COME: COmmit Message Generation with Modification Embedding

Anonymous Author(s)
Submission Id: 218*

ABSTRACT

Commit messages concisely describe code changes in natural language and are important for program comprehension and software maintenance. However, they are often overlooked by developers, so it is necessary to automatically generate high-quality commit messages from code changes. Previous studies benefit from representing code changes with abstract syntax trees, using contextualized code representation, and guiding translation with retrieval results. But the effect is still limited due to complex model structures, lack of contextual semantic information of code changes, and improper combination of translation-based and retrieval-based approaches.

This paper proposes a framework named COME, in which we use modification embedding to represent code changes in a fine-grained way, design a self-supervised generative task to learn contextualized code change representation, and combine retrieval-based and translation-based methods through a decision algorithm. To validate the performance of COME, extensive experiments are conducted on two widely used benchmark datasets with four automatic metrics (BLEU, METEOR, ROUGE-L and CIDEr). We also verify the quality of generated commit messages with human evaluation. All the results show that COME outperforms state-of-the-art approaches. In addition, we analyse the effectiveness of three main components of COME, including modification embedding, contextualized code change representation and decision module.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Automatic commit message generation, Contextualized code change representation learning, Self-supervised Learning

1 INTRODUCTION

As version control systems are widely used in software development, commit messages, which use natural language to describe code changes required by version control systems, become more and more important[8, 36]. As an example of code change and its commit message shown in Figure 1, a function name is changed to better describe its intent. The line marked with "-" in red background indicates that the code is deleted, while the line marked with "+" in green background is the newly added code, and the corresponding commit message is shown at the bottom. Commit messages record the purpose of code changes, such as modifying a function or fixing a bug, and help programmers understand the reason behind program evolution, greatly reducing code reading time[10, 24].

However, because writing high-quality commit messages manually is time-consuming, laborious, and not mandatory, programmers always overlook it[33]. According to the research of Dyer et al.[12], in more than 23K open source Java SourceForge projects, about 14% of the commit messages are completely empty.

code change:

```
1  1   public class PlaybackOverlayFragment extends DetailsFragment {
2      -    public final InputEventHandler getInputEventListener() {
       2    +    public final InputEventHandler getInputEventHandler() {
3  3            return mInputEventHandler;
4  4        }
```

commit message:

Rename getInputEventListener to getInputEventHandler

Figure 1: An example of code change and its commit message.

Therefore, it is necessary to automatically generate high-quality commit messages from code changes. Over the years, many approaches have been proposed to address this need. Retrieval-based approaches[17, 30] use similarity comparison algorithms to find the most similar code changes in the training set and reuse its commit message. Translation-based approaches[11, 22, 28, 32, 40, 62] treat commit message generation as a translation task from code changes to commit messages and use labeled data to train the model. Hybrid approaches[29, 47, 55] combine the above two approaches to obtain final commit messages.

Although these approaches have achieved comparative success, they still have some critical limitations. First, most of them treat code changes directly as text sequences, which contain a lot of duplicate information and do not highlight the changed part[22, 28, 32, 40, 62]. Other approaches represent code changes with abstract syntax trees and highlight editing operations through special edges[11, 29], but the difficulty of data processing and complexity of the model structure increase significantly, and the improvement of effect is still limited. Second, a large number of previous studies use models whose parameters do not contain contextual semantic information of code changes, which reduces the quality of the generated commit message. Third, retrieval-based and translation-based approaches have their own advantages and disadvantages, but few works properly combine the two approaches.

To solve these problems, we design a novel framework named **COME**, which consists of four modules. First, the embedding module represents the code changes in a fine grained way and obtains the modification embedding according to the edit distance algorithm. Then, a self-supervised generative task is used to train an encoder-decoder neural network to obtain the contextualized code change representation and the model is further fine-tuned with commit message generation data. The translation module utilizes the well-trained model to generate a translation result, and the retrieval module uses the encoder to retrieve a result at the same time. Finally, the decision module selects one of the above two results as the final commit message.

We evaluate COME on two common and widely used datasets[51, 62] and the results show that COME outperforms all state-of-the-art

approaches on four automatic evaluation metrics (BLEU, METEOR, ROUGE-L and CIDEr) and human evaluation. We also analyse the effectiveness of modification embedding, contextualized code change representation and decision module.

Our contributions are mainly as follows:

(1) We propose a **modification embedding** based on edit distance to represent code changes in a fine grained way.
(2) We obtain the **contextualized code change representation** which include the contextual semantic information of code changes using a self-supervised generative task.
(3) We design COME, a novel framework that combines retrieval-based and translation-based approaches properly with a **decision algorithm**.
(4) We perform extensive experiments on two datasets. The results demonstrate that COME outperforms state-of-the-art approaches.

2 BACKGROUND

Despite the impressive results of existing commit message generation approaches, all of them still face some limitations. First, they do not use code changes appropriately. Second, most of them do not contain contextual semantic information of code changes. Third, they failed to properly combine translation-based and retrieval-based approaches. In this section, we will further explain these limitations and introduce our solutions in each module of COME.

2.1 Usage of Code Change

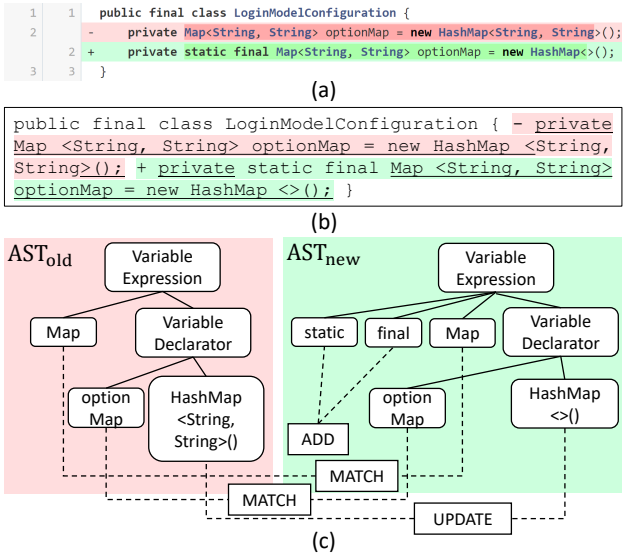

Figure 2: Usage of code changes in existing approaches. (a) git diff file; (b) coarse grained approaches represent code changes in their original sequential order; (c) fine grained approaches use abstract syntax trees to represent code changes.

The distributed version control system represents code changes by marking a whole line of code as deleted (start with "-"), added (start with "+"), or unchanged in git diff files, as shown in Figure 2(a). Coarse grained commit message generation approaches directly tokenize the words in the git diff file in their original sequential order[22, 28, 32, 40, 55], as shown in Figure 2(b), or add some special token to separate the added and deleted code[47, 62]. These methods do not highlight the specific difference between old and new code, and have a lot of duplicate information (underlined in Figure 2(b)), especially when code changes contain few edit operations with the majority of tokens unchanged. As a result, it is difficult for models to accurately capture subtle edit operations, and can lead to imprecise commit message generation. Fine grained approaches use abstract syntax trees to represent code changes, highlight edit operations with several special predefined edges[11], as shown in Figure 2(c), and use complex model structures such as graph neural networks to extract features. Although the effect has improved slightly, the difficulty of data processing and the complexity of the model structure have also increased significantly. In addition, fine grained approaches will not work when code fragments in git diff files are incomplete and cannot be resolved into a valid abstract syntax tree.

