CORE: Automating Review Recommendation for Code Changes

Jing Kai Siow, Cuiyun Gao*, Lingling Fan, Sen Chen, Yang Liu
School of Computer Science and Engineering, Nanyang Technology University, Singapore
jingkai001@e.ntu.edu.sg, {cuiyun.gao,yangliu}@ntu.edu.sg, {ecnujanefan,ecnuchensen}@gmail.com

***Abstract*—Code review is a common process that is used by developers, in which a reviewer provides useful comments or points out defects in the submitted source code changes via pull request. Code review has been widely used for both industry and open-source projects due to its capacity in early defect identification, project maintenance, and code improvement. With rapid updates on project developments, code review becomes a non-trivial and labor-intensive task for reviewers. Thus, an automated code review engine can be beneficial and useful for project development in practice. Although there exist prior studies on automating the code review process by adopting static analysis tools or deep learning techniques, they often require external sources such as partial or full source code for accurate review suggestion. In this paper, we aim at automating the code review process only based on code changes and the corresponding reviews but with better performance.**

The hinge of accurate code review suggestion is to learn good representations for both code changes and reviews. To achieve this with limited source, we design a multi-level embedding (i.e., word embedding and character embedding) approach to represent the semantics provided by code changes and reviews. The embeddings are then well trained through a proposed attentional deep learning model, as a whole named CORE. We evaluate the effectiveness of CORE on code changes and reviews collected from 19 popular Java projects hosted on Github. Experimental results show that our model CORE can achieve significantly better performance than the state-of-the-art model (DeepMem), with an increase of 131.03% in terms of Recall@10 and 150.69% in terms of Mean Reciprocal Rank. Qualitative general word analysis among project developers also demonstrates the performance of CORE in automating code review.

I. Introduction

Code review is a process that involves manual inspection of source code revisions, either by peers or by colleagues, before pushing the revisions to the live system. This process is commonly used by software engineers and open-source project developers to ensure that the new code revisions do not introduce errors and adhere to the guidelines of the projects. A formal and structured framework, commonly known as Fagan Inspection [1] proposed in 1976, to provide developers a way to identify defects in development phase. Code review aids developers in avoiding errors and finding defects during the development or maintenance phase. However, modern code review has evolved from finding bugs to providing maintainability and understanding of source code and their changes [2], [3].

* Cuiyun Gao is the corresponding author.

Although finding defects and errors are still one of the priorities [4], [5], developers from industries value more on motivations, such as transferring of knowledge, understanding of source code and code improvement [2], [6], [7]. Several aspects on code reviews have been explored by software engineering, such as link graph analysis [8], sentiment analysis [9], decision making [10], [11], reviewer recommendation [12]–[14], matching identical code review [15], security bug analysis [16]–[18], and motivations and objectives of code reviews [2], [6].

As the projects become more sophisticated over time, the amount of code change increases daily. Code reviewers often have difficulty in allocating a huge amount of time in performing code review. Furthermore, to conduct code review, reviewers need to understand the purpose and history of the code before they can perform a fair evaluation of the code changes [6]. This results in a huge amount of time needed for the code review process. Many research studies [12], [14], [15] aim to reduce the workload of reviewers such as recommending the best reviewer or even automating the code review generation process. One specific work that is related to our paper, by Gupta and Sundaresan [15], uses basic LSTM models to learn the relation between code changes and reviews.

In practice, developers often use code collaboration tools, such as Gerrit [19] and Review Board [20], to assist them in the process of code review. Their functionalities often include showing changed files, allowing reviewers to reject or accept changes and searching the codebase. Some research work, *e.g.,* ReviewBot [14], incorporates static analysis tools to publish reviews automatically. These static analyzers detect defect code and unconventional naming by the submitter and publish them as part of code review. Static analyzers require a comprehensive set of rules that allow them to detect defective source code. In comparison with human generated reviews, reviews by static analyzers are more rigid and have difficulty in finding errors that are emerged outside of their heuristic rules [21]. Other work, like DeepMem [15], aims to recommend reviews by learning the relevancy between source code and review. However, these methods often require a large amount of additional sources, such as full or partial source code. This additional information might not be available at all times, for instance, submitting pull request or commits.

In this paper, we propose a novel deep learning model for recommending relevant reviews given a code change. We

name the whole COde REview engine as CORE. CORE is built upon only code changes and reviews without external resources. Our motivation is to reduce the workload of developers by providing review recommendation without human intervene. By automating the code review process, developers can correct their code as soon as possible, hence, reducing the time between each revision of code changes. The challenging point of automating code review is to learn good semantic representations for both sources. Although word embeddings [22], [23] have been proven useful in representing the semantics of words, they may fail in capturing enough semantics of code since out-of-vocabulary words are often introduced into the project along development phase [24]. To overcome this challenge, we propose a two-level embedding method which combines both word-level embedding and character-level embedding [25] for representation learning. We then predict the consistency (*i.e.*, relevancy score) between two sources with an attentional neural network, where the attention mechanism [26] learns to focus on the important parts of the two sources during prediction. Experimental analysis based on 19 popular Java projects hosted on GitHub demonstrates the effectiveness of the proposed CORE. CORE can significantly outperform the state-of-the-art model by 131.03% in Recall@10 and 150.69% in Mean Reciprocal Rank.

Qualitative analysis also shows that project developers in the industry are interested in our work and agreed that our work are effective for practical software development.

In summary, our contributions are as follows:

- We propose a novel deep learning model, CORE, for recommending reviews given code changes. CORE combines multi-level embedding, *i.e.,* character-level and word-level embedding, to effectively learn the relevancy between source code and reviews. CORE can well learn the representations of code changes and reviews without external resources.
- Extensive experiments demonstrate the effectiveness of CORE against the state-of-the-art model. We also evaluate the impact of different modules in our model, from which multi-level embedding is proven to be more effective in improving the performance of CORE.
- We build and provide a benchmark dataset containing 57K pairs of <code change, review> collected from 19 popular Java projects hosted on GitHub. We release our dataset on our website[1] for follow-up code review-related tasks.

II. Preliminary

A. Code Review

Traditionally, reviews for source code and code revisions are contributed by humans manually. These reviews are generally in natural languages and short sentences. More often than not, these reviews serve as suggestions or comments for the authors, informing them if there are any errors or concerns with the newly submitted code. There are several tools in the market, such as Gerrit [19], Review Board [20] and Google's Critique [2] that could help the developers in facilitating better reviewing process and providing additional reviewing tools.

Both industry and open-source projects adopted code review process to improve their code quality and reduce the number of errors to be introduced into the codebase. In this paper, we explore the code reviews in open-source projects, for instance, projects that are hosted on GitHub. GitHub allows others to submit their changes to the project through a function known as Pull Request [27]. Meanwhile, the review would be required for these changes to determine whether they should be merged into the main branch of the project. The author of the pull request will request a review for the submitted code changes from the main contributors of the projects. These main contributors are usually experienced developers that involved deeply in the the requested open-source project.

