contains not only typical API methods but also randomly-named tokens such as class names and variable names. Although word-level embeddings can represent the semantics of tokens in the source code, the OOV issue still exists since low-frequency words are discarded during the training of the Word2Vec model. To mitigate the OOV issue, we use a RoBERTa tokenizer [53]. This tokenizer leverages Byte-Pair Encoding (BPE) subword tokenization, which splits a word into a sequence of frequently occurring subwords [84]. Since the vocabulary contains all letters and common subwords, it can address the OOV issue. Moreover, prior studies also show that BPE also handles large vocabulary issues, which is a common concern in Natural Language Processing (NLP) for prediction [41].

4.4.3 Evaluation Metrics

To evaluate the performance of our proposed approach, we use three different metrics – BLEU score [69], perfect prediction, and semantic similarity [31]. These evaluation measures were also used by the relevant studies [101, 103, 34, 73, 50, 56], which justify our choice. *We report all metric scores in terms of percentage.* We define these metrics as follows.

4.4.3.1 Bi-Lingual Evaluation of Understanding (BLEU)

BLEU score [69] is a widely used textual similarity metric with significant use in the software engineering context [34, 103, 36, 101, 50, 56]. BLEU score calculates the similarity between the recommended reviews and the ground truth reviews in terms

of their n-gram precisions as follows.

$$BLEU = BP \cdot exp\left(\sum_{n=1}^{N} w_n \; log(p_n)\right) \tag{4.3}$$

Here, p_n is the ratio between overlapping n-grams (from both recommended and ground truth reviews) and the total number of n-grams in the recommended reviews, and w_n is the weight of the n-gram length. Following the existing studies [34, 50, 56], we use $N = 4$ and $w_n = 0.25$ for all n. The brevity penalty, BP, penalizes the recommended review comments that are too small and ensures a moderate length of comments.

4.4.3.2 Perfect Prediction (PP)

Perfect prediction measures the exact match between recommended review comments and ground truth review comments. Previous studies [34, 98] use perfect prediction to evaluate the performance of their code review recommendation approach. In our study, we use four different top-k candidates (i.e., k=1, 3, 5, 10) for code review recommendations. For a given changed code fragment, if one of the k-recommended review comments matches the ground truth reviews, we consider that our approach achieves the perfect prediction in the review recommendation.

4.4.3.3 Semantic Similarity (SS)

Although the BLEU score is a widely adopted metric for measuring textual similarity, it omits the semantic meaning of the text. For instance, the BLEU score considers "this is good" and "this is nice" as different 3-grams. Haque et al. [31] conduct a human study to identify which metric captures the perception of human raters the best. According to them, Sentence-BERT encoder [79] with cosine similarity has the highest correlation with the human-evaluated similarity. Therefore, we use `stsb-roberta-large`[5] pre-trained Sentence-BERT model to generate the embedding for the input text. We compute the semantic similarity between the recommended and ground truth reviews as follows.

$$SemSim(G, R) = cos(sbert(G), sbert(R)) \tag{4.4}$$

[5]https://bit.ly/3dR9mxD

Here, $sbert(X)$ is the numerical representation from Sentence-BERT for any input text X, G is the ground truth review, and R is the recommended review.

4.4.4 Baseline for Comparison

We compare the effectiveness of our approach with the state-of-the-art IR-based technique for code review recommendation – CommentFinder [34]. To replicate this technique, we use the replication package provided by the original author [34]. Given a changed method, CommentFinder recommends review comments based on method-level similarity. On the other hand, RevCom uses different structured information from the source code to recommend the relevant review comments. To make a fair comparison between these two techniques, we needed the changed methods associated with the review comments in our dataset. Similar to prior study [34], we thus extract the changed methods from the relevant source file. We found that $\approx 48\%$ of the review comments discuss the changes outside of a method. Thus, we keep the *changed methods* field empty for those comments, resulting in $\approx 52\%$ method-level code changes in our dataset.

We also compare the effectiveness of RevCom with the state-of-the-art DL-based technique for code review generation – CodeReviewer [50]. Given a code diff, CodeReviewer generates review comments relevant to the diff. To replicate this technique, we use the pre-trained model provided by the original author [50] and fine-tuned it using our dataset. We fine-tuned their model on NVIDIA V100 GPUs with 32GB of memory. We use the same hyper-parameter settings as provided in their replication package [50]. The average model training time is one day for the within-project settings and two days for cross-project settings.

4.5 Study Result

In this section, we discuss the experimental results and answer our research questions.

4.5.1 Answering RQ1 – Performance of RevCom:

Table 4.2 shows the performance of RevCom in terms of BLEU score, perfect prediction, and semantic similarity. We evaluate its performance based on four top-k values

Table 4.2: Performance of RevCom

PL	Repo	Top-1			Top-3			Top-5			Top-10		
		BLEU	PP	SS	BLEU	PP	SS	BLEU	PP	SS	BLEU	PP	SS
Python	Ansible	17.19	3.23	32.3	24.37	4.52	40.94	26.72	4.71	43.68	29.22	4.78	46.77
	Keras	14.19	3.47	28.22	20.4	4.62	37.25	23.24	4.82	40.79	26.54	5.2	44.51
	Django	12.57	1.11	31.01	18.93	1.48	39.89	21.49	1.66	43.02	24.49	1.95	46.6
	Youtube-dl	11.11	1.23	29.18	17.79	2.23	37.9	20.34	2.31	41.12	23.22	2.39	44.47
Java	RxJava	14.25	2.8	30.67	21.94	2.8	41.6	24.29	2.8	43.81	26.62	3.5	47.79
	Kafka	15.32	1.55	33.75	22.01	2.22	42.31	24.55	2.51	45.19	27.64	2.79	48.56
	Elasticsearch	13.86	1.11	31.43	20.29	1.48	40.33	22.58	1.51	43.51	25.21	1.55	46.84
	Springboot	20.21	4.63	31.65	25.8	4.98	36.68	27.79	4.98	41.31	30.08	4.98	44.38
	Average (%)	**14.84**	**2.39**	**31.03**	**21.44**	**3.04**	**39.61**	**23.88**	**3.16**	**42.80**	**26.63**	**3.39**	**46.24**