To solve the above problems, we want to design a more concise, efficient, and easy to implement method for representing code changes. An intuitive idea is to use edit distance information between added and deleted code. We propose a novel modification embedding that represents code changes in a fine grained way in the embedding module. It can highlight the specific edit operations of the code change, while maintaining the input data as a sequence instead of other complex structures.

2.2 Contextualized Code Change Representation

Most existing translation-based approaches use commit message generation data to train the model directly after randomly initializing parameters. However, using these parameters that do not contain contextual semantic information of code changes leads to poor results.

Shi et al.[47] and Jung[23] use code-related pre-trained models CodeT5[57] and CodeBert[13] to obtain contextualized code representation, and fine-tune them with commit message generation datasets. But code changes in the commit message generation task focus more on contextual semantic information of code changes than the code itself, so they may be very different from the training data of the pre-trained model.

Nie et al.[40] propose two tasks to learn the representation of code. For explicit code changes, the deleted code is used to predict the added code. For implicit binary file changes, code blocks are masked and the model needs to predict those masked tokens. However, downstream tasks do not explicitly distinguish between explicit and implicit code changes, and this method can only predict added code by deleted code in one direction.

To learn the contextualized code change representation easily and effectively, we build a self-supervised generative task based on modification embedding of code changes. The encoder-decoder neural network used in the translation module and the retrieval module is trained by this method, and the performance is significantly improved.

2.3 Translation-based and Retrieval-based Approaches

Translation-based approaches can understand the semantics of code changes and generate high-quality commit messages. But they tend to generate high frequency and repetitive tokens, and the generated commit messages have the problem of insufficient information and poor readability.

Retrieval-based approaches reuse commit messages of similar code changes. They can benefit from similar examples, but reused commit messages may not accurately describe the content and intent of current code changes.

Some studies also explore the combination of translation-based and retrieval-based approaches. Liu et al.[29] use a convolutional neural network to select the results generated by translation-based and retrieval-based approaches. However, this network structure is very complex and the classification effect is poor. Wang et al.[55] use the retrieved information and new code changes as model input to generate the commit message together. Shi et al.[47] propose a retrieval-augmented approach that retrieves a similar commit message as an exemplar and guides the neural model to generate commit messages. However, the retrieval results may play a negative role if there are no similar code changes in the training dataset.

In order to take advantage of both translation-based and retrieval-based approaches, we use a decision module to select the final result. In this way, the two approaches generate the commit messages independently without interfering with each other. The support vector machine and thresholds in the decision module are trained and obtained based on the parallel validation dataset, so they can better fit the characteristics of the dataset.

3 APPROACH

3.1 Framework Overview

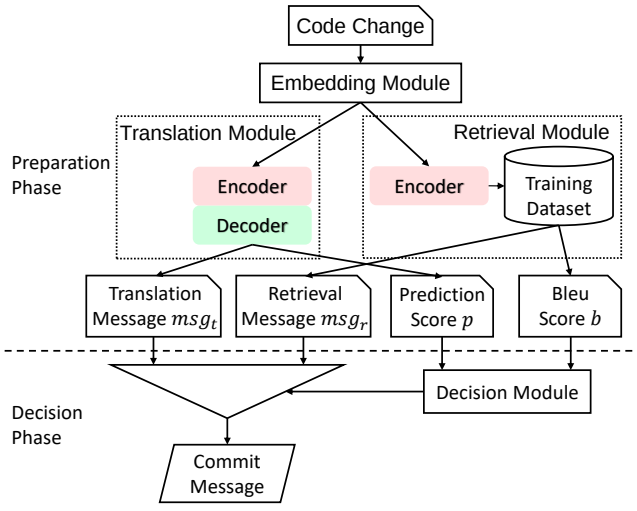

Figure 3: An overview workflow of COME framework.

In this section, we will introduce our approach COME, a novel framework for COmmit message generation using Modification Embedding. As shown in Figure 3, COME consists of two phases. The preparation phase, which includes the embedding module, the translation module and the retrieval module, is used to generate retrieval results and translation results. The decision phase uses the decision module to select a result as the final output. Next, we will introduce the four modules in detail.

3.2 Embedding Module

Code change information is very important for the commit message generation task. In order to highlight edit operations and represent code changes concisely at the same time, we design a novel modification embedding that rearranges the added and deleted code according to the edit distance algorithm. Edit distance refers to the minimum number of edit operations required to convert one string to another. Given two sentences A and B, the edit distance between them is the minimum number of delete and add operations to change A to B.

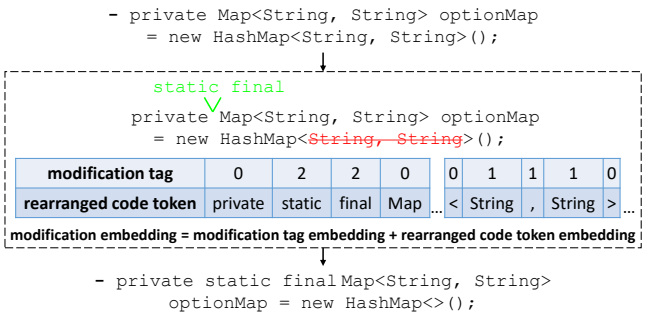

Figure 4: Generation of the modification embedding. Code changes are rearranged and modification tag is set to each rearranged token according to the edit distance algorithm.

Figure 4 shows the modification embedding generation process in the embedding module. For the code changes shown in Figure 2(a), the embedding module first uses the edit distance algorithm to find out the edit operations between the deleted code and the added code. That is, add "static" and "final" and delete "String", "," and "String". Then, it rearranges the input code tokens and sets the modification tag for each of them. The modification tag of the deleted token is set to 1, the added token is set to 2, and the unchanged token is set to 0. The modification embedding is the sum of the rearranged code token embedding and modification tag embedding. Specifically, we represent each token $r_i \in \mathbb{R}^{v_1}$ and tag $t_i \in \mathbb{R}^{v_2}$ as one-hot vectors, then the modification embedding $E_m \in \mathbb{R}^{l \times d}$ can be obtained as follows:

$$E_r = rW_r$$
$$E_t = tW_t$$
$$E_m = E_r + E_t$$

where $W_r \in \mathbb{R}^{v_1 \times d}$ and $W_t \in \mathbb{R}^{v_2 \times d}$ are two trainable matrices, and v_1 and v_2 denote to the vocabulary size and number of different modification tags respectively.

3.3 Translation Module

In the translation module, we use an encoder-decoder neural network as the backbone model. Although modification embedding has represented code changes in a fine grained way, it does not contain contextual semantic information of them. So in the first stage, the model is trained to learn contextualized code change representation. Based on this, the model is then trained to generate commit messages in the second stage. At the end of translation module, we get a commit message msg_t and a predict score s that representing its confidence for each code change.