B. Motivating Examples

Listing 1 shows an example of a code review that we aim to match. As we can see, given a code change snippet, the reviewer suggested that instead of using *completions.add(completion)*, the author should use *java.util.Collections.addAll()*. This suggestion is useful in keeping the codebase consistent and could help to reduce unnecessary mistakes. Automating such reviews could allow reviewers to save their precious time and efforts.

Another example in Listing 2 shows that a better naming convention should be used instead of *tcpMd5Sig()*. Such trivial reviews might be very simple but could be very time-consuming for reviewers. Hence, we aim to reduce the workload for reviewers by finding such useful and practical reviews that are commonly used throughout the open-source projects.

```
// Code changes
+      completion.add(completion);
+     }
+   }
+   for (OffsetCommitCompletion completion:
     completions){
-----------------------------------------------
// Review
    "Consider using java.util.Collection.addAll()"
```

Listing 1: Review regarding API replacement

```
// Code changes
+  private volatile Set<InetAddress> tcpMd5Sig =
     Collections.emptySet();
-----------------------------------------------
// Review
    "Could this field and the tcpMd5Sig() method
    have a better name? It does not contain any
    signature but a set of addresses only."
```

Listing 2: Review regarding naming convention

C. Word Embedding

Word embedding are techniques on learning how to represent a single word using vector representation. Each word

[1] https://sites.google.com/view/core2019/

is mapped to a unique numerical vector. For instance, given a word "*simple*", we embed it into a vector of $\boldsymbol{x_1} = [0.123, 0.45, ..., 0.415]$ where $\boldsymbol{x_1}$ is a vector of length n. Commonly, words that have similar meanings tend to have lower distance in the embedded latent space. There are several techniques that employ deep learning and deep neural network to learn richer word representations, such as Word2Vec [22] and ELMo [28]. Word2Vec uses fully connected layer(s) to learn the context around each word and outputs a vector for each word, while ELMo uses Bi-LSTM to learn deeper word representations. Word embedding are often pre-trained to ensure that downstream tasks can be performed as efficiently as possible.

D. Long Short-Term Memory

Long Short-Term Memory (LSTM) is a type of recurrent neural network that is typically used to learn the long term dependencies in sequence data, such as time-series or natural language. It is commonly used in natural language processing as it learns the dependencies in a long sequence of words. Assume that the embedding sequence is in form of $X = [x_1, x_2, ..., x_2]$ where X represents the embedded sequence of tokens, LSTM computes the current output o_t based on its previous state and the memory cell state.

$$o_t = tanh(h_{t-1}, x_t) \tag{1}$$

where h_{t-1} represents the previous state and x_t represents the current input at time-step t. The $\tanh(\cdot)$ is a typical activation function for learning a non-linear adaption of the input. These outputs from LSTMs contain latent representations of the tokens in the sentence.

E. Attention Mechanism

Attention mechanism is proposed by Bahdanau [26] and it is first used in natural language processing, mainly in neural machine translation. Attention mechanism computes a context vector for each sentence. It allows us to have better representation and provides a global context for each sentence that is beneficial to the relevancy learning task in our work. It is computed based on the outputs of LSTMs and learned weights, hence, giving it higher-level representation on the whole sequence. The context vector, shown in Fig. 1, is a weighted sum of the output of LSTMs by using attention weights, α_{ij}.

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})} \tag{2}$$

Equation 2 shows the formula for computing the attention weights, where n is the total number of LSTM outputs. The alignment score, e_{ij}, goes through a softmax function to achieve the attention weights. This allows the mechanism to have a higher weight for variables that contribute more towards the success of the model. e_{ij} can be learned using an activation function and learned weights.

$$e_{ij} = \tanh(W_s S_t + b) \tag{3}$$

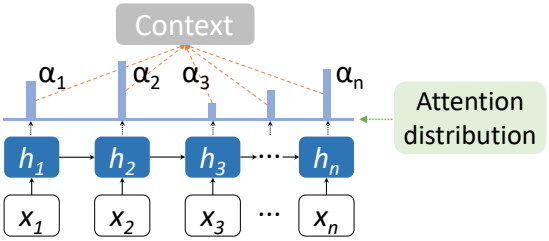


Fig. 1: Attention mechanism

where S_t is the output of LSTM, b and W_s is the bias and weight. Finally, the context vector can be computed by using a sum of the attention weights and the alignment score.

$$\mathbf{a}_t = \sum_{i}^{n} S_t \alpha_{ij} \tag{4}$$

F. Problem Definition

In this work, we define the comments or suggestions by the reviewers as "*reviews*" (denoted by $\mathbf{R}$) and the submitted code as "*code changes*" (denoted by $\mathbf{C}$). Given a snippet of code changes, we aim to find the top-k most relevant reviews in our dataset. We model it as a recommendation/ranking problem such that each code snippet will have a list of reviews that are sorted according to their relevancy score. For each $<c_i, r_i> \in <\mathbf{C}, \mathbf{R}>$, a score (*i.e.*, $Rel(c_i, r_i)$) that indicates the relevancy of the code changes c_i and review r_i will be learned through our proposed deep learning model, which is computed as follows:

$$Rel(c_i, r_i) = F(c_i, r_i, \theta) \tag{5}$$

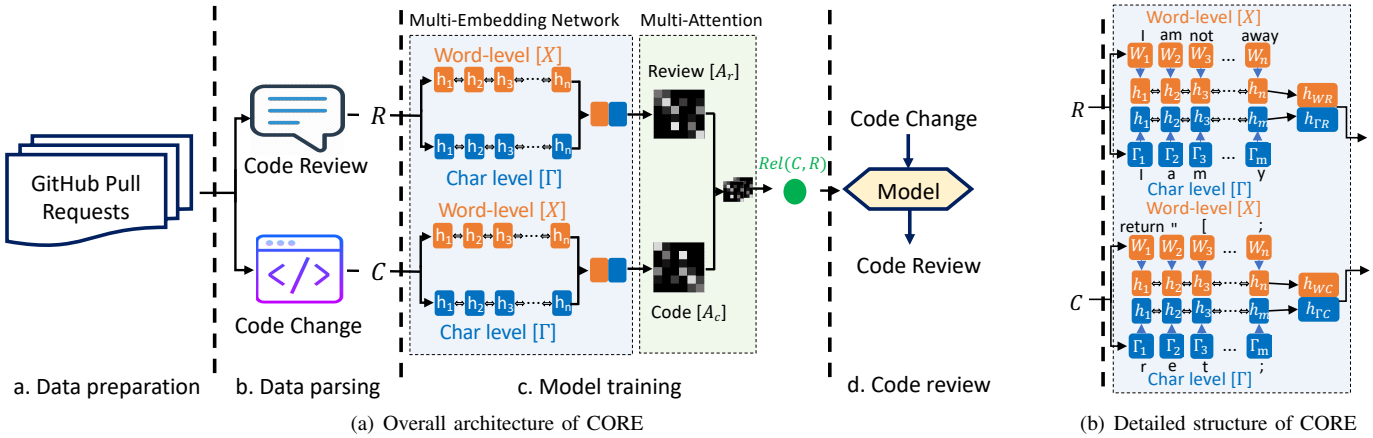
where θ represents the model parameters that will be learned and improved in the training process.