Table 4.3: Performance of RevCom in cross-project settings

PL	Dataset	Top-1			Top-3			Top-5			Top-10		
		BLEU	PP	SS	BLEU	PP	SS	BLEU	PP	SS	BLEU	PP	SS
Python	Cross-project	8.05	0.01	25.79	11.70	0.01	33.24	13.77	0.01	36.71	16.49	0.02	40.57
Java	Cross-project	8.56	0.00	25.95	14.14	0.03	35.15	16.71	0.03	38.62	19.55	0.03	42.42
	Average (%)	**8.30**	**0.003**	**25.87**	**12.92**	**0.004**	**34.19**	**15.24**	**0.01**	**37.67**	**18.02**	**0.01**	**41.49**

(k = 1, 3, 5, 10), where k is the number of recommended review comments for a given changed code fragment.

From Table 4.2, we see that RevCom achieves an average BLEU score of 14.84% when the top-k candidate is 1. Interestingly, for the top 10 candidates, the average BLEU score of the recommended reviews improves up to 26.63%, which is promising. According to Google's AutoML Translation documentation, such a BLEU score indicates that the review comments can deliver the actual intent of reviewers regarding the code change while containing minor grammatical issues.

Recommended review comments from RevCom also have a perfect prediction of 2.39% when the top-k candidate is 1. This score improves up to 3.39% when the top-k candidate is 10. Since RevCom recommends the review comments from the top 10 candidates, the improved perfect prediction might be explainable. Furthermore, we see that recommended review comments from RevCom have an average semantic similarity of 31.03% when the top-k candidate is 1. This score improves up to 46.24% when the best among the top-10 recommended reviews is considered. Such a semantic similarity score indicates that recommended review comments from Revcom have a

major semantic overlap with the actual review comments from the reviewer [56]. All these statistics are highly promising and demonstrate the potential of our approach in recommending relevant code review comments.

While our approach performs well for the within-project setting, we also evaluate the performance of RevCom in a cross-project setting. In the cross-project setting, we use the instances from three Java projects and three Python projects as the corpus and the remaining two projects for evaluation. To avoid any bias in this project selection, we apply a cross-validation approach and report the average performance for four different cross-validation results. From Table 4.3, we see that even though the performance of RevCom decreases in the cross-project setting, it is still promising, especially in terms of the semantic similarity metric. For the top 1 candidate, RevCom achieves an average BLEU score of 8.30%, which is $\approx$ 44% lower than the within-project setting. According to existing literature [56, 91], a performance drop in the cross-project setting is expected. However, we see interesting results in the case of the semantic similarity score. That is, for the top 1 candidate, recommended review comments from RevCom achieve a semantic similarity score of 25.87% in the cross-project setting. Even though it is $\approx$ 17% lower than the within-project setting, this drop is not as significant as the BLEU score. Such findings indicate that recommended review comments from RevCom might express similar information but with different words in the cross-project setting.

To verify this case, we manually compare 100 randomly sampled recommended reviews from RevCom (cross-project setting) with the ground truth reviews. The first and second authors annotate each pair as one of similar, partially similar and dissimilar categories. We also perform an agreement analysis and find an almost perfect agreement (0.95 kappa value) between the two annotators. Then, the first and second authors sit together and resolve the disagreement through discussions. We find that 16% of review pairs are semantically similar, while 37% are partially similar. Thus, 53% of recommended reviews from RevCom discuss the same changes with different phrases, which might cause the BLEU score to be low. For instance, for a particular code change, RevCom recommends – *"We would like to avoid wildcard import in the code base."*, whereas the ground truth is *"Please don't use star imports."*. Although the recommended review comment and ground truth review comment suggest the

same change, they have a semantic similarity of 0.51 and their BLEU score is only 0.16. Such a phenomenon might explain the low BLEU score and comparatively high semantic similarity score for the cross-project setting of RevCom.

Summary of RQ_1: RevCom can recommend reviews that can express the intent of the reviewers regarding the code change. It also shows promising results in terms of three evaluation metrics. Interestingly, it maintains a promising semantic similarity score even in the cross-project setting.

4.5.2 Answering RQ2 – Role of structured information in RevCom:

In this experiment, we analyze the impact of structured information from source code on review comment recommendations. First, we evaluate the performance of RevCom with each of three structured items – *changed code fragment*, library information and file path. We then combine these structured items and evaluate the performance of RevCom. Such an experiment helps us understand the contribution of individual structured items toward RevCom.

We first use only the file path as input for RevCom. From Table 4.4, we see that the average BLEU score, perfect prediction and semantic similarity of RevCom reduce by 24.80%, 70.29%, and 13.27%, respectively, when the top-k candidate is 1. The performance of RevCom also drops when the top 3, 5, and 10 results are analyzed. Since the file path only contains the name and path of the source file rather than any code change, the low performance of RevCom with the file path might be explainable.