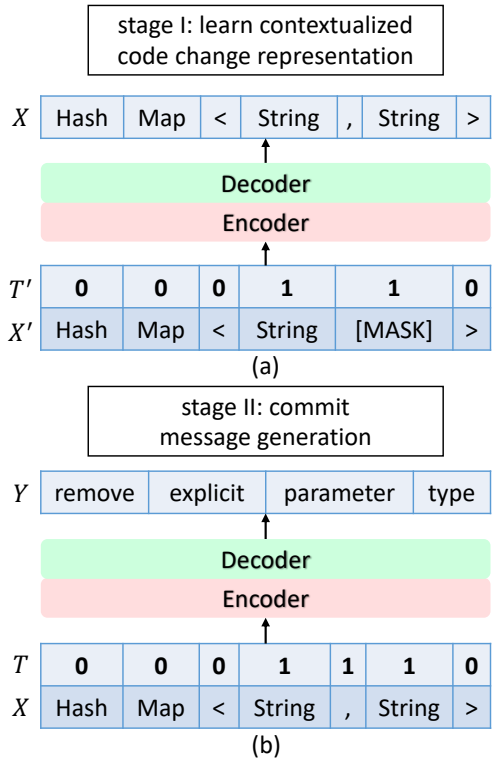

Figure 5: The two stages of generating a commit message. (a) the self-supervised generative task for learning contextualized code change representation; (b) the generation of commit messages using the model parameters trained in the first stage.

3.3.1 Contextualized Code Change Representation. Unlike previous works that directly use the pre-trained models with contextualized code representation[23, 47], we design a self-supervised generative task to learn the contextualized code change representation in the first training stage. As shown in Figure 5(a), the entire encoder-decoder neural network is optimized with back-propagation. Given a pair of rearranged code token sequence and modification tag sequence, we add some noise to it and use the noise-added data to predict the original data. To add noise, a token span is randomly sampled in each deleted or added code block, with the span length drawn from a Poisson distribution($\lambda = 3$). Each span is replaced by a single "[MASK]" token, and a "[MASK]" token is actually inserted when it comes to a span with a length of zero. The corresponding modification tag of the "[MASK]" token is set to 1 (for deleted code) or 2 (for added code).

The model is trained with the noise-added modification embedding sequences $X' + T'$ to predict the rearranged code token sequences X. The log likelihood function is used as the objective function:

$$\mathcal{L}_1 = -\frac{1}{|\mathbb{X}|}\sum_{i=1}^{|\mathbb{X}|}\sum_{j}\log\mathcal{P}(X_{ij}|X'_i + T'_i)$$

where X'_i and T'_i represent the ith noise-added rearranged code token sequence and its modification tag sequence respectively, and X_{ij} represents the jth token of the ith rearranged code token sequence in the dataset $\mathbb{X}$.

This task only uses code changes in the training dataset without additional supervision information, and does not need to distinguish between explicit and implicit code changes. In addition, unlike the delete-to-add method[40], this method allows the representation to fuse the deleted and added information, which allows us to obtain a deep bidirectional representation.

3.3.2 Commit Message Generation. As shown in Figure 5(b), in the second training stage, the model with contextual semantic information of code changes obtained in the first stage is fine-tuned to predict the commit message sequences Y by the modification embedding sequences $X + T$. The log likelihood function is used as the objective function:

$$\mathcal{L}_2 = -\frac{1}{|\mathbb{X}|}\sum_{i=1}^{|\mathbb{X}|}\sum_{j}\log\mathcal{P}(Y_{ij}|X_i + T_i;\theta)$$

where X_i and T_i represent the ith rearranged code token sequences and its modification tag sequences respectively, Y_{ij} represents the jth token of the ith commit message to be generated in the dataset $\mathbb{X}$ and θ represents the model parameters that have been trained in the first stage.

3.4 Retrieval Module

In the retrieval module, the encoder trained in the translation module is used to capture the features of the code changes and encodes them into a high-dimensional vector. And we get the most semantically similar code changes from the parallel training dataset by comparing the cosine similarity of vectors, and reuse its commit message as retrieval result.

Specifically, the input code changes represented by modification embedding $X = (x_1, x_2, ..., x_l)$ are first encoded to $Z = (z_1, z_2, ..., z_l)$ and the semantic vector is obtained by a dimension-wise max pooling operation:

$$vec = pooling(Z) = max(z_1, z_2, ..., z_l)$$

Notably, the dimension of this vector is much smaller than that of the bag-of-words vector used in [30], greatly improving the retrieval efficiency. Then, we obtain ten most similar code changes from the parallel training dataset by calculating cosine similarity of their semantic vectors. After that, the BLEU scores between X and the candidate training code changes are calculated, and the commit message of the code changes with the highest BLEU score b is selected as the retrieval result msg_r.

3.5 Decision Module

So far, for each code change data in validation and test datasets, we have obtained two candidate commit messages, msg_r from the retrieval model with its BLEU score b, and msg_t from the translation model with its prediction score s. In the decision module, we propose an algorithm and a support vector machine (SVM) to select the final commit message.

Algorithm 1 Decision Algorithm

Input: VALID SET (msg_t, msg_r, s, b, $ground\_truth$)
TEST SET (msg_t, msg_r, s, b)
Output: commit message $output$ for each data in TEST SET

1: **function** PREDICT($threshold_1$, $threshold_2$, SVM, $DataSet$)
2: **for** (msg_t, msg_r, s, b) $\in$ $DataSet$ **do**
3: **if** $b < threshold_1$ **then**
4: msg_o.ADD(msg_t)
5: **else if** $s < threshold_2$ **then**
6: msg_o.ADD(msg_r)
7: **else**
8: msg_o.ADD($SVM(msg_t, msg_r, s, b)$)
9: **end if**
10: **end for**
11: **return** msg_o
12: **end function**
13:
14: $best\_b \leftarrow 0$
15: **for** $t_1 \in \{0.5, 0.51, ..., 0.9\}, t_2 \in \{-0.8, -0.79, ..., -0.3\}$ **do**
16: **for** (msg_t, msg_r, s, b, $ground\_truth$) $\in$ VALID SET **do**
17: $b_1 \leftarrow$BLEU(msg_t, $ground\_truth$)
18: $b_2 \leftarrow$BLEU(msg_r, $ground\_truth$)
19: **if** $b_1, b_2 > 0.001$ && $b > t_1$ && $s > t_2$ **then**
20: **if** $b_1 < b_2$ **then** $positive\_example$.ADD(b, s)
21: **else** $negative\_example$.ADD(b, s)
22: **end if**
23: **end if**
24: **end for**
25: $SVM \leftarrow$TrainSVM($positive\_example$, $negative\_example$)
26: $all_b \leftarrow$BLEU(VALID SET.$ground\_truth$, PREDICT(t_1, t_2, SVM, VALID SET))
27: **if** $all\_b > best\_b$ **then**
28: $best\_b \leftarrow all\_b$
29: $best\_t_1, best\_t_2, best\_SVM \leftarrow t_1, t_2, SVM$
30: **end if**
31: **end for**
32: $output \leftarrow$PREDICT($best\_t_1$, $best\_t_2$, $best\_SVM$, TEST SET)

We describe our decision algorithm in Algorithm 1. In the decision module, an SVM classifier and two thresholds for retrieval result's BLEU score b and translation result's prediction score s are required. As the PREDICT function shown in lines 1-12, if either b or s is lower than the threshold, the commit message generated by the other approach is selected as the final result. Otherwise, the SVM classifier is utilized for final decision. Lines 15-31 explain the generation of the best thresholds and SVM classifier on validation dataset. Given the thresholds t_1, t_2, lines 16-24 generate a training dataset and line 25 trains an SVM classfier on the generated dataset. Lines 26-30 compare the effectiveness of SVMs generated with different thresholds and save the best results $best\_t_1$, $best\_t_2$ and $best\_SVM$. Line 32 utilizes the best thresholds and SVM to decide the final commit messages on test dataset according to the PREDICT function.