III. Approach

In this section, we present the overview of our proposed model, CORE, and design details that extend the basic attentional Long Short-Term Memory model for code review recommendation. We regard code change and corresponding review text as two source sequences and the relevancy score as the target. Fig. 2 shows the workflow of CORE.

A. Overview

As can be seen in Fig. 2, the workflow of CORE mainly contains four steps, including data preparation, data parsing, model training, and code review. We first collect pull requests from GitHub and conduct preprocessing. The preprocessed data are parsed into a parallel corpus of code changes and their corresponding reviews, *i.e.*, $<\boldsymbol{C}, \boldsymbol{R}>$. Based on the parallel corpus of code changes and reviews, we build and train a neural-based learning-to-rank model. The major challenge during the training process lies in using limited information to well represent the two sources. Finally, we deploy our model for automated code review. In the following, we will introduce the details of the CORE model and the approach we propose to resolve the challenge.



(a) Overall architecture of CORE (b) Detailed structure of CORE

Fig. 2: Structure of the code review model.

B. Multi-Level Embeddings

To better represent code changes and review texts, we propose to combine the two levels of embeddings: **Word-level embedding** and **Character-level embedding**.

1) Word-Level Embedding: Word embeddings are widely adopted in representing the semantics of tokens through training on a large text corpus. In this work, we adopt word2vec [22], a distributed representation of words, as pre-trained embeddings for the words in code changes and reviews, and retrain them in the respective source. The detailed retraining processes are described in the following.

Code review. Code reviews are generally mingled with texts and project-specific tokens. The project-specific tokens usually appear only a few times, including function names, variable names, version numbers of the projects, and hash IDs of commits, so the embeddings of such tokens could not be well learnt and may bring noise into the ultimate review text representation. To alleviate such noise, we first tokenize the reviews and then replace the project-specific terms with placeholders. Specifically, we convert hash IDs with "<HASHID>", numerical digits with "<NUM>", version numbers with "<VERSIONNUM>", and URLs with "<URL>". The preprocessed reviews are employed to retrain review-specific word embeddings.

Code change. Code representation learning is a challenging task in natural language processing field because code usually contains project-specific and rarely-appearing tokens. In spite of their low frequency, they can also help understand the semantics of the code changes. For example, a variable name, "SharedSparkSession", allows us to understand that the variable is closely related to a session and they are shared among multiple users. We adopt a typical parser, *pygments* [29], to parse code changes into tokens. For example, given a source code statement in Java, "*private final int shuffleId;*", we parse them using pygments and append each token to a list, such that we will receive a list of token, "["private", "final", "int", "shuffleId", ";"]". The preprocessed code changes are then fed into the word2vec trainer, from which we can obtain code-specific word embeddings.

2) Character-Level Embedding: The Out-of-Vocabulary (OOV) issue is common in code-related tasks [30]–[33]. This is because code contains not only typical API methods but also randomly-named tokens such as variable names and class names. Although word-level embeddings could represent the semantics of the tokens in source code, the OOV issue still exists since low-frequency words are not included in the pre-trained word2vec model. To alleviate the OOV issue, we propose to combine character-level embeddings. The character-level embeddings are independent of tokens in the collection and represent the semantics of each character. In this work, we embed each character into a one-hot vector. For example, given a review, "please fix this", we separate the sentence into a list of characters, ["p", "l", "e", "s", ..., "t", "h", "i", "s"]. We embed this list of characters using one-hot embedding, such that each character has their unique one-hot representation.

C. Multi-Embedding Network

The multi-embedding network aims at jointly encoding the word-level and character-level representations for both code changes $\mathbf{C}$ and reviews $\mathbf{R}$, with details illustrated in Fig.3(b). We denote the two-level embeddings for both sources, denoted as $\mathbf{W}_C$ and $\mathbf{W}_R$ for the respective word-level embeddings, and $\mathbf{\Gamma}_C$ and $\mathbf{\Gamma}_R$ for the respective character-level embeddings. We adopt Bi-directional LSTMs (Bi-LSTMs) [34] to learn the sequence representations for word-level and character-level code changes, denoted as $\mathbf{h}_{WC}$ and $\mathbf{h}_{\Gamma C}$ respectively. The $\mathbf{h}_{WC}$ and $\mathbf{h}_{\Gamma C}$ are the last hidden states produced by the corresponding Bi-LSTMs. The final embeddings for code changes are defined as:

$$\mathbf{h}_C = \tanh(\mathbf{W}^C[\mathbf{h}_{WC}; \mathbf{h}_{\Gamma C}]), \tag{6}$$

where $[\mathbf{h}_{WC}; \mathbf{h}_{\Gamma C}]$ is the concatenation of the two-level representations, $\mathbf{W}^C$ is the matrix of trainable parameters, C is the number of hidden units, and $\tanh(\cdot)$ is used as the activation function.

Similarly, we can obtain the two-level sequence representations $\mathbf{h}_{WR}$ and $\mathbf{h}_{\Gamma R}$ for code reviews $\mathbf{R}$. The final representations for reviews can be calculated as:

$$\mathbf{h}_R = \tanh(\mathbf{W}^R[\mathbf{h}_{WR}; \mathbf{h}_{\Gamma R}]), \tag{7}$$

where $[\mathbf{h}_{WR}; \mathbf{h}_{\Gamma R}]$ is the direct concatenation of the word-level and character-level embeddings for reviews R, $\mathbf{W}^R$ is the matrix of trainable parameters, and R is the number of hidden units. For simplicity, we assume that the dimensions of the two-level embeddings, *i.e.*, $\mathbf{h}_{WR}, \mathbf{h}_{\Gamma R}, \mathbf{h}_{WC}$, and $\mathbf{h}_{\Gamma C}$, are the same.

D. Multi-Attention Mechanism

To alleviate the influence of noisy input, we employ the attention mechanism [26] on the learned representations of code changes and reviews, *i.e.*, $\mathbf{h}_C$ and $\mathbf{h}_R$. The attention mechanism can make the training process pay attention to the words and characters that are representative of code changes and reviews. As can be seen in Fig. 2, both representations are further enhanced through attention layers:

$$\mathbf{a}_C = \sum_{i}^{n} \mathbf{H}_C \alpha_{ij}^c \tag{8}$$

$$\mathbf{a}_R = \sum_{i}^{n} \mathbf{H}_R \alpha_{ij}^r \tag{9}$$

where outputs, *i.e.*, $\mathbf{a}_C$ and $\mathbf{a}_R$, indicate the learned attended representations of code reviews and reviews respectively. The two attended vectors are then concatenated into a *multi-attention* vector, $\mathbf{a}_{\mathbf{C},\mathbf{R}}$, which is finally trained for predicting relevancy scores $Rel(\mathbf{C}, \mathbf{R})$ between code changes $\mathbf{C}$ and reviews $\mathbf{R}$.

$$\mathbf{a}_{C,R} = [\mathbf{a}_C; \mathbf{a}_R], \tag{10}$$

$$Rel(C, R) = \tanh(\mathbf{w}^\Lambda \mathbf{a}_{C,R}), \tag{11}$$

where $\mathbf{W}^\Lambda$ is the matrix of trainable parameters and $Rel(C, R)$ indicates the predicted relevancy score between one code change and review.