We further evaluate the performance of RevCom using only the library information as input. From Table 4.4, we see that the average BLEU score, perfect prediction, and semantic similarity of RevCom reduce by 15.90%, 49.79%, and 2.15%, respectively, while recommending reviews from the top 1 candidate. We also observe a performance drop of RevCom when the top 3, 5, and 10 candidates are used for review recommendations. Interestingly, library information improves the performance by 11.83%, 69.01%, and 5.78% compared to file paths in terms of BLEU score, perfect prediction, and semantic similarity, respectively, when the top-k candidate is 1. Library information comprises import statements that capture important code tokens (e.g., class names, library names) relevant to a source file, whereas the file path

Table 4.4: Role of structured information in Revcom

Approach	Structured Information	Top-1			Top-3			Top-5			Top-10		
		BLEU	**PP**	**SS**	**BLEU**	**PP**	**SS**	**BLEU**	**PP**	**SS**	**BLEU**	**PP**	**SS**
RevCom	file	11.16	0.71	28.70	17.87	1.18	37.32	20.86	1.46	40.74	24.34	1.93	43.96
	library	12.48	1.2	30.36	19.43	1.76	38.04	22.21	2.20	41.05	25.49	2.80	44.94
	changed code	13.96	2.10	30.41	20.75	2.66	38.85	23.27	2.80	41.72	26.36	3.18	46.05
	all (%)	**14.84**	**2.39**	**31.03**	**21.44**	**3.04**	**39.61**	**23.88**	**3.16**	**42.80**	**26.63**	**3.39**	**46.24**

contains only the name and path of the file. Thus library information might contain more salient information, and hence, the higher performance might be explainable.

Finally, we evaluate the performance of RevCom using changed code fragments from *diff* as input. From Table 4.4, we see that the average BLEU score, perfect prediction, and semantic similarity of RevCom reduce only by 5.92%, 12.13%, and 2.01%, respectively, for the top-1 candidate in the review recommendation. We also observe a similar performance drop of RevCom when the top 3, 5, and 10 candidates are used for recommendation. Interestingly, the metrics score from only using changed code fragments is significantly closer to RevCom than that of only file path or library information. Li et al. [50] show that *diff* can help better understand the structure of the code changes. It also contains more information about the changed code than file path or library information. Therefore, a performance close to RevCom by using only changed code fragments from *diff* might be explainable.

In summary, we see different amounts of performance drops in RevCom while evaluating separately with file paths, library information or changed code fragments. However, our approach performs best when all the structured items are combined.

Summary of RQ_2: Structured items have a major contribution to the performance RevCom. Among them, changed code from *diff* contributes the most to the performance of RevCom. Furthermore, they are most effective when all three items are used together.

Table 4.5: Role of different vectorization in Revcom

PL	Approach	Vectorization Technique	Top-1			Top-3			Top-5			Top-10		
			BLEU	PP	SS	BLEU	PP	SS	BLEU	PP	SS	BLEU	PP	SS
Python	RevCom	TF-IDF	13.77	2.26	30.18	20.37	3.21	39.01	22.95	3.38	42.15	25.87	3.58	45.59
		word2vec + Cosine	14.28	2.44	30.98	20.95	3.45	40.27	23.92	3.58	43.17	26.89	3.88	47.83
		Improvement (%)	**3.70**	**7.96**	**2.65**	**2.85**	**7.48**	**3.23**	**4.23**	**5.92**	**2.42**	**3.94**	**8.38**	**4.91**
Java		TF-IDF	15.91	2.52	31.88	22.51	2.87	40.23	24.81	2.95	43.46	27.39	3.21	46.89
		word2vec + Cosine	16.52	2.76	32.89	23.38	3.12	41.59	25.89	3.19	44.63	28.15	3.46	48.96
		Improvement (%)	**3.83**	**9.52**	**3.17**	**3.86**	**8.71**	**3.38**	**4.35**	**8.14**	**2.69**	**2.77**	**7.79**	**4.41**

4.5.3 Answering RQ3 – Role of different vectorization techniques in RevCom:

In this experiment, we analyze the impact of different vectorization techniques in recommending review comments. In particular, we evaluate the performance of RevCom with two vectorization techniques – TF-IDF and word embedding. To generate the word embedding, we use a popular technique named Word2Vec [62], and train it on GitHub CodeSearchNet [38].

From Table 4.5, in the context of Python-based projects, we see that the word embedding improves the average BLEU score, perfect prediction, and semantic similarity of RevCom by 3.70%, 7.96%, and 2.65%, respectively, when the top-k candidate is 1. We also observe performance improvement when the top 3, 5, and 10 candidates are used for review comment recommendations. According to Shapiro-Wilk test [85], the metrics score obtained from RevCom is normally distributed. We thus perform paired Student's t-test [14] to determine the performance difference of RevCom using two different vectorization techniques. According to this test, this improvement is not statistically significant ($p = 0.84 < 0.05$). Similarly, for Java-based projects, word embedding improves the performance of RevCom by 3.83%, 9.52%, and 3.17% in terms of BLEU score, perfect prediction, and semantic similarity, respectively, when the top-k candidate is 1. The performance of RevCom also improves when the top 3, 5, and 10 candidates are analyzed for review recommendations. However, according to statistical test, this improvement is not statistically significant ($p = 0.76 < 0.05$). According to the existing study [88], word embedding can capture the semantics in the changed code effectively. We also use subword tokenization [84] to overcome the OOV issues of word embedding. Thus, the performance improvement of RevCom

Table 4.6: Performance comparison with the baselines

Approach	Top-1			Top-3			Top-5			Top-10		
	BLEU	**PP**	**SS**	**BLEU**	**PP**	**SS**	**BLEU**	**PP**	**SS**	**BLEU**	**PP**	**SS**
CommentFinder	9.93	1.77	26.12	17.25	1.77	37.24	20.08	1.87	39.77	22.85	2.17	43.05
RevCom	14.84	2.39	31.03	21.44	3.04	39.61	23.88	3.16	42.80	26.63	3.39	46.24
Improvement (%)	**49.45**	**35.03**	**18.80**	**24.29**	**71.75**	**6.36**	**18.92**	**68.98**	**7.62**	**16.54**	**56.22**	**7.41**
Code Reviewer	13.95	2.17	29.87	17.35	2.66	35.86	19.97	3.01	39.35	22.94	3.26	43.18
RevCom	14.84	2.39	31.03	21.44	3.04	39.61	23.88	3.16	42.80	26.63	3.39	46.24
Improvement (%)	**6.38**	**10.14**	**3.88**	**23.57**	**14.29**	**10.46**	**19.58**	**4.98**	**8.77**	**16.09**	**3.99**	**7.09**

with word embedding might be explainable.