4 EXPERIMENTAL EVALUATION

4.1 Datasets

We select two widely used datasets, MCMD and CoDiSum, to evaluate COME and state-of-the-art approaches.

MCMD[51] is a commit message generation dataset containing five programming languages: C++, C#, Java, Python and JavaScript. For each language, it collects commits from the top 100 starred repositories on GitHub. Shi et al.[47] further filter out commits with files that cannot be parsed (such as .jar, .ddl, .mp3, and .apk) to reduce noise data, and build a higher quality dataset, which is a subset of MCMD. Table 1 shows the statistics for the subset of MCMD we used in our experiment.

Table 1: Statistics of the evaluation dataset, which is a subset of MCMD.

Language	Training	Validation	Test
C++	160,948	20,000	20,141
C#	149,907	18,688	18,702
Java	160,018	19,825	20,159
Python	206,777	25,912	25,837
JavaScript	197,529	24,899	24,773

CoDiSum[62] is a benchmark dataset consisting of commits from the top 1000 popular Java projects on GitHub, excluding rollback and merge commits. This dataset contains 90,661 code change and commit message pairs in total, and has not been divided into training, validation and test sets in advance. For the fairness of the experiment, we divide it into 75,000 training data, 8,000 validation data and 7,661 test data according to the division method of the existing work[11]. The human evaluation is also conducted on this dataset.

4.2 State-of-the-art Approaches

We compare COME with ten state-of-the-art commit message generation approaches as follows.

- retrieval-based: NNGen[30] and Lucene[6];
- translation-based: CommitGen[22], CoreGen[40], CoDiSum[62] and FIRA[11];
- hybrid: RACE[47] and CoRec[55];
- pre-trained: CodeT5-base[57] and CommitBERT[23].

For implementation, if the approach uses the same dataset as ours and its generated commit messages are available, we directly reuse its results. Otherwise, we implement it strictly following the open source code or paper.

Table 2: Overall commit message generation results on the MCMD dataset. Met., Rou., and Cid. are short for METEOR, ROUGH-L, and CIDEr respectively.

Approach	Java				C#				C++				Python				JavaScript			
	Bleu	Met.	Rou.	Cid.	Bleu	Met.	Rou.	Cid.	Bleu	Met.	Rou.	Cid.	Bleu	Met.	Rou.	Cid.	Bleu	Met.	Rou.	Cid.
NNGen[30]*	19.41	12.40	25.15	1.23	22.15	14.77	26.46	1.55	13.61	9.39	18.21	0.73	16.06	10.91	21.69	0.92	18.65	12.50	24.45	1.21
Lucene[6]*	15.61	10.56	19.43	0.94	20.68	13.34	23.02	1.36	13.43	8.81	16.78	0.67	15.16	9.63	18.85	0.85	17.66	11.25	21.75	1.02
CommitGen[22]•★	14.07	7.52	18.78	0.66	13.38	8.31	17.44	0.63	11.52	6.98	16.75	0.45	11.02	6.43	16.64	0.42	18.67	11.88	24.10	1.08
CoDiSum[62]•★	13.97	6.02	16.12	0.39	12.71	5.56	14.40	0.36	12.44	6.00	14.39	0.42	14.61	8.59	17.02	0.42	11.22	5.32	13.26	0.28
CoRec[55]▲★	18.51	11.26	24.78	1.13	18.41	11.70	23.73	1.12	14.02	8.63	20.10	0.72	15.09	9.60	22.35	0.80	21.30	13.84	27.53	1.40
CoreGen[40]•★○	21.30	13.17	28.04	1.31	17.08	11.36	22.74	0.94	16.74	11.72	22.83	0.90	17.74	12.22	24.75	0.97	22.53	14.91	29.23	1.50
CommitBERT[23]•★○	22.32	12.63	28.03	1.42	20.67	12.31	25.76	1.25	16.16	10.05	19.90	0.94	17.29	11.31	22.36	1.01	23.40	15.64	30.51	1.54
CodeT5-base[57]•★○	22.76	14.57	30.23	1.43	22.21	14.51	29.08	1.33	16.73	11.69	22.86	0.85	17.99	12.74	25.27	0.96	22.87	15.12	29.81	1.50
RACE[47]▲★○	25.66	15.46	32.02	1.76	26.33	16.37	31.31	1.84	19.13	12.55	24.52	1.14	21.79	14.68	28.35	1.40	25.55	16.31	31.79	1.84
Retrieval Module	20.22	12.64	26.02	1.26	22.18	14.28	26.80	1.50	14.11	9.64	18.99	0.74	16.75	11.31	22.77	0.94	19.10	12.70	25.14	1.21
Translation Module	26.06	15.97	33.64	1.77	23.90	15.37	30.47	1.52	20.40	14.27	26.76	1.19	22.30	15.98	30.15	1.36	26.72	17.74	34.44	1.88
COME	**27.17**	**16.91**	**34.59**	**1.90**	**27.29**	**17.77**	**33.33**	**1.91**	**20.80**	**14.55**	**27.01**	**1.25**	**23.17**	**16.46**	**30.48**	**1.50**	**26.91**	**17.84**	**34.44**	**1.92**

Approach Classification: ∗ Retrieval−Based ; • Translation−Based ; ▲ Hybrid ; ★ Coarse Grained ; ○ Contextualized Code Representation

4.3 Evaluation Metrics

4.3.1 Automatic Metrics. We verify COME's effectiveness with automatic evaluation metrics that are widely used in natural language generation tasks, including BLEU, METEOR, ROUGE-L and CIDEr.

BLEU measures the precision of generated sequence by calculating its word precision from 1-*gram* to 4-*gram*, and penalizes overly short sentences[42]. BLEU is usually calculated at the corpus level, and is more similar to human judgments[27]. We also employ ROUGE-L, a recall-oriented metric based on longest common subsequence[26, 35]. In addition, we use METEOR[7] metric, which calculates harmonic mean of 1-*gram* precision and 1-*gram* recall of the generated sentence against the ground truth. The last automatic metric we use is CIDEr[53], which takes each sentence as a document and calculates the cosine similarity of its Term Frequency Inverse Document Frequency (TF-IDF) vector at *n-gram* level to obtain the similarity between the generated sentence and the ground truth.