E. Model Training and Testing

1) Training Setting: Since CORE aims at scoring the more related reviews higher given one code change, we determine the training goal as the *Mean Square Error* [35] loss function.

$$Loss = \frac{1}{N} \sum_{i=1}^{N} (Rel(c_i, r_i) - \hat{Rel}(c_i, r_i))^2, \tag{12}$$

where $Rel(c_i, r_i)$ is the true relevancy label, $\hat{Rel}(c_i, r_i)$ is the predicted result of CORE, and N is the total number of code change-review pairs. We use ADAM [36] as our optimizer, with a learning rate of $1e-4$. The number of epochs is set to 50 and we use a dropout rate of 0.2. The word embedding size is set to 300 and the one-hot embedding size for characters is set to 60. The number of hidden states for Bi-LSTMs is set to 400 and the dimension of the attention layer is set to 100. For training the neural networks, we limit the vocabularies of the two sources to the top 50,000 tokens that are most frequently used in code changes and reviews. For implementation, we use PyTorch [37], an open-source deep learning framework, which is widely-used in previous research [38], [39]. We train our model in a server with one Tesla P40 GPU with 12GB memory. The training lasts 35 hours.

2) Testing Setting: We test our model using the GPU as above. Each testing phase took about 10 minutes. Each review is ranked with a random set of 50 reviews which only one of which is the review with the true label.

IV. Experimental Setup

A. Data Preparation

We crawled the experimental datasets from GitHub, where the communities frequently submit pull requests to many open-source projects. We selected the projects for collection based on two criteria: The projects are i) popular JAVA projects on GitHub - to ensure the quality of the pull-request pairs; ii) projects with enough pull-request pairs - which necessitates an automated code review recommendation for code changes. To obtain projects that satisfy the two criteria, we randomly inspect the projects ranked at the top 200 on GitHub in terms of the number of stars, and keep the ones with more than 400 pull requests. We selected 19 projects, including Spark [40], Neo4j [41], and elasticsearch [42]. For each selected project, we created a GitHub API crawler to collect code changes and corresponding reviews. We ran our crawler in July 2019. In total, we crawled 85,423 reviews from the 19 projects.

To further ensure the quality of the experimental datasets, we conducted preprocessing. We first filter out the reviews which are acknowledgement or feedback from the pull request author. This is because their replies are commonly acknowledgement or discussion of any feedback from the reviewer. Then we eliminate the reviews that are not written in English, and convert all the remained reviews into lowercase. We also conduct word lemmatization using NLTK [43], where each word is converted into its base or dictionary form. After removing empty review texts, we finally obtained 57,260 <code change, review> pairs. We randomly split the dataset by $7 : 0.5 : 2.5$, as the training, validation, and test sets, *i.e.*, there are 40,082, 2,863, and 14,315 pairs in the training, validation, and test sets, respectively.

We label the relevancy scores of all the 57,260 change-review pairs as 1, *i.e.*, these pairs are regarded as ground truth or positive samples. To ensure that the model can also learn irrelevant pairs, we generate negative samples. We follow the typical learning-based retrieval strategies [44]. Specifically, we randomly select m reviews corresponding to the other code changes as negative reviews of the current code change, *i.e.*, $<c_i, r_j>$ where $i \neq j$ and the number of r_j equals m. In this work, we experimentally set $m = 5$. Detailed statistics of the experimental datasets are shown in Table I.

B. Evaluation Metrics

We adopt two common metrics for validating the effectiveness of code review recommendation, namely, $Recall@k$ [45], [46] and Mean Rank Reciprocal (MRR) [47], [48], which

TABLE I: Statistics of collected data

	Training Data	Validation Data	Testing Data
#Positive Samples	40,082	2,863	14,315
#Negative Samples	200,410	14,315	71,575
Total	240,492	17,178	85,890

are widely used in information retrieval and the code review generation literature [49]–[51].

$Recall@k$ measures the percentage of code changes for which more than one correct result could exist in the top k ranked results [45], [46] , calculated as follows:

$$Recall@k = \frac{1}{|C|}\sum_{c \in C}\delta(Rank_c \leq k), \tag{13}$$

where C is a set of code changes, $\delta(\cdot)$ is a function which returns 1 if the input is true and 0 otherwise, and $Rank_c$ is the rank of correct results in the retrieved top k results. Following prior studies [15], we evaluate $Recall@k$ when the value of k is 1, 3, 5, and 10. $Recall_k$ is important because a better code review recommendation engine should allow developers to discover the needed review by inspecting fewer returned results. The higher the metric value is, the better the performance of code review recommendation is.

Mean Reciprocal Rank (MRR) is the average of the reciprocal ranks of results for a set of code changes C. The reciprocal rank of a code change is the inverse of the rank of the first hit result [47], [48]. MRR is defined as follows:

$$MRR = \frac{1}{|C|}\sum_{c \in C}\frac{1}{Rank_c}. \tag{14}$$

The higher the metric value is, the better the performance of code review recommendation is.

C. Baselines for Comparison

We compare the effectiveness of our approach with *TF-IDF+LR* (Logistic Regression with Term Frequency–Inverse Document Frequency) [15], [52], [53] and a deep-learning-based approach, *DeepMem* [15].

TF-IDF+LR is a popular, conventional text retrieval engine. It computes the relevancy score between one code change and review text based on their TF-IDF (Term Frequency–Inverse Document Frequency) [54] representations. The training process is implemented by using the logistic regression (LR) method [53]. Specifically, we concatenate the TF-IDF representations of code changes and reviews as the input of the LR method. *DeepMem* [15] is a state-of-the-art code review recommendation engine proposed recently by Microsoft. It recommends code reviews based on existing code reviews and their relevancy with the code changes. To learn the relevancy between the review and code changes, the review, code changes and the context (*i.e.* three statements before and after the code changes) are input into a deep learning network for learning. They employed the use of LSTM for learning the relevancy between code changes and reviews. We use similar settings according to their paper, such as LSTM dimensions and model components. Since the dataset is not publicly released by the authors [15] and our crawled data do not contain context information of code changes, we only take code changes and reviews as the input of DeepMem.

TABLE II: Comparison results with baseline models. $R@K$ indicates the metric $Recall@K$. Statistical significance results are indicated with * for $p - value < 0.01$.

Model	MRR	R@1	R@3	R@5	R@10
TF-IDF+LR	0.089*	0.019*	0.060*	0.100*	0.201*
DeepMem	0.093*	0.021*	0.065*	0.108*	0.208*
CORE	**0.234**	**0.113**	**0.247**	**0.333**	**0.482**

V. Evaluation

According the experimental setup in the above section, we conduct a quantitative analysis to evaluate the effectiveness of CORE in this section. In particular, we aim at answering the following research questions.

