EasyChair Preprint

№ 10595

Recommending Code Reviews Leveraging Code Changes with Structured Information Retrieval

Ohiduzzaman Shuvo, Parvez Mahbub and
Mohammad Masudur Rahman

July 19, 2023

Recommending Code Reviews Leveraging Code Changes with Structured Information Retrieval

***Abstract*—Review comments are one of the main building blocks of modern code reviews. Manually writing code review comments could be time-consuming and technically challenging. Recently, an information retrieval (IR) based approach has been proposed to automatically recommend relevant code review comments for method-level code changes. However, this technique overlooks the structured items (e.g., class name, library information) from the source code and is applicable only for method-level changes. In this paper, 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% margins in 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, 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.**

***Index Terms*—Software Engineering, Code Reviews, Code Changes, Structured Information Retrieval**

I. Introduction

Code review is one of the popular quality assurance practices in software development and maintenance. It has been widely adopted in open-source and commercial software projects [1, 2]. During code reviews, developers first submit changed code to their peers (a.k.a. reviewers). The reviewers then manually examine the changed code and provide feedback in the form of code review comments. Then the code is revised based on the review comments across multiple iterations and is made ready for integration into the main code base [3]. Besides finding fine-grained defects (e.g., logical errors), code review helps improve the readability [2, 4, 5], maintainability [4], and design quality [6] of the source code.

Modern code review (MCR) has been reported to be beneficial for software maintenance [7, 8]. However, performing the review 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 [9]. Due to the size and complexity of modern software projects, the number of review requests is also high [10, 11]. A code submitter might need to wait for 15 – 64 hours before receiving any code review [12], 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 [13].

Several existing approaches [9, 14–16] recommend or generate code review comments using deep neural (DL) networks. Earlier works [14, 15] use Long Short-Term Memory (LSTM) networks with an attention mechanism [17] to recommend code review comments. Later approaches [9, 16] employ more sophisticated architecture such as Text-To-Text Transfer Transformer (T5) [18] to generate code review comments. However, they require specialized computing resources (e.g., 16 $\times$ 40GB GPU [18]), which could hurt their scalability. They also might require long training time (e.g., 12 days [18]).

Recent studies [19–22] 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. [13] 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 [23] 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 1, 2). According to Li et al. [9], 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. [13] overlooks this structured information and recommends reviews only for the changes in the method.

In this paper, we propose a novel technique, namely – RevCom, that recommends relevant review comments leveraging various code-level changes with structured information retrieval. Our work is inspired by the work of Hong et al. [13] that relies on method-level similarity and the Bag of Words model to recommend code reviews. However, unlike the earlier 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 (four Python + four Java) projects. We use three different metrics – BLEU score [24], perfect prediction, and semantic similarity [25] to evaluate the performance of RevCom. 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 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 [13] and CodeReviewer [9] show that RevCom outperforms them in all three metrics. In particular, our approach is lightweight compared to DL-based techniques and can recommend reviews for both method-level and non-method-level changes where the existing IR-based technique falls short.

We thus make the following contributions in this paper.

- A novel, IR-based approach – RevCom that can recommend relevant code review comments leveraging various structured information from the changed code.
- Extensive experiments demonstrating the effectiveness of RevCom against two state-of-the-art techniques – CommentFinder [13] and CodeReviewer [9]. We also assess the role of different structured information (e.g., *diff*, library information, and file path) in our proposed approach.
- A benchmark dataset containing $\approx$ 56K pairs of $\langle code\ change,\ comment \rangle$ collected from eight popular Java-based and Python-based projects. We also release our dataset[2] and replication package for third-party reuse.

II. Motivating Example

To demonstrate the capability of our approach – RevCom, let us consider the example in Listing 1. The code snippet is taken from *elastic/elasticsearch* Java repository[3]. 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 [13]. Our approach can exploit structured information from those changes and can recommend exact review comments occasionally.