4.3.2 Human Evaluation Metrics. Automatic metrics calculate text similarity between the generated message and the ground truth. However, these metrics do not consider syntax, grammar and sentence structures of the generated messages, so we conduct a human evaluation on the CoDiSum[62] dataset to further evaluate COME's usefulness in practice. We invite six Ph.D. students to participate in our survey, all of whom are not co-authors, major in computer science and have industrial experience in Java programming for more than 3 years. We randomly select 100 commits from the test set and design a questionnaire for participants. For each commit, code changes and commit messages generated by different approaches are given. Each participant scores 50 commits, and each commit is scored by 3 participants. We shuffle commit messages generated by different approaches, so participants do not know which approach each commit message comes from.

Each participant is asked to score the commit messages from three aspects: relevance, usefulness, and content adequacy. Relevance refers to the correlation between commit messages and code changes. Usefulness refers to how helpful commit messages are for understanding code changes. Content adequacy refers to whether the commit messages include all information about code changes. Scores range from 1 to 5, and a commit message with higher scores suggests better relevance, usefulness and content adequacy. The final score of each commit message is calculated with the average of three scores.

4.4 Experimental Settings

Experiments are conducted using different sets of hyperparameters to optimize COME's performance on the benchmark datasets. While training the encoder-decoder neural network, we use the weight of the CodeT5-base[57] to initialize our model. The original vocabulary size of CodeT5 is 32,100. We adopt the AdamW optimizer[31] with 5e-5 learning rate and the batch size is 12. The max epoch is 5 for the first training stage and 10 for the second. For the support vector machine in the decision module, the Radial Basis Function Kernel is selected and γ is set to 20.

All experiments are performed on a Dell workstation with Intel Xeon Gold 6130 CPU @ 2.10GHz, running Debian 5.4.143.bsk. The models are trained on one 32G GPU of NVIDIA Tesla v100.

4.5 Experimental Results

The experiments we conduct answer the following research questions.

RQ1: Overall effectiveness of COME. How does COME perform compared with the state-of-the-art commit message generation approaches?

RQ2: Effectiveness of the decision module. How does the decision module benefit to COME?

RQ3: Effectiveness of the modification embedding. How does modification embedding benefit to COME?

RQ4: Effectiveness of the contextualized code change representation. How does contextualized code change representation benefit to COME?

RQ1: *Overall Effectiveness of COME.* The overall effectiveness of each approach is measured by automatic metrics and human evaluation.

Table 3: Overall commit message generation results on the CoDiSum dataset.

Approach	Bleu	Met.	Rou.	Cid.
NNgen[30]*	9.20	5.19	11.32	0.31
Lucene[6]*	8.97	4.31	10.53	0.20
CommitGen[22]●★	6.28	3.74	8.91	0.13
CoDiSum[62]●★	16.56	7.15	19.05	0.45
CoRec[55]▲★	13.06	6.44	15.37	0.38
CoreGen[40]●★○	14.10	6.79	17.92	0.38
RACE[47]▲★○	17.23	8.57	20.49	0.57
CommitBERT[23]●★○	17.44	9.10	21.64	0.64
CodeT5-Base[57]●★○	17.70	9.30	22.01	0.66
FIRA[11]●△	17.66	8.31	20.90	0.56
Retrieval Module	10.01	5.40	12.29	0.30
Translation Module	19.47	10.49	24.39	0.79
COME	**19.64**	**10.70**	**24.56**	**0.82**

Approach Classification:
∗ Retrieval−Based ; ● Translation−Based ; ▲ Hybrid ;
★ Coarse Grained ; ○ Contextualized Code Representation ;
△ abstract syntax tree ;

Automatic Metrics. As shown in Table 2 and Table 3, COME outperforms all state-of-the-art approaches on all automatic metrics. (Note that FIRA[11] requires special data preprocessing that is not available, so we can only obtain its results on the CoDiSum dataset.)

Besides, our retrieval module outperforms other retrieval-based approaches, and our translation module outperforms most state-of-the-art approaches. COME also performs better than using the translation module or retrieval module only.

Table 4: Human evaluation on commit messages generated by different approaches on the CoDiSum dataset.

Approach	Total Number			Average Sore
	Low	Medium	High	
NNgen[30]	37	46	17	2.71
Lucene[6]	39	48	13	2.65
CommitGen[22]	45	40	15	2.17
CoDiSum[62]	21	48	31	3.30
Corec[55]	43	39	18	2.69
CoreGen[40]	38	35	27	2.92
RACE[47]	20	49	31	3.31
CommitBERT[23]	17	52	31	3.37
CodeT5-Base[57]	16	53	31	3.39
FIRA[11]	29	42	29	3.25
COME	**14**	**46**	**40**	**3.66**

Human Evaluation. Table 4 shows the results of the human evaluation. The performance of each approach is measured by the average score of all its generated commit messages. Following previous work[11, 30, 55], a score of 1 or 2 is considered as low quality, a score of 3 as medium quality, and a score of 4 or 5 as high quality.

COME generates the highest proportion of high-quality commit messages (40%) and the lowest proportion of low-quality commit messages (14%). It also gets the best average score (3.66), which is 8% higher than the second-best CodeT5-Base[57]. We also perform a Wilcoxon signed-rank test[59] with the Bonferroni correction[1] between the scores of COME and other approaches. The results further confirm that the difference is statistically significant at the confidence level of 98%. Overall, from the human evaluation, we conclude that COME outperforms state-of-the-art approaches and generates commit messages with higher grammatical and semantic rationality.

RQ2: *Effectiveness of the decision module.* As shown in Table 2 and Table 3, retrieval-based approaches, such as NNGen[30] and Lucene[6], perform quite while on the MCMD dataset, especially on the MCMD-C# dataset. However, their performance is much worse on the CoDiSum dataset. It indicates that the similarity of training and test sets has a great impact on the results of these approaches.

Similar to these retrieval-based approaches, for most datasets with low similarity between training and test sets, the results of our retrieval module are poor compared with our translation module. In this way, the retrieval module contributes little to the final decision, but the performance of COME still improves slightly compared with the translation module. And when the retrieval result is as good as the translation result, performance can increase significantly after our decision phase, such as 14% on the MCMD-C# dataset.

RQ3: *Effectiveness of the modification embedding.* Several approaches[22, 23, 40, 47, 55, 57, 62] directly represent code changes with text sequences and we regard them as coarse grained approaches. As shown in Table 3, the performance of FIRA which represents code changes with abstract syntax tree is better than most coarse grained approaches, indicating that fine grained representation really helps to generate high-quality commit messages. COME still outperforms FIRA, because modification embedding not only represents code changes in a fine grained way, but also maintains the input data as a sequence instead of a graph, thus avoiding the use of complex models such as Graph Neural Network.

We also perform an ablation study to further validate the effectiveness of modification embedding and contextualized code change representation in the translation neural network.

As shown in Table 5, we build two variants of the translation neural network to validate the effect of two components. Translation- - is the first variant that excludes modification embedding and contextualized code change representation. Translation- is the second variant that excludes contextualized code change representation. Translation is what we use in the translation module with both modification embedding and contextualized code change representation included. The improvment of performance from Translation- - to Translation- validates the effectiveness of the modification embedding.

Table 5: The ablation study for translation neural network.