- **RQ1**: What is the accuracy of CORE?
- **RQ2**: What is the impact of different modules on the performance of CORE? The modules include word-level embedding, character-level embedding, and the attention mechanism.
- **RQ3**: How accurate is CORE under different parameter settings?

A. RQ1: What is the accuracy of CORE?

The comparison results with the baseline approaches are shown in Table II. We can see that our CORE model outperforms all baselines. Specifically, the result of the non-deep-learning-based TF-IDF+LR model achieves the lowest performance, with 0.019 in Recall@1, 0.201 in Recall@10, and 0.089 in MRR score. The result is consistent with the finding by Gupta and Sundaresan [15]. This indicates that deep-learning-based models tend to better learn the semantic consistency between code changes and reviews.

CORE can increase the performance of DeepMem by 438.1% in Recall@1, 131.0% in Recall@10, and 150.7% in the MRR score. We then use t-test and effect size measures for statistical significance test and Cliff's Delta (or d) to measure the effect size [55]. The significance test result ($p - value < 0.01$) and large effect size ($d > 1$) on all the five metrics of CORE and DeepMem/TF-IDF+LR confirm the superiority of CORE over TF-IDF+LR and DeepMem. This explains that the reviews recommended by CORE are more relevant to the code changes than those from TF-IDF+LR and DeepMem.

B. RQ2: What is the impact of different modules on the performance of CORE?

We study the impact of each of the three modules, including character-level embedding, word-level embedding, and the multi-attention network, on the performance of CORE. We

TABLE III: Comparison results with different module removed. The "CORE-WV", "CORE-CV", and "CORE-ATTEN" indicate the proposed CORE without considering word-level embedding, character-level embedding, the multi-attention network, respectively.

Model	MRR	R@1	R@3	R@5	R@10
CORE-WV	0.091	0.020	0.062	0.102	0.202
CORE-CV	0.103	0.026	0.077	0.123	0.233
CORE-ATTEN	0.233	0.107	0.246	**0.339**	**0.498**
CORE	**0.234**	**0.113**	**0.247**	0.333	0.482

evaluate the model when each of the modules is removed individually, *i.e.*, only one module is removed from CORE in each experiment. The comparison results are listed in Table III.

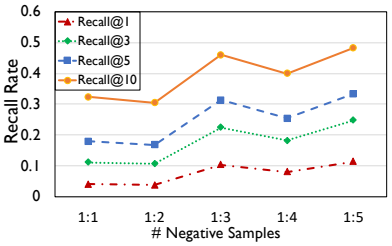
As can be observed in Table III, the combination of all modules achieve the highest improvements in performance. With only word-level embedding or character-level embedding involved, the model can only achieve comparable results as DeepMem. This indicates that the two-level embeddings can compensate for each other for better representing code changes and reviews. By comparing CORE with CORE without the multi-attention network, we can find that the strength brought by the attention mechanism is not that obvious. Without considering attention, the model can achieve slightly higher performance than that with attention in terms of Recall@10 and Recall@5. With attention involved, CORE returns better results for Recall@1. Since developers generally prefer relevant reviews to be ranked higher, our proposed CORE can be regarded as more effective.

C. RQ3: How accurate is CORE under different parameter settings?

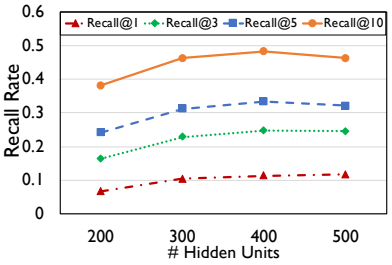
We compare the performance of CORE under different parameter settings. We conduct the parameter analysis for the number of negative samples for each positive code change-review pair, the number of hidden units, and embedding size, respectively. We vary the values of these three parameters and evaluate their impact on the performance.

Fig. 3(a) and Fig. 3(c) show the performance of CORE when different numbers m of negative samples are randomly selected for each collected <code, review> pair. We range m from 1 to 5, denoted as 1:1, 1:2, 1:3, 1:4, and 1:5. As can be seen, the performance of CORE shows a general upward trend in spite that the trend is non-monotonic. Since more negative samples can increase the time cost for model training, we set $m = 5$ for balancing the time cost and performance.

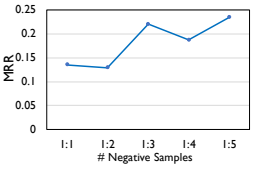
According to Fig. 3(b) and Fig. 3(d), the performance of CORE also varies along with different numbers of hidden units. We can see that more hidden units may not help improve accuracy. CORE generates the best result when we define the number of hidden units as 400. Fig. 3(e) shows the performance of CORE under different dimensions of word-level embeddings. As can be seen, more dimensions can benefit CORE in terms of all metrics. This explains that higher dimensions of word-level embedding can help better learn the representations of code changes and reviews. We can observe that the increase in performance slows down with the embedding dimension growing in 100. We set the dimension of word-level embedding as 300 due to its optimum performance.



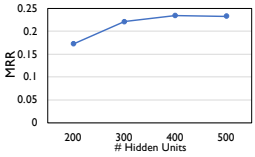
(a) Impact of #negative samples on recall rate



(b) Impact of #hidden units on recall rate



(c) MRR with #negative samples



(d) MRR with #hidden units

Dimension of Word-level Embedding	MRR	Recall@1	Recall@3	Recall@5	Recall@10
100	0.154	0.057	0.143	0.204	0.338
200	0.219	0.105	0.224	0.308	0.452
300	**0.234**	**0.114**	**0.247**	**0.334**	**0.483**

(e) Impact of different dimensions of word-level embedding

Fig. 3: Performance under different parameter settings

VI. User Study

In this section, we conduct a human evaluation to further verify the effectiveness of CORE in automating code review. As the effectiveness evaluation may be subjective, we consulted industrial developers to assess the effectiveness of CORE. We collaborated with Alibaba, which is one of the largest e-commerce companies worldwide and the developers are proficient in software development. We set up several online meetings with them and conduct our human evaluation in the company with their help. Our human evaluation involves 12 front-line Java developers and code reviewers that have at least 3 years of coding and code reviewing experience. Specifically, we applied CORE to the 19 popular Java projects

TABLE IV: Questions in the developer survey

Questions	#Participants
Q1. Do you think the recommended reviews are effective for practical development?	12
Q2. If you think the automated code review tool is useful, which parts make you think so? If not useful, which parts?	12

hosted on GitHub, and randomly selected 10 recommended reviews for code changes from each project. We asked the two questions shown in Table IV to developers.

Developers showed great interest in CORE, and all the developers agreed that our tool can help them in practical development. For the answers to the second question, the developers point out the useful aspects and the aspects that need further improvement. Specifically, the useful aspects given by the developers include relevant reviews that CORE recommended and suggested actions in the recommended review. The aspects for enhancement are mainly about the recommendation of the code changes for review (*e.g.*, pointing to the second line and giving review "*There exists a potential NullPointerException issue*".) and highlighting the review keywords. In fact, for the industrial companies, it is a great demand for project development and maintenance to automatically generate code review. The code review process is a critical phase to ensure the code quality of the project, however, it costs substantial human effort. The demand for automated code review generation will be more urgent, especially for companies like Alibaba which provide services to a large number of users.