However, word embedding-based vectorization requires 5X time compared to TF-IDF vectorizer to calculate the similarity between the query vectors and corpus vectors. Furthermore, it requires domain-specific training to extract the meaningful embedding for any analysis. Thus, in terms of cost-benefit analysis, word embedding-based vectorization might not be a cost-effective choice. Since the goal of RevCom is to reduce the time and effort for both reviewers and the changed code submitter, we keep the TF-IDF vectorizer as the default vectorization technique in RevCom, given the above analysis. However, as demonstrated above, our approach has the potential to perform even better with more sophisticated vectors.

Summary of RQ_3: Word embedding-based vectorization can improve the performance of RevCom. However, it requires 5X time compared to the TF-IDF vectorizer, which makes it a costly choice for our approach.

4.5.4 Answering RQ4 – Comparison with the existing baselines:

In this research question, we compare RevCom with existing techniques from the literature and investigate whether RevCom can outperform them in terms of various evaluation metrics. To the best of our knowledge, CommenFinder [34] is the only IR-based technique to recommend review comments, whereas CodeReviewer [50] is the state-of-the-art DL-based technique to generate review comments. We thus compare RevCom with two baselines CommentFinder and CodeReviewer in our experiment.

Table 4.7: Performance comparison with the baselines in cross-project settings

Approach	Top-1			Top-3			Top-5			Top-10		
	BLEU	**PP**	**SS**	**BLEU**	**PP**	**SS**	**BLEU**	**PP**	**SS**	**BLEU**	**PP**	**SS**
CommentFinder	6.11	0.00	24.43	12.72	0.00	33.07	14.88	0.00	36.3	17.86	0.00	40.74
RevCom	8.30	0.00	25.87	12.92	0.00	34.19	15.24	0.01	37.67	18.02	0.01	41.49
Improvement (%)	**35.84**	**0.00**	**5.89**	**1.57**	**0.00**	**3.39**	**2.42**	**0.00**	**3.77**	**0.90**	**0.00**	**1.84**
Code Reviewer	10.76	0.01	29.62	12.86	0.01	34.11	14.19	0.02	36.77	16.46	0.03	40.68
RevCom	8.33	0.00	25.87	12.92	0.00	34.19	15.24	0.01	37.67	18.02	0.01	41.49
Improvement (%)	**0.00**	**0.00**	**0.00**	**0.47**	**0.00**	**0.23**	**7.40**	**0.00**	**2.45**	**9.48**	**0.00**	**1.99**

Table 4.6 shows the comparison between RevCom and two baselines in terms of BLEU score, perfect prediction, and semantic similarity. We find that RevCom outperforms CommentFinder in BLEU score, perfect prediction and semantic similarity with 16.54% – 49.45%, 35.03% – 71.75% and 6.36% – 18.80% margins, respectively, when top 1, 3, 5, and 10 candidates are used for review comment recommendation. According to Shapiro-Wilk test [85], the metrics score obtained from each technique is normally distributed. We thus perform paired Student's t-test [14] to understand the performance difference between any two techniques. According to this test, RevCom performs significantly higher than CommentFinder, i.e., $p\text{-}value = 5.17e{-}06 < 0.05$, Cohen's $d = 0.93$ (*large*) for all three metrics. The dataset for evaluating the baseline contains $\approx$ 48% non-method-level changes (see Section 4.4.4). Since RevCom leverages different structured information from the diff, library, and file path information, the improved performance of our approach might be explainable.

Furthermore, From Table 4.6, we see that RevCom outperforms the DL-based baseline – CodeReviewer in BLEU score, perfect prediction and semantic similarity with 6.38% – 23.57%, 3.99% – 14.29% and 3.88% – 10.46% margins respectively when top 1, 3, 5, and 10 candidates are used for review comment recommendation. According to Student's t-test, the performance of RevCom is also significantly higher than CodeReviewer, i.e., $p\text{-}value = 0.001 < 0.05$, Cohen's $d = 0.64$ (*medium*) for all three metrics. While generative AI models can generate new review comments, IR-based approaches recommend from an existing corpus. Given an unseen code change, the IR-based technique searches for a similar code change in the corpus and recommends the corresponding comments as review comments. On the other hand, generative AI

models generate review comments based on their training data. However, we find that a structured information retrieval technique can outperform existing deep-learning models. This further confirms the finding of Hong et al. [34] that IR-based techniques perform better than DL-based techniques in recommending code review comments.

Although existing studies do not evaluate the performance of our selected baseline techniques in a cross-project setting, we compare RevCom with the baselines using a cross-project setting. In the cross-project setting, we use the instances from three Java projects and three Python projects as the corpus and the remaining two projects for evaluation. To avoid any bias in this project selection, we apply a cross-validation approach and report the average performance for four different cross-validation results. From Table 4.7, we see that the performance of RevCom, CommentFinder and CodeReviewer drops significantly in the cross-project setting. However, RevCom outperforms CommenFinder in BLEU score and semantic similarity with 0.90% – 35.84% and 1.84% – 5.89% margins, respectively, when top 1, 3, 5, and 10 candidates are used for review comment recommendation. On the other hand, RevCom also outperforms CodeReviewer in BLEU score and semantic similarity with 0.47% – 9.48% and 0.23% – 2.45% margins, respectively, when top 3, 5, and 10 candidates are used for review comment recommendation. However, we see no improvement in RevCom over CodeReviewer when the top-1 candidates are used for review comment recommendations. According to statistical tests, the performance of RevCom is also statistically significant compared to the baseline techniques. For CommentFinder, $p = 0.007 < 0.05$ and Cohen's $d = 0.31$ (*small*), where for CodeReviewer, $p = 0.02 < 0.05$ and Cohen's $d = 0.21$ (*small*).