Listing 2 shows another example containing a library-level change. The code snippet is from *ansible/ansible* Python repository[4]. 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  Code Change:
2  @@ -50,7 +52,7 @@
3  public static class Bucket extends
       InternalMultiBucketAggregation.InternalBucket
       implements Histogram.Bucket {
4
5  -        final long key;
6  +        public final long key;
7  ----------------------------------------
8  Ground Truth: "Could you explain why this needs to
       be public now? I think we should try to keep
       this package private if possible".
9
10 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 1. Example of class-level code change

Unfortunately, the state-of-the-art IR-based technique – CommentFinder [13] could be limited for these change scenarios. First, in Listing 1, the change is related to a class-level variable which is declared outside of a method. Second, in Listing 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.

```
1  Code Change:
2  @@ -0,0 +1,125 @@
3  +#!/usr/bin/python
4  +from __future__ import absolute_import, division,
       print_function
5  +from ansible.module_utils.aws.core import
       AnsibleAWSModule
6  + from ansible.module_utils.ec2 import (
       camel_dict_to_snake_dict,
7      ec2_argument_spec)
8  ----------------------------------------------
9  Ground Truth: "These imports aren't needed but you
       will need `camel_dict_to_snake_dict` from `
       ansible.module_utils.ec2`"
10
11 Recommended Comment: "Sorry, use `
       camel_dict_to_snake_dict` from `ansible.
       module_utils.ec2`"
```

Listing 2. Example of library-level code change

III. Background

In this section, we describe the important concepts that will be required to follow the rest of the paper.