VII. Discussion

In this section, we discuss the advantages and limitations of CORE. We also illustrate the threats of validity of our work regarding the dataset and evaluation.

A. Why does CORE work?

We have identified two advantages of CORE that may explain its effectiveness in code review recommendation.

Advantage 1: CORE can well learn representations of source code and reviews. CORE represents the code changes and reviews based on the two-level (including character-level and word-level) embeddings and also multi-attention network. The two-level embeddings can well capture the semantics of both code changes and reviews, and the multi-attention network allows CORE to focus on the words and characters that are representative of code changes and reviews. In this way, CORE can well learn the representations of both sources. We highlight two real cases to demonstrate the advantages.

```
// Code changes
-   return "[" + nodeId + "][" + taskId + "] failed
     , reason [" + getReason() + "]";
-----------------------------------------------
// Review by CORE
    "please add a message otherwise this just
    throws a nullpointerexception with no helpful
    message about what was null"
// Review by DeepMem
    "do we need to fully qualify this io netty
    handler codec http2 http2stream state
    http2stream state"
// Review by TF-IDF+LR
    "do we need to fully qualify this io netty
    handler codec http2 http2stream state
    http2stream state"
```

Listing 3: Case study on retrieving top review (return message)

Listing 3 shows the code changes and the top reviews recommended by CORE and baseline models. As observed in Listing 3, CORE recommends the most relevant review for the code change, providing actions such as adding a message or warning about an exception. CORE can learn the relevancy between semantically-related tokens, such as "*failed*" in the code and "*NullPointerException*" in the review. The two tokens are generally used to express an exception or error that occurs unexpectedly. But both reviews recommended by the baselines do not capture the failure-related token in the code changes and produce wrong results. We also discover that DeepMem and TF-IDF+LR prioritize the same review, which may be due to both models focus on general words, such as "*return*" and "*nodeID*"

```
// Code changes
-package org.elasticsearch.node;
-import org.elasticsearch.common.collect.
    ImmutableOpenMap;
-import org.elasticsearch.common.settings.Settings
    ;
-import org.elasticsearch.test.ESTestCase;
-import static org.hamcrest.Matchers.equalTo;
-public class NodeModuleTest extends ESTestCase{
- public void testIsNodeIngestEnabledSettings(){
-----------------------------------------------
// Review by CORE
    "can you split this into two different tests
    rather than using the randomness here i dont
    think it buys us anything"
// Review by DeepMem
    "i'd make the string here junk or not
    interpreted or something if you dont read the
    file carefully it looks like the script is run
     because that is a valid looking script"
// Review by TF-IDF+LR
    "we need the version check here as well for bw
     comp"
```

Listing 4: Case study on retrieving top review (test cases)

A similar case can be found in Listing 4. The code change is mainly about adding a new test case which can be well captured by CORE (*e.g.*, the token "*tests*" in the review can be well-matched with "*test*" in the code). Such a semantic match cannot be observed in the recommended reviews of baselines. For example, the prioritized review of DeepMem discusses string manipulation and the top review of TF-IDF+LR talks about versioning.

Advantage 2: CORE can better solve Out-of-Vocabulary (OOV) issues. To solve the OOV issue in code changes,

CORE builds upon character-level embedding. We present two real cases for illustrating the effectiveness of character-level embedding on the OOV issue.

```
// Code changes
- deleteSnapshot(snapshot.snapshotId(), new
    DeleteSnapshotListener(){
+ deleteSnapshot(snapshot.snapshotId().getSnapshot
    (), new DeleteSnapshotListener() {
-----------------------------------------------
// Review by CORE
    "can you add a comment here as to why we have
    a special handling for external versions and
    deletes here"
// Review by CORE-WV
    "should values lower than versionnum be
    allowed here"
```

Listing 5: Case study on comparing CORE-WV & CORE

```
// Code changes
-  inSyncAllocationIds.contains(shardRouting.
    allocationId().getId()) == false)
+  inSyncAllocationIds.contains(shardRouting.
    allocationId().getId()) == false &&
+  (inSyncAllocationIds.contains(RecoverySource.
    ExistingStoreRecoverySource.
    FORCED_ALLOCATION_ID) == false
-----------------------------------------------
// Review by CORE
    "can we remove this allocation"
// Review by CORE-WV
    "that s just a minor thing but i think the
    recommended order in the java styleguide is
    static final"
```

Listing 6: Case study on comparing CORE-WV & CORE 2

As observed in Listing 5, CORE recommends a review suggesting to add comments for the handling and the deletes. The review is strongly relevant to the code change which revolves around deletion. Without the character-level embedding considered, CORE-WV ranks a different and unrelated review at the top. One possible reason is that CORE-WV could not well learn the semantic representation of "*deleteSnapshotListener*" and "*deleteSnapshot*" due to the low co-occurrence frequency[2] of the subwords (such as "*delete*" and "*snapshot*"). A similar case can be found in Listing 6, CORE-WV cannot well capture the semantics of the variable "*inSyncAllocationIds*" while CORE can learn that the token is semantically related to "*allocation*". Thus, with character-level embeddings, CORE can better focus on the important characters in the variable and function name.

B. Limitations of CORE

We show the limitations of CORE by using Listing 7. CORE ranks this example at position 30. One possible reason for the low relevancy score could be the lack of related keywords between the code change and the review. Without relevant keywords, (*e.g.*, shown in Section VII-A), CORE fails to capture the relevancy between the two of them. Furthermore, the review contains only generic keywords (*e.g.*, "*supposed*", "*were*") and does not have any keywords relevant to "*asset*" or "*HashSet*". Despite that CORE does not retrieve the ground truth, its recommended review shows that CORE captures the main semantics of the code change. As can be seen in the code change, the function "*assetEquals*" is invoked to evaluate a condition. CORE understands that the code change is related to checks and assertions; hence, the retrieved review is relevant to an assertion. This further shows that CORE can focus on the key information in the code changes. Moreover, CORE is flexible and extensible to integrate external information, and can involve source code (such as code structure) and code comments to better learn the semantic relevancy between code changes and reviews in future.

[2]The token "*deleteSnapshotListener*" only appears 349 times among the whole GitHub repository [56].