Language	Approach	Bleu	Met.	Rou.	Cid.
MCMD Java	Translation- -	22.76	14.57	30.23	1.43
	Translation-	24.02	14.66	31.07	1.59
	Translation	26.06	15.97	33.64	1.77
MCMD C#	Translation- -	22.21	14.51	29.08	1.33
	Translation-	23.48	15.11	29.92	1.48
	Translation	23.90	15.37	30.47	1.52
MCMD C++	Translation- -	16.73	11.69	22.86	0.85
	Translation-	18.77	13.27	24.16	1.06
	Translation	20.40	14.27	26.76	1.19
MCMD Python	Translation- -	17.99	12.74	25.27	0.96
	Translation-	20.13	14.47	27.09	1.16
	Translation	22.30	15.98	30.15	1.36
MCMD JavaScript	Translation- -	22.87	15.12	29.81	1.50
	Translation-	24.22	16.14	31.47	1.69
	Translation	26.91	17.84	34.44	1.92
CoDiSum Java	Translation- -	17.70	9.30	22.01	0.66
	Translation-	18.40	9.88	22.96	0.71
	Translation	19.47	10.49	24.39	0.79

Figure 6: A case study. (a) code changes; (b) representations of the circled code changes and generated commit messages of Translation- - and Translation-.

To further explain the improvement, we provide a case study. As the code changes shown in Figure 6, only the class name is changed from "LambdaTest" to "LambdaIT", but Translation- - represents it with a lot of duplicated information and generates an inaccurate commit message. However, Translation- represents code changes with modification embedding, which helps the model capture the true motivation for code changes.

RQ4: *Effectiveness of the contextualized code change representation.* As shown in Table 2 and Table 3, approaches with contextualized code representation[23, 47, 57] generally outperform other baselines, but still underperform our COME framework. It indicates that while contextualized code representation is helpful in the commit message generation task, the contextualized code change representation we propose can do better. The improvement of performance from Translation- to Translation in Table 5 also validates the effectiveness of the contextualized code change representation.

To further verify the ability of contextualized code change representation, we apply it to another downstream task, just-in-time (JIT) defect prediction. JIT defect prediction aims to identify defective patches, which provides early feedback to software developers to optimize their inspection efforts, and has been widely used in large software companies[37, 48, 50].

Following the state-of-the-art DeepJIT[16] approach, JIT defect prediction is modeled as a binary classification task, in which each data is labeled 1 or 0 to indicate whether it contains defects or not. Given a pair of code change and commit message, the model outputs a score to predict the probability that the input data contains defects. As shown in Figure 7, an encoder-decoder network is trained by the self-supervised generative task we propose to learn contextualized code change representation. Then the CNN in the original DeepJIT model is replaced by the well-trained encoder, and the model with replaced encoders is used as our backbone. Two encoders extract the features of code changes and commit messages respectively by max pooling the output vectors. Finally, the feature fusion layer takes the extracted features as input and output the probability of containing defects. We conduct the experiments on the QT and OPENSTACK datasets, which are used by Hoang et al.[16]. The results in Table 6 prove that contextualized code change representation is also useful for the JIT defect prediction task.

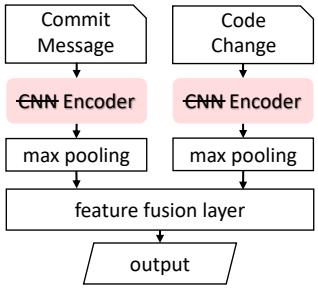

Figure 7: DeepJIT model with CNN replaced by the encoder.

Table 6: The Area Under Curve (AUC) results of different JIT defect prediction approaches.

Approach	QT	OPENSTACK
JIT	0.701	0.691
DBNJIT	0.705	0.694
DeepJIT	0.768	0.751
COME	**0.787**	**0.798**

5 DISCUSSION

5.1 Threats to Validity

The internal threats to validity include possible errors in implementing state-of-the-art approaches and setting hyperparameters. To alleviate the threats, we directly use the experimental results and code published by these approaches if available, or carefully reproduce them according to the descriptions in their papers. And our experimental results are consistent with the results mentioned in their papers.

The external threats to validity lie in the datasets and metrics used in the experiment. To mitigate these threats, we conduct experiments on two well-established datasets MCMD and CoDiSum. We also adopted four automatic metrics (BLEU, METEOR, ROUGE-L and CIDEr) that are widely used in previous commit message generation works. In addition, we conduct a human evaluation to verify the grammatical and semantic rationality of the generated messages.

But human evaluation may lead to another threat. That is, participants have different standards while scoring commit messages. To reduce the threat, we show our standards in detail with some specific examples in advance. And six participants instead of three are invited, which is more reliable according to Wang et al.[55]. After scoring, we evaluate the quality of each commit message by averaging the scores of participants to reduce the deviation.

5.2 Limitations

There are three main limitations in COME. First, it takes a long time to train the model because of two training stages. Second, long code changes are truncated to 512 tokens, so the information of code changes longer than 512 tokens may be lost. Third, it is impossible to generate out-of-vocabulary tokens in commit messages.

6 RELATED WORK

6.1 Commit Message Generation

Commit message generation approaches are mainly divided into four categories: rule-based approaches[8, 9, 46, 52], retrieval-based approaches[17, 20, 30], translation-based approaches[11, 22, 28, 32, 40, 62] and hybrid approaches[29, 47, 55].

Rule-based approaches extract information from code changes and translate it into natural language according to predefined rules or templates[8, 9, 46, 52]. However, they can only handle certain predefined types of code changes and do not cover every circumstance, so the generated commit messages always fail to explain the reason and purpose of code changes.

Retrieval-based approaches find the most similar code changes in the training dataset and reuse their commit messages[20, 30, 47]. Liu et al.[30] calculate cosine similarity of bag-of-words vectors to find similar code changes. Shi et al.[47] use commit message generation data to train an encoder-decoder model, then use the output of the encoder as the high-dimensional vector representation of the code changes and finally retrieve the similar code changes by calculating cosine similarity. Despite their impressive effectiveness and efficiency, retrieval-based approaches always generate inaccurate messages when there are no similar differences in the corpus.

Translation-based approaches use neural networks to translate code changes into commit messages. Jiang et al.[22] and Loyola et al.[32] use the LSTM[18] neural machine translation model to automatically generate commit messages. Liu et al.[28] use a pointer generation network[45] to solve the out of vocabulary problems. Xu et al.[62] extract the code structure by identifying the identifiers and replace each of them with a uniform token. For each identifier, the model also learns its semantic representation, and combines it with the corresponding uniform token as the overall representation. Nie et al.[40] use the original transformer structure as the backbone model and design two different pre-training tasks for explicit code changes and implicit binary files to learn the context representation of input code changes. Dong et al.[11] represent code changes with a fine grained abstract syntax tree, and use graph neural networks to extract features and generate commit messages.

Hybrid approaches combine the retrieval-based and translation-based approaches to generate commit messages. Wang et al.[55] retrieve the most similar code changes in the training set and fuse the output of origin and retrieval code changes to generate the commit message. Shi et al.[47] translate code changes into commit messages with a neural machine translation model guided by similar commit messages. Liu et al.[29] use an abstract syntax tree to represent code changes, and a CNN hybrid sorting module is trained to select results generated by retrieval-based and translation-based approaches. However, in these hybrid approaches, if the fusion method is not properly designed, retrieval-based and translation-based approaches may affect each other and reduce the quality of the final results.