A. Modern Code Review

In 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

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

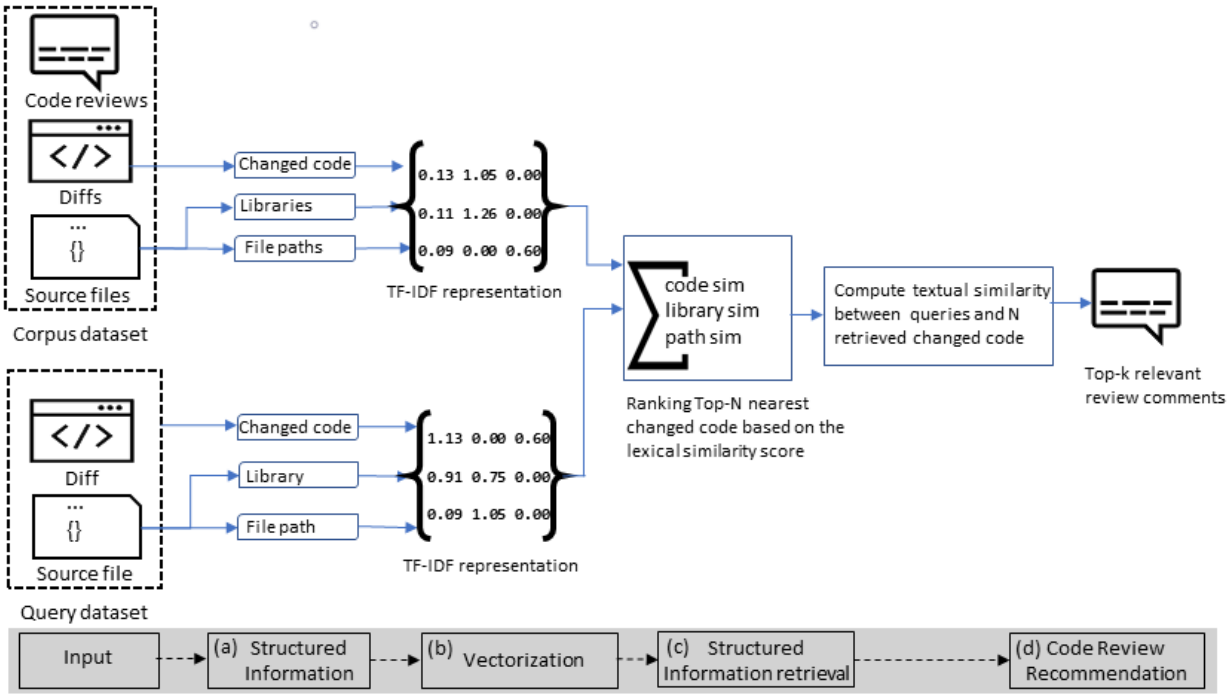

Fig. 1. An overview of our proposed approach– RevCom

finished yet. The reviewers can further provide feedback 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 reviewers' significant time and cognitive effort. Thus, our technique automatically recommends review comments which could reduce the time and effort required for MCR.

B. Word Embedding

Word embedding is a distributed representation of words in a vector space model where semantically similar words appear close to each other [26]. 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 [27]. 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 [28]. Word embedding overcomes many limitations of other vector space models (VSM), 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 [29]. 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 [30] dataset to generate embeddings for our analysis.

IV. Approach

In this section, we present our proposed technique, *RevCom*, that recommends relevant review comments by leveraging structured information from the source code. Figure 1 shows an overview of our technique, which consists of four steps. We describe the details of each step below.

A. Structured Information Extraction

RevCom uses a structured information retrieval-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 [13, 21]. RevCom takes a *diff* and corresponding source code document as input (Step a, Figure 1). A *diff* hunk is a sequence of code that represents the code changes between two versions of the same source file [15]. 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 1, 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. [31] 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. [32] uses file path similarity to determine a reviewer's expertise and then recommend the relevant code reviewers. On the other hand, Hong et al. [13] 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 information to recommend the relevant review comments.

B. Code Token Vectorization

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. [33], 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 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.

C. 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 [34]. It calculates the lexical similarity between two documents (e.g., structured items between two code changes).

We perform a weighted sum of the similarity scores for all three structured items using Equation 1 (Step c, Figure 1). To generate an optimized weight for each structured item, we follow an existing algorithm by Tian et al. [35].

$$LexSim(Q, T) = \sum_{q \epsilon Q} w_q \cdot BM25(q, T_q) \tag{1}$$

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. 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 [13, 15], we use $N = 10$ in our experiment. We thus retrieve the top-10 similar *changed code fragments* from the corpus.

D. 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 IV-C) 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 [13]. This similarity measure calculates the textual similarity between the two documents while preserving the order of the tokens. 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.

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

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 top-N *changed code fragments* 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).

V. 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 [24], perfect prediction, and semantic similarity [25]. We also compare the performance of RevCom with two state-of-the-art baselines [9, 13]. 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?

TABLE I
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**

A. Dataset Construction

To conduct our experiments, we curate a dataset of $\approx$ 56K *diff* and review comment pair from GitHub[5] using its REST API. We first collect the top 20 Java and the top 20 Python repositories based on their star count from GitHub [36, 37]. 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 III-A, 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 [9]. 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 $\approx$ 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 $\approx$ 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 $\approx$ 189K review comments in our dataset. Similar to prior studies [15, 38], we also filter out trivial or short comments (e.g., "nice", "thank you", "LGTM"), which results in $\approx$ 122K review comments. Since RevCom leverages various structured information to recommend relevant review comments, it requires the source code file to extract the file path or library information. We thus eliminate the reviews that are related to documentation or other kinds of source files (e.g., .md or .rst files). Finally, our dataset contains $\approx$ 56K review comments, their associated *diff*, and source files. The summary statistics of our dataset are shown in Table I.

Once we complete the filtration and have a refined dataset, we split it into corpus and query. Similar to earlier work [13], 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.

[5]Accessed: October 12, 2022

B. Embedding Generation

In this work, we adopt Word2Vec [29] – 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 [39]. Therefore, we train a Word2Vec model using GitHub CodeSearchNet [30] 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 [40–43] 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 [44]. This tokenizer leverages Byte-Pair Encoding (BPE) subword tokenization, which splits a word into a sequence of frequently occurring subwords [45]. 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 [13, 46].