TABLE V: Performance of CORE regarding different code token length

Code Length (l)	MRR	R@1	R@3	R@5	R@10
l <**25**	0.1954	0.0789	0.2171	0.2828	0.4342
25$\leq l <$**50**	0.2186	0.1029	0.2242	0.3010	0.4685
50$\leq l <$**75**	**0.2423**	**0.1246**	**0.2468**	**0.3441**	**0.5000**
$l >$**75**	0.2352	0.1138	0.2490	0.3353	0.4829

```
// Code changes
-  assertEquals(3, exec("def x = new HashSet(); x.
    add(2); x.add(3); x.add(-2); def y = x.
    iterator(); " +
-  "def total = 0; while (y.hasNext()) total += (
    int)y.next(); return total;"));
-----------------------------------------------
// Ground Truth
    "were these supposed to be removed"
// Review by CORE
    "that check should be reversed we assert that
    it s not null the else part dealing with the
    case when it is"
```

Listing 7: Case study on limitation of CORE

We further investigate the relation between the length of code tokens and the performance of CORE. Table V shows the MRR and Recall@K for different code lengths. Our experiments show that CORE performs better for the code changes with lengths ranging from 50 to 75. One possible reason might be that longer code sequences are trimmed to a fixed length, and only the beginning part of the sequence may not fully represent the semantics of the code changes.

C. Threats to Validity

1) Subject Dataset: One of the threats is the quality of code changes and reviews in the dataset. Overlap of data in the training and testing set has been a great issue among deep learning. One of the biggest concerns for our work is that for any pair of <code changes, review>, the negative data might exist in training set while one true positive data exists in the testing set, vice versa. This results in the model learning some parts of the testing set during the training phase. In our

work, we ensure that our training and testing set do not have any overlapping pairs of <code changes, review> by splitting the data into two sets before we negatively sample them. Therefore, the training set will have its own set of positive and negative pairs while the testing set has its own set of positive and negative pairs.

2) Comparison with DeepMem: Another threat is that the results of the DeepMem in our implementation could be lower than the original model. The result on DeepMem in the original paper [15] shows a Recall@10 of 0.227 and 0.2 MRR score. However, in our implementation of DeepMem, it only achieved 0.208 in Recall@10 and 0.1 in MRR score. This may be due to the reason that their method uses additional code contexts, *i.e.*, three statements of code between and after the code changes, for additional context. We do not consider the context as their data are not publicly available and our crawled data do not contain such information. To combat such a threat, we carefully review the technical part of DeepMem in the published paper and confirm the implementation with the other three co-authors. Furthermore, we evaluate both CORE and the implemented DeepMem with similar settings and on the same dataset for a fair comparison.

3) User Study: The quality of user study might be a threat to the validity of this paper. Our user study involves some perspectives from the developers and the feedback might varies from each developers. We mitigate the threat by seeking developers with at least 3-5 years of coding experiences. Furthermore, we ensure that they have at least have prior experiences to code reviewing in the industry.

VIII. Related Work

A. Automating Code Review

The techniques for automating code review can be generally divided into static analysis and deep learning. Static Analysis tools can provide a fast and efficient preliminary analysis of the source code. A code collaboration tool, ReviewBot [14], uses the results of existing static analysis tools and generates reviews and reviewer recommendations. A machine learning model, by Michal et al. [11], uses metrics such as information about the author, files attributes and source code metrics, to classify the code changes. Going beyond the machine learning and static analysis, Pavol et al. [57] uses the results of static analysis tools and synthesis algorithms to learn edge cases that common static analysis tools could not find. This approach solved the overfitting problem that is commonly found in such methods. There are several works that employ the use of deep learning and code review [15]. One of the most relevant work, a model by Anshul Gupta et al. [15], uses information retrieval techniques, such as LSTM and fully connected layer to learn the relationship between source code and the reviews. A recent work, by Toufique et al. [9], uses Sentiment Analysis on code review to determine if the reviews are positive, neutral or negative comments. Shi et al. [10] presents a deep learning-based model that uses source codes that are before modification and after modification. The model could determine if the submitted code changes are likely to be approved or rejected by the project administrators. Chen et al. [58] proposed to extract components like UI pages from Android apps to help the review process. Unfortunately, some of these works do not focus on review generation or retrieval. In those works that concern with code reviews retrieval or generation, they often require much more code contexts than just the code changes.

B. Code to Embedding

Due to the nature of source code, several works focus on source code embedding. The most commonly used method in embedding source is by using word2vec [22]. Word2Vec models the distributions of the word in a large corpus and embed words into a common latent dimension. It is one of the most commonly used embeddings in both natural languages and source code. A work by Gu et al. [59] uses several different features of the function to embed the source code. It uses method name, API sequences and tokens in the embedding function to obtain the word/token vector. Graph-based embedding, such as Code2Seq [60], uses Abstract Syntax Tree (AST) to embed code sequences to represent their underlying structure. These embedding techniques allow us to understand the inner representation of code structures, enhancing any code-related deep learning task. for instance, our work.

C. Code Summarization

Summarization of source code has been a research problem for a long time. Despite the granularity of code summarization is much bigger, some similarities exist between code reviews and summarization. Several automated tools, such as Javadocs [61] and Doxygen [62], can be used to provide comments for source code. Gu et al. [59] uses multiple features to embed source codes and uses them to search for similar comments. Iyer et al. [63] uses LSTM with attention to perform code summarization. Hu et al. [33] uses Seq2Seq and AST embedding to provide a more sophisticated deep learning approach for the same problem. Much earlier works, such as Haiduc [64], implemented summarization by text mining and text retrieval methods. Some papers [65], [66] also researched on heuristic-based and natural language techniques to provide comments for source code of small functions/methods.

IX. Conclusion

We proposed a novel multi-level embedding attentional neural network, CORE, for learning the relevancy between code changes and reviews. Our model is based on word-level and character-level that aims to capture both the semantic information in both source code and reviews. In the future, the generation of reviews could be improved by using neural translation and better embedding for code changes.

Acknowledgement

We appreciate the constructive comments from reviewers. This work is supported by Alibaba Cloud Singapore Grant AN-GC-2018-019. The authors would like to thank Nvidia for their GPU support.

References

[1] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[2] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," in *International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP)*, 2018.

[3] 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 Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 202–211. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597082

[4] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 408–419.

[5] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, "Efficiently manifesting asynchronous programming errors in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 486–497.

[6] C. Bird and A. Bacchelli, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the International Conference on Software Engineering*. IEEE, May 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/expectations-outcomes-and-challenges-of-modern-code-review/

[7] M. Greiler, C. Bird, M.-A. Storey, L. MacLeod, and J. Czerwonka, "Code reviewing in the trenches: Understanding challenges, best practices and tool needs," Tech. Rep. MSR-TR-2016-27, May 2016.

[8] T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto, "The review linkage graph for code review analytics: A recovery approach and empirical study," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 578–589. [Online]. Available: http://doi.acm.org/10.1145/3338906.3338949

[9] T. Ahmed, A. Bosu, A. Iqbal, and S. Rahimi, "Senticr: A customized sentiment analysis tool for code review interactions," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 106–111.

[10] S.-T. Shi, M. Li, D. Lo, F. Thung, and X. Huo, "Automatic code review by learning the revision of source code," in *AAAI*, 2019.

[11] M. Madera and R. Tomoń, "A case study on machine learning model for code review expert system in software engineering," in *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Sep. 2017, pp. 1357–1363.

[12] 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 International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970306

[13] S. Asthana, R. Kumar, R. Bhagwan, C. Bird, C. Bansal, C. Maddila, S. Mehta, and B. Ashok, "Whodo: Automating reviewer suggestions at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 937–945. [Online]. Available: http://doi.acm.org/10.1145/3338906.3340449

[14] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 931–940.