Although the performance of RevCom and baselines drops in the cross-project setting, RevCom still shows better performance than the baselines in terms of two similarity metrics. Thus, our idea of leveraging structured information has high potential in recommending relevant code review comments.

Summary of RQ$_4$: RevCom outperforms the IR-based and DL-based baselines by up to 49.45% and 23.57% margins in BLEU score respectively in within-project settings. In cross-project settings, it also outperforms both baselines by up to 35.84% and 9.48% margins in the BLEU score.

4.6 Related Work

Review comments are one of the main building blocks of modern code reviews (MCR). Prior studies found that review comments are beneficial for finding software defects or designing impactful changes in the source code [7, 20, 58, 99]. Existing studies on automated code review focus on code review generation and code review recommendation.

Review Generation: Several existing approaches [98, 50] use Neural Machine Translation (NMT) to generate code review comments. Tufano et al. [98] pre-trained a Transformer model [100] for automating the code review activities. However, they did not integrate the changed code into the pre-trained model. Furthermore, their pre-training data is not directly related to code reviews. Recently, Li et al. [50] propose CodeReviewer – a pre-train encoder-decoder transformer model to automate different code review activities. Unlike the previous work, CodeReviewer is pre-trained on a large code review dataset, consisting of *diff* hunks and review comments. Although existing approaches show the potential of generating code reviews using NMT, they require specialized computing resources (e.g., 16 × 40GB GPU [101]) and long computing time (e.g., 12 days of training time [101]), which might not always be available. Different from these approaches, we propose an IR-based approach to recommend relevant code reviews for any code change using structured information retrieval. We consider CodeReviewer as a baseline and compare it with RevCom using experiments. Please consult Section 4.5–RQ4 for a detailed comparison between the two techniques.

Review Recommendation: Prior approaches use both deep learning and information retrieval (IR) based techniques to recommend relevant review comments for a given code change. Gupta and Sundaresan [29] propose the LSTM-based model DeepMem, which recommends review comments based on existing code reviews and code changes. Siow et al. [88] employs an attention-based deep neural network that captures the semantics from both source code and reviews to recommend relevant review comments. They represent the semantics in the source code and review comments using multi-level word embedding. They show that their technique can mitigate the Out-of-vocabulary problem [47, 103, 36]. Recently, Hong et al. [34] propose an IR-based technique – CommentFinder, which recommends the review comments based on the method-level similarity. However, their approach overlooks the structured information in code change and recommends reviews only for the method-level change.

Their work serves as our baseline, and we compare our work with theirs experimentally (Table 4.6, RQ4). Unlike the prior approach, our technique, RevCom, leverages various code-level changes (i.e., both method and non-method-level changes) and performs structured information retrieval to recommend relevant code review comments, which makes our work novel.

4.7 Threats To Validity

Threats to *internal validity* relate to experimental errors and biases [95]. Replication of the existing baseline technique could pose a threat. However, we use the replication package provided by the original author of CommentFinder [34] and CodeReviewer [50]. Thus, threats related to replication might be mitigated.

The quality of the dataset and the change granularity could impact the result of RevCom. However, we follow an existing work [96] to perform rigorous data cleaning and filtering (e.g., removal of author comments) to mitigate the noise in the dataset. Furthermore, to capture various changes in the source code, RevCom leverages different structured information and can recommend relevant code reviews for both method-level and non-method-level changes. Thus, threats related to datasets and change granularity might be mitigated.

Finally, the threat to *external validity* relates to the generalizability of our work [74]. To mitigate this threat, we evaluate RevCom using the code changes from both Java-based and Python-based projects. As we see from Table 4.2 and 4.3, the performance of RevCom does not vary significantly between Python-based and Java-based projects. Furthermore, we evaluate the performance using both within-project and cross-project settings. Thus threat to external validity might be mitigated.

4.8 Summary

While our previous study (Chapter 3) focuses on assessing various facets (e.g., sentiments, usefulness) of code review comments, in Chapter 4, we propose RevCom, a novel technique that uses structured information retrieval to recommend relevant review comments. Our technique leverages different structured information and can recommend relevant reviews for both method-level and non-method-level changes.

We evaluate our technique using eight (four Python + four Java) projects and three popular metrics (i.e., BLEU score, perfect prediction, and semantic similarity), where our technique outperforms both IR-based and DL-based baselines by up to 49.45% and 23.57% margins in BLEU score respectively.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Review comments play a significant role in modern code reviews. It is crucial to ensure the quality of review comments. However, manually writing high-quality review comments could be technically challenging and time-consuming. A number of existing studies [3, 22, 16, 19, 76, 32, 88, 34] use automated tools and techniques to assess and recommend code review comments, but they could be limited in several aspects. First, according to an existing work [42], various development practices including code reviews could be different between open-source and closed-source systems, but only a little research has been done to investigate how existing techniques perform differently when assessing review comments from these two types of systems. Second, existing review generation or recommendation techniques either require specialized resources or suffer from a lack of generalizability. In this thesis, we attempt to address the above two challenges by empirically assessing and recommending code review comments. We conduct two separate but complementary studies (Chapter 3 and Chapter 4), and we have the following outcomes.

- The first study (Chapter 3) empirically assesses the performance of seven existing tools and techniques in measuring two quality aspects – *sentiments* and *usefulness* – of review comments and then contrasts their performance between open and closed-source systems. We find that the performance of these automated tools significantly differs between open and closed-source systems. We also observe that reviewers from these two types of systems have varying levels of reviewing experience, which might have led to the varying prevalence of sentiments and usefulness in their code review comments. Since existing tools and techniques were trained on these comments, their sentiment and usefulness detection performance thus might also have been affected.