6.2 Code Summarization

Code summarization aims to generate short descriptions for code snippets in natural language, and mainly includes three categories: rule-based approaches[38, 49], retrieval-based approaches[15, 34, 60, 61] and learning-based approaches[5, 19, 21, 41, 43, 54].

Rule-based approaches extract useful information from source code and generate summaries following predefined rules. Sridhara et al.[49] design a framework that can identify significant statements from a Java method and then transform them into natural language using predefined templates. Moreno et al.[38] use summarization tools and heuristic algorithms to select information from code and use existing lexicalization tools to generate summaries.

Information retrieval methods are widely used in summary generation tasks. Wong et al.[60] apply code clone detection techniques to discover similar code segments and use natural language processing techniques to select relevant comment sentences. McBurney et al.[34] select keywords and topics based on topic modeling, establish a hierarchy in which the more general topics are near the top, and use top topics as code summaries. Wong et al.[61] get code and descriptions from StackOverflow and reuse descriptions of similar code snippets to automatically generate code summaries.

In addition, some researchers use learning-based approaches to generate code summaries. Iyer et al.[21] use an LSTM encoder-decoder network with attention mechanism to generate descriptions for C# and SQL code. Wan et al.[54] generate code summaries using a deep reinforcement learning framework that includes an abstract syntax tree structure and sequential code snippets. Hu

et al.[19] use an external dataset to train an API sequence encoder and generate code summaries using an encoder-decoder model with the learned representation. Alon et al.[5] use the syntactic structure of programming languages to better encode source code and use the attention mechanism to select relevant paths while decoding. Oda et al.[41] use statistical machine translation to automatically learn the relationship between source code and pseudo code to improve code readability. Phan et al.[43] treat source code and documentations as two levels of abstraction for the same intent and use statistical machine translation to translate between them.

Unlike code summarization, commit message generation aims to summarize changed information in code snippets instead of semantic structure information for an entire function or method.

6.3 Code Representation Learning

In traditional machine learning algorithms, code are treated as sequences of tokens and n-gram language models are widely used to model source code[2, 4, 14]. With the development of deep learning, Raychev et al.[44] use RNN models to learn code features together with downstream tasks. Wang et al.[56] and White et al.[58] transform abstract syntax trees into vectors to learn syntactical and semantic code information. Mou et al.[39] design a tree-based convolutional neural network to capture structural information from abstract syntax trees. Zhang et al.[63] split each large abstract syntax tree into many small statement trees, and encode them into vectors by capturing lexical and syntactical knowledge of statements. Then a bidirectional RNN model is used to produce the vector representation of a code fragment. Allamanis et al.[3] represent the syntactic and semantic structure of the source code using a gated graph neural network. Following the success of the pre-trained language models, SCELMo[25], CodeBERT[13] and CodeT5[57] pre-train code representations on large unlabeled corpus.

Compared with these methods, our contextualized code change representation aims to learn context information of code changes which is helpful for generating high-quality commit messages from incomplete code snippets without using abstract syntax trees or constructed graphs.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose COME, a framework for commit message generation with three main innovations. First, the modification embedding is used to highlight edit operations of code changes and avoid duplicated information. Second, a self-supervised generative task is designed to learn the contextualized code change representation. Third, a decision algorithm is used to select the final commit message from the translation and retrieval results.

We conduct evaluations on two widely used datasets, and COME outperforms all state-of-the-art approaches on four automatic evaluation metrics (BLEU, METEOR, ROUGE-L and CIDEr) and human evaluation. We further validate the effectiveness of three important components in COME including the decision module, modification embedding and contextualized code change representation.

In our experiment, we use only a small amount of data in the existing dataset to get contextual semantic information of code changes. In future work, researchers can use large amounts of code change data to obtain a pre-trained model with more general representation and fine-tune it on the downstream tasks that use code changes as input. In addition, the current commit message generation datasets are built using only commit information crawled from GitHub's top code repositories, which leads to weak semantic relevance between code changes and commit messages in some data. This emphasizes the demand for richer datasets that include more information (issue, comment, etc.).

REFERENCES

[1] Hervé Abdi et al. 2007. Bonferroni and Šidák corrections for multiple comparisons. *Encyclopedia of measurement and statistics* 3 (2007), 103–107.

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 281–293. https://doi.org/10.1145/2635868.2635883

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BJOFETxR-

[4] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim (Eds.). IEEE Computer Society, 207–216. https://doi.org/10.1109/MSR.2013.6624029

[5] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=H1gKYo09tX

[6] Apache. 2011. Apache Lucene. https://lucene.apache.org/

[7] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005*, Jade Goldstein, Alon Lavie, Chin-Yew Lin, and Clare R. Voss (Eds.). Association for Computational Linguistics, 65–72. https://aclanthology.org/W05-0909/

[8] Raymond P. L. Buse and Westley Weimer. 2010. Automatically documenting program changes. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 33–42. https://doi.org/10.1145/1858996.1859005

[9] Luis Fernando Cortes-Coy, Mario Linares Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On Automatically Generating Commit Messages via Summarization of Source Code Changes. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*. IEEE Computer Society, 275–284. https://doi.org/10.1109/SCAM.2014.14

[10] Brian de Alwis and Jonathan Sillito. 2009. Why are software projects moving from centralized to decentralized version control systems?. In *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering, CHASE 2009, Vancouver, BC, Canada, 17 May 2009*. IEEE Computer Society, 36–39. https://doi.org/10.1109/CHASE.2009.5071408

[11] Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. 2022. FIRA: Fine-Grained Graph-Based Code Change Representation for Automated Commit Message Generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 970–981. https://doi.org/10.1145/3510003.3510069

[12] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 422–431. https://doi.org/10.1109/ICSE.2013.6606588

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

[14] Georgia Frantzeskou, Stephen G. MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis. 2008. Examining the significance of high-level programming features in source code author classification. *J. Syst. Softw.* 81, 3 (2008), 447–460. https://doi.org/10.1016/j.jss.2007.03.004

[15] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 223–226. https://doi.org/10.1145/1810295.1810335

[16] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 34–45. https://doi.org/10.1109/MSR.2019.00016

[17] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: distributed representations of code changes. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 518–529. https://doi.org/10.1145/3377811.3380361

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[19] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, Jérôme Lang (Ed.). ijcai.org, 2269–2275. https://doi.org/10.24963/ijcai.2018/314

[20] Yuan Huang, Qiaoyang Zheng, Xiangping Chen, Yingfei Xiong, Zhiyong Liu, and Xiaonan Luo. 2017. Mining Version Control System for Automatically Generating Commit Comment. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, Ayse Bener, Burak Turhan, and Stefan Biffl (Eds.). IEEE Computer Society, 414–423. https://doi.org/10.1109/ESEM.2017.56

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics. https://doi.org/10.18653/v1/p16-1195

[22] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 135–146. https://doi.org/10.1109/ASE.2017.8115626

[23] Tae-Hwan Jung. 2021. CommitBERT: Commit Message Generation Using Pre-Trained Programming Language Model. *CoRR* abs/2105.14242 (2021). arXiv:2105.14242 https://arxiv.org/abs/2105.14242