C. Evaluation Metrics

To evaluate the performance of our proposed approach, we use three different metrics – BLEU score [24], perfect prediction, and semantic similarity [25]. These evaluation measures were also used by the relevant studies [9, 13, 18, 28, 39, 41], which justify our choice. *We report all metric scores in terms of percentage.* We define these metrics as follows.

1) Bi-Lingual Evaluation of Understanding (BLEU): BLEU score [24] is a widely used textual similarity metric with significant use in the software engineering context [9, 13, 18, 28, 41, 43]. BLEU score calculates the similarity between the recommended reviews and the ground truth reviews in terms of their n-gram precisions as follows.

$$BLEU = BP \cdot exp\left(\sum_{n=1}^{N} w_n \, log(p_n)\right) \quad (3)$$

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 [9, 13, 28], 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.

2) Perfect Prediction (PP): Perfect prediction measures the exact match between recommended review comments and ground truth review comments. Previous studies [13, 16] 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.

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. [25] conduct a human study to identify which metric captures the perception of human raters the best. According to them, Sentence-BERT encoder [47] with cosine similarity has the highest correlation with the human-evaluated similarity. Therefore, we use `stsb-roberta-large`[6] 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.

$$SemSim(G, R) = cos(sbert(G), sbert(R)) \quad (4)$$

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.

D. Baseline for Comparison

We compare the effectiveness of our approach with the state-of-the-art IR-based technique for code review recommendation – CommentFinder [13]. To replicate this technique, we use the replication package provided by the original author [13]. 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 [13], 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 [9]. 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 [9] 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 [9]. The average model training time is one day for the within-project settings and two days for cross-project settings.

VI. Study Result

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

Answering RQ_1 – Performance of RevCom: Table II 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 (k = 1, 3, 5, 10), where k is the number of recommended review comments for a given changed code fragment.

From Table II, 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 high semantic similarity score indicates that recommended review comments from Revcom have a major semantic overlap with the actual review comments from the reviewer. 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

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

TABLE II
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 III
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**

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 III, 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 [28, 48], 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.

Answering RQ_2 – 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 IV, 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 IV, 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

TABLE IV
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**

TABLE V
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**

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 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 IV, 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. [9] 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.

Answering RQ_3 – 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 [29].

From Table V, 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. 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. According to the existing study [15], word embedding can capture the semantics in the changed code effectively. We also use subword tokenization [45] to overcome the OOV issues of word embedding. Thus, the performance improvement of RevCom 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. That is, in terms of cost-benefit analysis, word embedding-based vectorization might not be a feasible 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. Still and all, 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 an infeasible choice for our approach.

TABLE VI
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**

TABLE VII
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**

Answering RQ$_4$ – 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 [13] is the only IR-based technique to recommend review comments, whereas CodeReviewer [9] 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 VI 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 [49], the metrics score obtained from each technique is normally distributed. We thus perform paired Student's t-test [50] to understand the performance difference between any two techniques. According to this test, RevCom performs significantly higher than CommentFinder, i.e., *p-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 V-D). 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 VI, 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-value*$= 0.001 < 0.05$, Cohen's *d* = 0.64 (*medium*) for all three metrics. Thus our findings further confirm the finding of Hong et al. [13] that IR-based techniques perform better than DL-based techniques for code review recommendation.

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 this cross-project setting, we use the instances from three Java-based projects as the corpus and the remaining one for evaluation. To avoid any bias in this project selection, we apply a cross-validation approach and report the average performance for eight different cross-validation results. From Table VII, 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.

VII. 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 [1, 51–53]. Existing studies on automated code review focus on code review generation and code review recommendation.