[15] A. Gupta, "Intelligent code reviews using deep learning," 2018.

[16] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu, and L. Xu, "An empirical assessment of security risks of global android banking apps," in *Proceedings of the 42st International Conference on Software Engineering*. IEEE Press, 2020, pp. 596–607.

[17] S. Chen, T. Su, L. Fan, G. Meng, M. Xue, Y. Liu, and L. Xu, "Are mobile banking apps secure? what can be improved?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 797–802.

[18] S. Chen, G. Meng, T. Su, L. Fan, M. Xue, Y. Xue, Y. Liu, and L. Xu, "Ausera: Large-scale automated security risk assessment of global mobile banking apps," *arXiv preprint arXiv:1805.05236*, 2018.

[19] "Gerrit open-source code review tool," https://www.gerritcodereview.com/, accessed: 2019-10-05.

[20] "Review Board code review tool review bot," https://www.reviewboard.org/, accessed: 2019-10-05.

[21] P. Anderson, "The use and limitations of static-analysis tools to improve software quality," *CrossTalk-Journal of Defense Software Engineering*, vol. 21, 06 2008.

[22] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. USA: Curran Associates Inc., 2013, pp. 3111–3119. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999792.2999959

[23] T. Mikolov, G. Corrado, K. Chen, and J. Dean, "Efficient estimation of word representations in vector space," 01 2013, pp. 1–12.

[24] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," *CoRR*, vol. abs/1602.03001, 2016. [Online]. Available: http://arxiv.org/abs/1602.03001

[25] X. Zhang and Y. LeCun, "Text understanding from scratch," *CoRR*, vol. abs/1502.01710, 2015. [Online]. Available: http://arxiv.org/abs/1502.01710

[26] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.

[27] "Pull Request github pull request," https://help.github.com/en/articles/about-pull-requests, accessed: 2019-10-05.

[28] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *CoRR*, vol. abs/1802.05365, 2018. [Online]. Available: http://arxiv.org/abs/1802.05365

[29] "Pygments pygments parser," http://pygments.org/, accessed: 2019-10-15.

[30] J. Li, Y. Wang, I. King, and M. R. Lyu, "Code completion with neural attention and pointer networks," *CoRR*, vol. abs/1711.09573, 2017. [Online]. Available: http://arxiv.org/abs/1711.09573

[31] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 3975–3981. [Online]. Available: https://doi.org/10.24963/ijcai.2019/552

[32] 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*, ser. KDD '19. New York, NY, USA: ACM, 2019, pp. 2727–2735. [Online]. Available: http://doi.acm.org/10.1145/3292500.3330699

[33] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 200–210. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196334

[34] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, Nov 1997.

[35] (2019) Mean Square Error. [Online]. Available: https://en.wikipedia.org/wiki/Mean_squared_error

[36] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.

[37] "Pytorch pytorch - from research to production," https://pytorch.org/, accessed: 2019-10-20.

[38] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," *arXiv preprint arXiv:1909.06727*, 2019.

[39] R. Feng, S. Chen, X. Xie, L. Ma, G. Meng, Y. Liu, and S.-W. Lin, "Mobidroid: A performance-sensitive malware detection system on mobile platform," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2019, pp. 61–70.

[40] "Spark apache spark," https://github.com/apache/spark, accessed: 2019-10-20.

[41] "NEO4J neo4j," https://github.com/neo4j/neo4j, accessed: 2019-10-20.

[42] "elasticsearch elasticsearch," https://github.com/elastic/elasticsearch, accessed: 2019-10-20.

[43] "nltk nltk," https://www.nltk.org, accessed: 2019-10-20.

[44] B. Hu, Z. Lu, H. Li, and Q. Chen, "Convolutional neural network architectures for matching natural language sentences," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2042–2050. [Online]. Available: http://papers.nips.cc/paper/5550-convolutional-neural-network-architectures-for-matching-natural\-language-sentences.pdf

[45] J. Weston, C. Wang, R. J. Weiss, and A. Berenzweig, "Latent collaborative retrieval," *CoRR*, vol. abs/1206.4603, 2012. [Online]. Available: http://arxiv.org/abs/1206.4603

[46] D. Sharma and D. Garg, "Information retrieval on the web and its evaluation," *CoRR*, vol. abs/1209.6492, 2012. [Online]. Available: http://arxiv.org/abs/1209.6492

[47] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2015, pp. 260–270.

[48] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," 11 2014.

[49] D. R. Radev, H. Qi, H. Wu, and W. Fan, "Evaluating web-based question answering systems," in *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC'02)*. Las Palmas, Canary Islands - Spain: European Language Resources Association (ELRA), May 2002. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2002/pdf/301.pdf

[50] E. Voorhees and D. Tice, "The trec-8 question answering track evaluation," *Proceedings of the 8th Text Retrieval Conference*, 11 2000.

[51] A. Severyn and A. Moschitti, "Learning to rank short text pairs with convolutional deep neural networks," in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '15. New York, NY, USA: ACM, 2015, pp. 373–382. [Online]. Available: http://doi.acm.org/10.1145/2766462.2767738

[52] J. Ramos, "Using tf-idf to determine word relevance in document queries," 01 2003.

[53] W. P. Ramadhan, S. T. M. T. A. Novianty, and S. T. M. T. C. Setianingsih, "Sentiment analysis using multinomial logistic regression," in *2017 International Conference on Control, Electronics, Renewable Energy and Communications (ICCREC)*, Sep. 2017, pp. 46–49.

[54] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing & Management*, vol. 24, no. 5, pp. 513 – 523, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0306457388900210

[55] S. E. Ahmed, "Effect sizes for research: A broad application approach," *Technometrics*, vol. 48, no. 4, p. 573, 2006.

[56] "Frequency of token deletesnapshot," https://github.com/search?q=DeleteSnapshotListener&type=Code.

[57] P. Bielik, V. Raychev, and M. T. Vechev, "Learning a static analyzer from data," *CoRR*, vol. abs/1611.01752, 2016. [Online]. Available: http://arxiv.org/abs/1611.01752

[58] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: Automated generation of storyboard for android apps," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 596–607.

[59] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, pp. 933–944.

[60] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *CoRR*, vol. abs/1808.01400, 2018. [Online]. Available: http://arxiv.org/abs/1808.01400

[61] D. Kramer, "Api documentation from source code comments: a case study of javadoc," in *SIGDOC*, 1999.

[62] "Doxygen generate documentation from source code," http://www.doxygen.nl/, accessed: 2019-10-01.

[63] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, vol. 1. Association for Computational Linguistics, 8 2016, pp. 2073–2083.

[64] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, May 2010, pp. 223–226.

[65] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 43–52. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859006

[66] P. Rodeghero, C. Liu, P. W. McBurney, and C. McMillan, "An eye-tracking study of java programmers and application to source code summarization," *IEEE Transactions on Software Engineering*, vol. 41, pp. 1038–1054, 2015.