- The second study (Chapter 4) proposes a novel technique that recommends relevant review comments using structured information retrieval. We evaluate our technique using eight (four Python + four Java) projects and three popular metrics (i.e., BLEU score, perfect prediction, and semantic similarity), where our technique outperforms both IR-based and DL-based baselines by achieving up to 49.45% and 23.57% higher BLEU scores respectively. Our technique leverages different structural information (e.g., diff, library information) and can recommend relevant reviews for both method-level and non-method-level changes where existing IR-based technique [34] might fall short.

5.2 Future Work

Given our conducted studies and findings, there are several potential directions for future work. We discuss them in detail as follows.

5.2.1 Code Review Assessment

The finding from our first study suggests that the performance of existing tools and techniques can vary significantly when assessing the code review comments from open-source and closed-source systems. In future, we will focus on designing novel tools and techniques that are (a) customizable for open-source and closed-source code reviews, (b) can recommend reviewers capable of writing useful code review comments, (c) can leverage cost-effective features (e.g., sentiments) instead of costly metric (e.g., review history) for detecting useful review comments.

5.2.2 Code Review Recommendation

In our second study, we found that structured information in the source code plays a significant role in recommending code review comments. In future, we plan to investigate the following potential research problems.

- How to encode different structured information from source code in a more compact and efficient manner? The idea is to keep the approach lightweight while capturing high-quality embeddings representing the semantics of code-level changes.

- Design a hybrid code review recommendation system by combining both information retrieval and deep learning techniques. First, train a model using pairs of changed code and review comments. The model's task is to predict whether a given code snippet is relevant to a review comment. Then, the recommender system can use information retrieval to quickly retrieve a pool of candidate code changes and use the deep learning model to further rank and refine the recommendations.

Bibliography

[1] Shaza M Abd Elrahman and Ajith Abraham. A review of class imbalance problem. *Journal of Network and Innovative Computing*, 1(2013):332–340, 2013.

[2] Basant Agarwal and Namita Mittal. Machine learning approach for sentiment analysis. In *Prominent feature extraction for sentiment analysis*, pages 21–45. Springer, 2016.

[3] Toufique Ahmed, Amiangshu Bosu, Anindya Iqbal, and Shahram Rahimi. Senticr: a customized sentiment analysis tool for code review interactions. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 106–111. IEEE, 2017.

[4] Akiko Aizawa. An information-theoretic perspective of tf–idf measures. *Information Processing & Management*, 39(1):45–65, 2003. ISSN 0306-4573. doi: https://doi.org/10.1016/S0306-4573(02)00021-3. URL `https://www.sciencedirect.com/science/article/pii/S0306457302000213`.

[5] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015.

[6] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100. Proceedings of Machine Learning Research, 2016.

[7] 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.

[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[9] 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*, pages 931–940. IEEE, 2013.

[10] Victor R Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[11] Tobias Baum and Kurt Schneider. On the need for a new generation of code review tools. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17*, pages 301–308. Springer, 2016.

[12] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. The influence of non-technical factors on code review. In *2013 20th working conference on reverse engineering (WCRE)*, pages 122–131. IEEE, 2013.

[13] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th working conference on Mining Software Repositories (MSR)*, pages 202–211, 2014.

[14] R Clifford Blair and James J Higgins. Comparison of the power of the paired samples t test to that of Wilcoxon's signed-ranks test under various population shapes. *Psychological Bulletin*, 97(1):119, 1985.

[15] 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.

[16] Fabio Calefato, Filippo Lanubile, Federico Maiorano, and Nicole Novielli. Sentiment polarity detection for software development. *Empirical Software Engineering*, 23(3):1352–1382, 2018.

[17] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference*, pages 964–974, 2019.

[18] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[19] Zhenpeng Chen, Yanbin Cao, Xuan Lu, Qiaozhu Mei, and Xuanzhe Liu. Sentimoji: an emoji-powered learning approach for sentiment analysis in software engineering. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 841–852, 2019.

[20] Chun Yong Chong, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. Assessing the students' understanding and their mistakes in code review checklists: an experience report of 1,791 code review checklist questions from 394 students. In *2021 IEEE/ACM 43rd International Conference on Software Engineering-SEET*, pages 20–29. IEEE, 2021.

[21] Vasiliki Efstathiou and Diomidis Spinellis. Code review comments: language matters. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 69–72, 2018.

[22] Ikram El Asri, Noureddine Kerzazi, Gias Uddin, Foutse Khomh, and MA Janati Idrissi. An empirical study of sentiments in code reviews. *Information and Software Technology*, 114:37–54, 2019.

[23] Bjarke Felbo, Alan Mislove, Anders Søgaard, Iyad Rahwan, and Sune Lehmann. Using millions of emoji occurrences to learn any-domain representations for detecting sentiment, emotion and sarcasm. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2017. doi: 10.18653/v1/d17-1169. URL `https://doi.org/10.18653%2Fv1%2Fd17-1169`.

[24] Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[25] Todd Michael Franke, Timothy Ho, and Christina A Christie. The chi-square test: Often used and more often misinterpreted. *American Journal of Evaluation*, 33(3):448–458, 2012.

[26] Wei Fu and Tim Menzies. Easy over hard: A case study on deep learning. In *Proceedings of the 2017 11th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 49–60, 2017.

[27] Salvador García, Alberto Fernández, Julián Luengo, and Francisco Herrera. A study of statistical techniques and performance measures for genetics-based machine learning: accuracy and interpretability. *Soft Computing*, 13(10):959–977, 2009.

[28] Github. Graphql api, mar 2022. URL `https://docs.github.com/en/graphql`.

[29] 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.