Review Generation: Several existing approaches [9, 16] use neural machine translation (NMT) to generate code review comments. Tufano et al. [16] pre-trained a Transformer model [54] 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. [9] 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 $\times$ 40GB GPU [18]) and long computing time (e.g., 12 days of training time [18]), 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 VI–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 [14] propose the LSTM-based model DeepMem, which recommends review comments based on existing code reviews and code changes. Siow et al. [15] 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 [40, 41, 43]. Recently, Hong et al. [13] 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 VI, 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.

VIII. Threats To Validity

Threats to *internal validity* relate to experimental errors and biases [35]. Replication of the existing baseline technique could pose a threat. However, we use the replication package provided by the original author of CommentFinder [13] and CodeReviewer [9]. 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 [55] 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 [56]. To mitigate this threat, we evaluate RevCom using the code changes from both Java-based and Python-based projects. As we see from Table II and III, 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.

IX. Conclusion and Future Works

Review comments play a significant role in modern code reviews. Manually writing code review comments requires significant time and effort. A recent study proposes an automated IR-based approach to recommend relevant reviews based on method similarity. However, their approach overlooks the structured information in the code change and recommends the reviews only for the method-level changes. In this paper, 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.

In future, we will investigate how to encode different structured information from source code in a more compact and efficient format. The idea is to keep the approach lightweight while capturing high-quality embeddings representing the semantics of code changes.

References

[1] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th ICSE (ICSE)*. IEEE, 2013, pp. 712–721.

[2] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proceedings of the 11th working conference on MSR*, 2014, pp. 202–211.

[3] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," *IEEE Software*, vol. 35, no. 4, pp. 34–42, 2017.

[4] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th ICSE: Software Engineering in Practice*, 2018, pp. 181–190.

[5] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating code review practices in defective files: An empirical study of the qt system," in *2015 IEEE/ACM 12th Working Conference on MSR*. IEEE, 2015, pp. 168–179.

[6] R. Morales, S. McIntosh, and F. 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)*. IEEE, 2015, pp. 171–180.

[7] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *2013 35th- ICSE*. IEEE, 2013, pp. 931–940.

[8] C. Hannebauer, M. Patalas, S. Stünkel, and V. Gruhn, "Automatically recommending code reviewers based on their expertise: An empirical comparison," in *Proceedings of the 31st IEEE/ACM ASE*, 2016, pp. 99–110.

[9] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM ESEC/FSE*, 2022, pp. 1035–1047.

[10] X. Yang, R. G. Kula, N. Yoshida, and H. Iida, "Mining the modern code review repositories: A dataset of people, process and product," in *Proceedings of the 13th MSR*, 2016, pp. 460–463.

[11] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th ESEC/FSE*, 2013, pp. 202–212.

[12] Y. Hong, C. K. Tantithamthavorn, and P. P. Thongtanunam, "Where should i look at? recommending lines that reviewers should pay attention to," in *2022 IEEE SANER*. IEEE, 2022, pp. 1034–1045.

[13] Y. Hong, C. Tantithamthavorn, P. Thongtanunam, and A. Aleti, "Commentfinder: a simpler, faster, more accurate code review comments recommendation," in *Proceedings of the 30th ACM ESEC/FSE*, 2022, pp. 507–519.

[14] A. Gupta and N. 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.

[15] J. K. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu, "Core: Automating review recommendation for code changes," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 284–295.

[16] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," *arXiv preprint arXiv:2201.06850*, 2022.

[17] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[18] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proc.2021 EMNLP*, 2021, pp. 8696–8708.

[19] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proceedings of the 2017 11th ESEC/FSE*, 2017, pp. 49–60.

[20] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th ESEC/FSE*, 2017, pp. 763–773.

[21] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?" in *Proceedings of the 33rd ACM/IEEE-ASE*, 2018, pp. 373–384.

[22] T. Menzies, S. Majumder, N. Balaji, K. Brey, and W. Fu, "500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow)," in *2018 IEEE/ACM 15th MSR*. IEEE, 2018, pp. 554–563.