[24] Mira Kajko-Mattsson. 2005. A Survey of Documentation Practice within Corrective Maintenance. *Empir. Softw. Eng.* 10, 1 (2005), 31–55. https://doi.org/10.1023/B:LIDA.0000048322.42751.ca

[25] Rafael-Michael Karampatsis and Charles Sutton. 2020. SCELMo: Source Code Embeddings from Language Models. *CoRR* abs/2004.13214 (2020). arXiv:2004.13214 https://arxiv.org/abs/2004.13214

[26] Chin-Yew Lin and Franz Josef Och. 2004. Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-Bigram Statistics. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics, 21-26 July, 2004, Barcelona, Spain*, Donia Scott, Walter Daelemans, and Marilyn A. Walker (Eds.). ACL, 605–612. https://doi.org/10.3115/1218955.1219032

[27] Chia-Wei Liu, Ryan Lowe, Iulian Serban, Michael D. Noseworthy, Laurent Charlin, and Joelle Pineau. 2016. How NOT To Evaluate Your Dialogue System: An Empirical Study of Unsupervised Evaluation Metrics for Dialogue Response Generation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, Jian Su, Xavier Carreras, and Kevin Duh (Eds.). The Association for Computational Linguistics, 2122–2132. https://doi.org/10.18653/v1/d16-1230

[28] Qin Liu, Zihe Liu, Hongming Zhu, Hongfei Fan, Bowen Du, and Yu Qian. 2019. Generating commit messages from diffs using pointer-generator network. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 299–309. https://doi.org/10.1109/MSR.2019.00056

[29] Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. 2022. ATOM: Commit Message Generation Based on Abstract Syntax Tree and Hybrid Ranking. *IEEE Trans. Software Eng.* 48, 5 (2022), 1800–1817. https://doi.org/10.1109/TSE.2020.3038681

[30] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 373–384. https://doi.org/10.1145/3238147.3238190

[31] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=Bkg6RiCqY7

[32] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 2: Short Papers*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 287–292. https://doi.org/10.18653/v1/P17-2045

[33] Walid Maalej and Hans-Jörg Happel. 2010. Can development work describe itself?. In *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, Jim Whitehead and Thomas Zimmermann (Eds.). IEEE Computer Society, 191–200. https://doi.org/10.1109/MSR.2010.5463344

[34] Paul W. McBurney, Cheng Liu, Collin McMillan, and Tim Weninger. 2014. Improving topic model source code summarization. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, Chanchal K. Roy, Andrew Begel, and Leon Moonen (Eds.). ACM, 291–294. https://doi.org/10.1145/2597008.2597793

[35] Shamima Mithun, Leila Kosseim, and Prasad Perera. 2012. Discrepancy Between Automatic and Manual Evaluation of Summaries. In *Proceedings of Workshop on Evaluation Metrics and System Comparison for Automatic Summarization@NACCL-HLT 2012, Montrèal, Canada, June 2012, 2012*, John M. Conroy, Hoa Trang Dang, Ani Nenkova, and Karolina Owczarzak (Eds.). Association for Computational Linguistics, 44–52. https://aclanthology.org/W12-2606/

[36] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes using Historic Databases. In *2000 International Conference on Software Maintenance, ICSM 2000, San Jose, California, USA, October 11-14, 2000*. IEEE Computer Society, 120–130. https://doi.org/10.1109/ICSM.2000.883028

[37] Audris Mockus and David M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Tech. J.* 5, 2 (2000), 169–180. https://doi.org/10.1002/bltj.2229

[38] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. IEEE Computer Society, 23–32. https://doi.org/10.1109/ICPC.2013.6613830

[39] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, Dale Schuurmans and Michael P. Wellman (Eds.). AAAI Press, 1287–1293. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775

[40] Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. 2021. CoreGen: Contextualized Code Representation Learning for Commit Message Generation. *Neurocomputing* 459 (2021), 97–107. https://doi.org/10.1016/j.neucom.2021.05.039

[41] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 574–584. https://doi.org/10.1109/ASE.2015.36

[42] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 311–318. https://doi.org/10.3115/1073083.1073135

[43] Hung Phan, Hoan Anh Nguyen, Tien N. Nguyen, and Hridesh Rajan. 2017. Statistical Learning for Inference between Implementations and Documentation. In *39th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track, ICSE-NIER 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 27–30. https://doi.org/10.1109/ICSE-NIER.2017.9

[44] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 419–428. https://doi.org/10.1145/2594291.2594321

[45] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 1073–1083. https://doi.org/10.18653/v1/P17-1099

[46] Jinfeng Shen, Xiaobing Sun, Bin Li, Hui Yang, and Jiajun Hu. 2016. On Automatic Summarization of What and Why Information in Source Code Changes. In *40th IEEE Annual Computer Software and Applications Conference, COMPSAC 2016,*

Atlanta, GA, USA, June 10-14, 2016. IEEE Computer Society, 103–112. https://doi.org/10.1109/COMPSAC.2016.162

[47] Ensheng Shi, Yanlin Wang, Wei Tao, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. RACE: Retrieval-augmented Commit Message Generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, 5520–5530. https://aclanthology.org/2022.emnlp-main.372

[48] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An industrial study on the risk of software changes. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 62. https://doi.org/10.1145/2393596.2393670

[49] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 43–52. https://doi.org/10.1145/1858996.1859006

[50] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, Akinori Ihara, and Ken-ichi Matsumoto. 2015. The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 812–823. https://doi.org/10.1109/ICSE.2015.93

[51] Wei Tao, Yanlin Wang, Ensheng Shi, Lun Du, Shi Han, Hongyu Zhang, Dongmei Zhang, and Wenqiang Zhang. 2021. On the Evaluation of Commit Message Generation Models: An Experimental Study. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*. IEEE, 126–136. https://doi.org/10.1109/ICSME52107.2021.00018

[52] Mario Linares Vásquez, Luis Fernando Cortes-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. ChangeScribe: A Tool for Automatically Generating Commit Messages. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 709–712. https://doi.org/10.1109/ICSE.2015.229

[53] Ramakrishna Vedantam, C. Lawrence Zitnick, and Devi Parikh. 2015. CIDEr: Consensus-based image description evaluation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 4566–4575. https://doi.org/10.1109/CVPR.2015.7299087

[54] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 397–407. https://doi.org/10.1145/3238147.3238206

[55] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. 2021. Context-aware Retrieval-based Deep Commit Message Generation. *ACM Trans. Softw. Eng. Methodol.* 30, 4 (2021), 56:1–56:30. https://doi.org/10.1145/3464689

[56] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 297–308. https://doi.org/10.1145/2884781.2884804

[57] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[58] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 87–98. https://doi.org/10.1145/2970276.2970326

[59] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[60] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. CloCom: Mining existing source code for automatic comment generation. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik (Eds.). IEEE Computer Society, 380–389. https://doi.org/10.1109/SANER.2015.7081848

[61] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. AutoComment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 562–567. https://doi.org/10.1109/ASE.2013.6693113

[62] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit Message Generation for Source Code Changes. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 3975–3981. https://doi.org/10.24963/ijcai.2019/552

[63] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 783–794. https://doi.org/10.1109/ICSE.2019.00086