[30] Christoph Hannebauer, Michael Patalas, Sebastian Stünkel, and Volker Gruhn. Automatically recommending code reviewers based on their expertise: An empirical comparison. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 99–110, 2016.

[31] Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. Semantic similarity metrics for evaluating source code summarization. *2022 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, 2022.

[32] Masum Hasan, Anindya Iqbal, Mohammad Rafid Ul Islam, AJM Rahman, and Amiangshu Bosu. Using a balanced scorecard to identify opportunities to improve code review effectiveness: an industrial experience report. *Empirical Software Engineering*, 26(6):1–34, 2021.

[33] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 763–773, 2017.

[34] 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 (ESEC/FSE)*, pages 507–519.

[35] Yang Hong, Chakkrit Kla Tantithamthavorn, and Patanamon Pick Thongtanunam. Where should i look at? recommending lines that reviewers should pay attention to. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 1034–1045. IEEE, 2022.

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

[37] Syed Fatiul Huq, Ali Zafar Sadiq, and Kazi Sakib. Understanding the effect of developer sentiment on fix-inducing changes: An exploratory study on github pull requests. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 514–521. IEEE, 2019.

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

[39] Md Rakibul Islam and Minhaz F Zibran. Sentistrength-se: Exploiting domain specificity for improved sentiment analysis in software engineering text. *Journal of Systems and Software*, 145:125–146, 2018.

[40] Robbert Jongeling, Proshanta Sarkar, Subhajit Datta, and Alexander Serebrenik. On negative results when using sentiment analysis tools for software engineering research. *Empirical Software Engineering*, 22(5):2543–2584, 2017.

[41] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering*, pages 1073–1085. IEEE, 2020.

[42] Atieh Khanjani and Riza Sulaiman. The aspects of choosing open source versus closed source. In *2011 IEEE Symposium on Computers & Informatics*, pages 646–649. IEEE, 2011.

[43] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. Investigating code review quality: Do people and participation matter? In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 111–120. IEEE, 2015.

[44] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. Code review quality: How developers see it. In *Proc. International Conference on Software Engineering*, pages 1028–1038, 2016.

[45] Diana Kornbrot. Point biserial correlation. *Wiley StatsRef: Statistics Reference Online*, 2014.

[46] Andrey Krutauz, Tapajit Dey, Peter C Rigby, and Audris Mockus. Do code review measures explain the incidence of post-release defects? *Empirical Software Engineering*, 25(5):3323–3356, 2020.

[47] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, jul 2018. doi: 10.24963/ijcai.2018/578. URL `https://doi.org/10.24963%2Fijcai.2018%2F578`.

[48] Ruiyin Li, Peng Liang, and Paris Avgeriou. Code reviewer recommendation for architecture violations: An exploratory study. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. ACM, jun 2023. doi: 10.1145/3593434.3593450. URL `https://doi.org/10.1145%2F3593434.3593450`.

[49] 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.

[50] 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 (ESEC/FSE)*, pages 1035–1047, 2022.

[51] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Codereviewer: Pre-training for automating code review activities. *arXiv preprint arXiv:2203.09095*, 2022.

[52] Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, Michele Lanza, and Rocco Oliveto. Sentiment analysis for software engineering: How far can we go? In *Proceedings of the 40th international conference on software engineering*, pages 94–104, 2018.

[53] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[54] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: how far are we? In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, pages 373–384, 2018.

[55] Alvi Mahadi, Karan Tongay, and Neil A Ernst. Cross-dataset design discussion mining. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 149–160. IEEE, 2020.

[56] Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. Explaining software bugs leveraging code structures in neural machine translation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 640–652. IEEE, 2023.

[57] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[58] Mika V Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3): 430–448, 2008.

[59] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

[60] Tim Menzies, Suvodeep Majumder, Nikhila Balaji, Katie Brey, and Wei Fu. 500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow). In *2018 IEEE/ACM 15th Mining Software Repositories (MSR)*, pages 554–563. IEEE, 2018.

[61] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[62] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[63] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pages 171–180. IEEE, 2015.

[64] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, pages 181–190, 2008.

[65] Alessandro Murgia, Parastou Tourani, Bram Adams, and Marco Ortu. Do developers feel emotions? an exploratory analysis of emotions in software artifacts. In *Proceedings of the 11th working conference on mining software repositories*, pages 262–271, 2014.

[66] Nicole Novielli, Daniela Girardi, and Filippo Lanubile. A benchmark study on sentiment analysis for software engineering research. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 364–375. IEEE, 2018.

[67] Nicole Novielli, Fabio Calefato, Davide Dongiovanni, Daniela Girardi, and Filippo Lanubile. Can we use se-specific sentiment analysis tools in a cross-platform setting? In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 158–168, 2020.

[68] Nicole Novielli, Fabio Calefato, Filippo Lanubile, and Alexander Serebrenik. Assessment of off-the-shelf se-specific sentiment analysis tools: An extended replication study. *Empirical Software Engineering*, 26(4):1–29, 2021.

[69] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proc.40th The Association for Computational Linguistics (ACL)*, pages 311–318, 2002.

[70] Rajshakhar Paul, Amiangshu Bosu, and Kazi Zakia Sultana. Expressions of sentiments during code reviews: Male vs. female. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 26–37. IEEE, 2019.

[71] John W. Pratt. Remarks on zeros and ties in the wilcoxon signed rank procedures. *Journal of the American Statistical Association*, 54(287):655–667, 1959. doi: 10.1080/01621459.1959.10501526.

[72] Wisam A Qader, Musa M Ameen, and Bilal I Ahmed. An overview of bag of words; importance, implementation, applications, and challenges. In *2019 International Engineering Conference (IEC)*, pages 200–204. IEEE, 2019.

[73] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21:1–67, 2020.