[23] W. A. Qader, M. M. Ameen, and B. I. Ahmed, "An overview of bag of words; importance, implementation, applications, and challenges," in *2019 International Engineering Conference (IEC)*. IEEE, 2019, pp. 200–204.

[24] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proc.40th ACL*, 2002, pp. 311–318.

[25] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, "Semantic similarity metrics for evaluating source code summarization," *2022 IEEE/ACM 26th ICPC*, 2022.

[26] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th ICSE*, 2008, pp. 181–190.

[27] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference*, 2019, pp. 964–974.

[28] P. Mahbub, O. Shuvo, and M. M. Rahman, "Explaining

software bugs leveraging code structures in neural machine translation," 2023.

[29] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[30] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[31] M. M. Rahman, C. K. Roy, and R. G. Kula, "Predicting usefulness of code review comments using textual features and developer experience," in *2017 IEEE/ACM 14th MSR*. IEEE, 2017, pp. 215–226.

[32] R. Li, P. Liang, and P. Avgeriou, "Code reviewer recommendation for architecture violations: An exploratory study," *arXiv preprint arXiv:2303.18058*, 2023.

[33] M. Rahman, D. Palani, and P. C. Rigby, "Natural software revisited," in *2019 IEEE/ACM 41st ICSE*. IEEE, 2019, pp. 37–48.

[34] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[35] Y. Tian, D. Lo, and J. 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)*. IEEE, 2014, pp. 44–53.

[36] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th ESEC/FSE*, 2015, pp. 38–49.

[37] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *ICML*. PMLR, 2016, pp. 2091–2100.

[38] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *2021 IEEE/ACM 43rd ICSE*. IEEE, 2021, pp. 163–174.

[39] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *JMLR*, vol. 21, pp. 1–67, 2020.

[40] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," *arXiv preprint arXiv:1711.09573*, 2017.

[41] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *IJCAI*, 2019.

[42] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: Ai-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2727–2735.

[43] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[44] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[45] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[46] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *2020 IEEE/ACM 42nd ICSE*. IEEE, 2020, pp. 1073–1085.

[47] N. Reimers, I. Gurevych, N. Reimers, I. Gurevych, N. Thakur, N. Reimers, J. Daxenberger, I. Gurevych, N. Reimers, I. Gurevych *et al.*, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proc.2019 EMNLP*. ACL, 2019, pp. 671–688.

[48] W. Tao, Y. Wang, E. Shi, L. Du, S. Han, H. Zhang, D. Zhang, and W. Zhang, "On the evaluation of commit message generation models: an experimental study," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 126–136.

[49] S. S. Shapiro, M. B. Wilk, and H. J. Chen, "A comparative study of various tests for normality," *Journal of the American statistical association*, vol. 63, no. 324, pp. 1343–1372, 1968.

[50] R. C. Blair and J. 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*, vol. 97, no. 1, p. 119, 1985.

[51] C. Y. Chong, P. Thongtanunam, and C. 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 ICSE-SEET*. IEEE, 2021, pp. 20–29.

[52] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE TSE*, vol. 35, no. 3, pp. 430–448, 2008.

[53] A. Uchôa, C. Barbosa, D. Coutinho, W. Oizumi, W. K. Assunçao, S. R. Vergilio, J. A. Pereira, A. Oliveira, and A. Garcia, "Predicting design impactful changes in modern code review: A large-scale empirical study," in *2021 IEEE/ACM 18th MSR*. IEEE, 2021, pp. 471–482.

[54] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *NeurIPS*, vol. 30, 2017.

[55] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st ICSE*. IEEE, 2019, pp. 25–36.

[56] M. M. Rahman, C. K. Roy, and D. Lo, "Automatic query reformulation for code search using crowdsourced knowledge," *EMSE*, vol. 24, no. 4, pp. 1869–1924, 2019.