[74] Mohammad M Rahman, Chanchal K Roy, and David Lo. Automatic query reformulation for code search using crowdsourced knowledge. *Empirical Software Engineering*, 24(4):1869–1924, 2019.

[75] Mohammad Masudur Rahman, Chanchal K Roy, and Jason A Collins. Correct: code reviewer recommendation in github based on cross-project and technology experience. In *Proceedings of the 38th international conference on software engineering companion*, pages 222–231, 2016.

[76] 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.

[77] Mohammad Masudur Rahman et al. *Supporting Source Code Search with Context-Aware and Semantics-Driven Query Reformulation*. PhD thesis, University of Saskatchewan, 2019.

[78] Musfiqur Rahman, Dharani Palani, and Peter C Rigby. Natural software revisited. In *2019 IEEE/ACM 41st International Conference on Software Engineering*, pages 37–48. IEEE, 2019.

[79] Nils Reimers, Iryna Gurevych, Nils Reimers, Iryna Gurevych, Nandan Thakur, Nils Reimers, Johannes Daxenberger, Iryna Gurevych, Nils Reimers, Iryna Gurevych, et al. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proc.2019 Empirical Methods in Natural Language Processing*, pages 671–688. The Association for Computational Linguistics (ACL), 2019.

[80] Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 9th The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 202–212, 2013.

[81] Stephen Robertson, Hugo Zaragoza, et al. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

[82] 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.

[83] Klaus R Scherer, Tanja Wranik, Janique Sangsue, Véronique Tran, and Ursula Scherer. Emotions in everyday life: Probability of occurrence, risk factors, appraisal and reaction patterns. *Social Science Information*, 43(4):499–570, 2004.

[84] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *CoRR*, abs/1508.07909, 2015. URL `http://arxiv.org/abs/1508.07909`.

[85] Samuel S Shapiro, Martin B Wilk, and Hwei J Chen. A comparative study of various tests for normality. *Journal of the American statistical association*, 63 (324):1343–1372, 1968.

[86] Phillip Shaver, Judith Schwartz, Donald Kirson, and Cary O'connor. Emotion knowledge: further exploration of a prototype approach. *Journal of personality and social psychology*, 52(6):1061, 1987.

[87] Forrest J Shull, Jeffrey C Carver, Sira Vegas, and Natalia Juristo. The role of replications in empirical software engineering. *Empirical software engineering*, 13(2):211–218, 2008.

[88] 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.

[89] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[90] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2727–2735, 2019.

[91] Wei Tao, Yanlin Wang, Ensheng Shi, Lun Du, Shi Han, Hongyu Zhang, Dongmei Zhang, and Wenqiang Zhang. On the evaluation of commit message generation models: an experimental study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 126–136. IEEE, 2021.

[92] Mike Thelwall, Kevan Buckley, Georgios Paltoglou, Di Cai, and Arvid Kappas. Sentiment strength detection in short informal text. *Journal of the American society for information science and technology*, 61(12):2544–2558, 2010.

[93] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)*, pages 168–179. IEEE, 2015.

[94] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150. IEEE, 2015.

[95] Yuan Tian, David Lo, and Julia Lawall. Automated construction of a software-specific word similarity database. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 44–53. IEEE, 2014.

[96] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering*, pages 25–36. IEEE, 2019.

[97] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 163–174. IEEE, 2021.

[98] 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.

[99] 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 Mining Software Repositories (MSR)*, pages 471–482. IEEE, 2021.

[100] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Conference on Neural Information Processing Systems*, 30, 2017.

[101] 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 *Proc.2021 Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021.

[102] Theresa Wilson. Annotating subjective content in meetings. In *The International Conference on Language Resources and Evaluation*. Citeseer, 2008.

[103] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. Commit message generation for source code changes. In *The International Joint Conference on Artificial Intelligence*, 2019.

[104] Cheng Yang, Xun-hui Zhang, Ling-bin Zeng, Qiang Fan, Tao Wang, Yue Yu, Gang Yin, and Huai-min Wang. Revrec: A two-layer reviewer recommendation algorithm in pull-based development model. *Journal of Central South University*, 25(5):1129–1143, 2018.

[105] Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. Mining the modern code review repositories: A dataset of people, process and product. In *Proceedings of the 13th Mining Software Repositories (MSR)*, pages 460–463, 2016.

[106] 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.

Appendix A

Complementary Materials

A.1 Replication Package

A.1.1 Code Review Assessment

GithHub Repository: https://github.com/wahid-shuvo/Code-Review-Assessment

A.1.1.1 Sentiment Annotation

Sentiment Annotation tutorial- https://bit.ly/3qaakvx
Sentiment Annotation guidelines- https://bit.ly/3OeO77B

A.1.2 Code Review Recommendation

GithHub Repository: https://github.com/wahid-shuvo/RevCom-Replication

Appendix B

Copyright Permission Letters

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

Recommending Code Reviews Leveraging Code Changes with Structured Information Retrieval

Ohiduzzaman Shuvo, Parvez Mahbub, Mohammad Masudur Rahman

2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the IEEE PSPB Operations Manual.
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall be null and void. In this case, IEEE will retain a copy of the manuscript for internal administrative/record-keeping purposes.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

You have indicated that you DO wish to have video/audio recordings made of your conference presentation under terms and conditions set forth in "Consent and Release."

CONSENT AND RELEASE

1. In the event the author makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the author, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction of the Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast of the Presentation during or after the conference.
2. In connection with the permission granted in Section 1, the author hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES

YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

ohiduzzaman shuvo — **Signature**

31-07-2023 — **Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- Personal Servers. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- Classroom or Internal Training Use. An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- Electronic Preprints. Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the IEEE work with a Digital Object Identifier (DOI) or link to the article abstract in IEEE Xplore, or (2) the accepted version only (not the IEEE-published version), including the IEEE copyright notice and full citation, with a link to