



A large-scale empirical study of commit message generation: models, datasets and evaluation

Wei Tao¹ · Yanlin Wang² · Ensheng Shi³ · Lun Du⁴ · Shi Han⁴ · Hongyu Zhang⁵ · Dongmei Zhang⁴ · Wenqiang Zhang¹

Accepted: 29 July 2022 / Published online: 26 October 2022

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

Commit messages are natural language descriptions of code changes, which are important for program understanding and maintenance. However, writing commit messages manually is time-consuming and laborious, especially when the code is updated frequently. Various approaches utilizing generation or retrieval techniques have been proposed to automatically generate commit messages. To achieve a better understanding of how the existing approaches perform in solving this problem, this paper conducts a systematic and in-depth analysis of the state-of-the-art models and datasets. We find that: (1) Different variants of the BLEU metric used in previous works affect the evaluation. (2) Most datasets are crawled only from Java repositories while repositories in other programming languages are not sufficiently explored. (3) Dataset splitting strategies can influence the performance of existing models by a large margin. (4) For pre-trained models, fine-tuning with different multi-programming-language combinations can influence their performance. Based on these findings, we collect a large-scale, information-rich, **Multi-language Commit Message Dataset (MCMD)**. Using MCMD, we conduct extensive experiments under different experiment settings including splitting strategies and multi-programming-language combinations. Furthermore, we provide suggestions for comprehensively evaluating commit message generation models and discuss possible future research directions. We believe our work can help practitioners and researchers better evaluate and select models for automatic commit message generation. Our source code and data are available at <https://anonymous.4open.science/r/CommitMessageEmpirical>.

Keywords Commit message generation · Empirical study · Evaluation · Dataset · Multi-lingual programming languages

Communicated by: Denys Poshyvanyk

Yanlin Wang work done during the author's employment at Microsoft Research Asia.

✉ Yanlin Wang
wangylin36@mail.sysu.edu.cn

Extended author information available on the last page of the article.

1 Introduction

A tremendous amount of code is generated and updated every day, which is often maintained by version control systems such as Git. In the course of software development and maintenance, developers could frequently change their code to fix bugs, add features, perform refactoring, etc. Commits keep track of these code changes. Each commit is associated with a message which describes what and why these code changes are made. Commit messages can help developers understand and analyze code changes. For example, they can provide additional explanatory power in maintenance classification (Hindle et al. 2009) and Just-In-Time defect prediction (Barnett et al. 2016). Refactoring opportunities can be found with the analysis of commit messages (Rebai et al. 2020).

However, writing commit messages manually is time-consuming and laborious especially when the code is updated frequently (Liu et al. 2020). Generating commit messages automatically is very helpful to developers. Early work on commit message generation (Vásquez et al. 2015; Buse and Weimer 2010; Cortes-Coy et al. 2014) is based on expert rules. However, commit messages generated by rule-based methods tend to have too many lines, making it difficult to convey the key intention of the code changes (Liu et al. 2018). Later, information retrieval techniques are introduced to commit message generation. For instance, NNGen proposed by Liu et al. (2018) is a simple yet effective retrieval-based method utilizing the nearest neighbor algorithm. ChangeDoc proposed by Huang et al. (2020) is another method that retrieves the most similar commits according to the syntax and semantics in the changed code. Recently, various deep learning-based models are proposed for commit message generation. Some studies (Jiang et al. 2017; Loyola et al. 2018; Loyola et al. 2017; Jiang 2019) represent code changes as textual sequences and use Neural Machine Translation techniques to translate the source code changes into target commit messages. In addition, Liu et al. (2019) adopt the pointer-generator network (See et al. 2017) to handle the out-of-vocabulary (OOV) words. Other studies leverage the rich structural information of source code. Xu et al. (2019) jointly model the semantic representation and structural representation of code changes, Liu et al. (2020) capture both the AST structure of code changes and its semantics for commit message generation.

However, we notice that several important aspects are overlooked in existing work. 1) When evaluating commit message generation models, evaluation metrics being used vary a lot. The differences and applicability of these metrics have received little attention. 2) The evaluation datasets are different for different models and most studies experiment on only one dataset (Jiang et al. 2017; Loyola et al. 2017; Xu et al. 2019; Liu et al. 2018; Hoang et al. 2020; Liu et al. 2020). 3) The programming languages (PLs) used in different projects are rarely taken into consideration. 4) The applicable scenarios are rarely discussed, such as different data splitting strategies. 5) Recent pre-trained code models (such as CodeBERT (Feng et al. 2020)) with multiple programming languages support are rarely taken into consideration. In view of the limitations of the above-mentioned existing work, this paper would like to dive deep into the problem and answer: *How far are we in commit message generation?*

To answer the above question, we conduct a systematic analysis of commit message generation methods and their performance. We analyze the features in commit messages and compare the BLEU variants with human judgment. Moreover, we collect a large-scale commit message dataset from 500 repositories with more than 3.6M commit messages in five popular PLs. Benefiting from the comprehensive information provided by our dataset, we evaluate the performance of existing methods on different PLs. We also compare the influence of different data splitting strategies on model performance and discuss the applicable scenarios. Furthermore, we evaluate the influence of different combinations of PLs on a

CodeBERT-based (Feng et al. 2020) commit message generation model, which is fine-tuned with our multi-PL dataset.

Through extensive experimental and human evaluation, we obtain the following findings about the current commit message generation models and datasets:

- Most existing datasets are crawled from only Java repositories while repositories in other PLs are rarely taken into account. Moreover, we find some context information contributing to generation models is not available in the existing public datasets.
- By comparing three BLEU variants commonly used in existing work, we find they show inconsistent results in many cases. From the correlation coefficient between human evaluation and different BLEU variants, we find that B-Norm is a more suitable BLEU variant for this task and it is recommended to be used in future research.
- By comparing different models on existing datasets and our multi-PL dataset, we find existing models show different performance. On the positive side, most of them can be migrated to repositories in other PLs.
- Dataset splitting strategies have a significant impact on the evaluation of commit message generation models. Many studies randomly split datasets by commit. Such a splitting strategy cannot simulate the Just-In-Time situation, where the training set does not contain data later than the test set. Our results show that evaluating models on datasets split by timestamp shows much worse performance than split by commit. Therefore, in the Just-In-Time scenario, we recommend splitting by timestamp for a practical evaluation of models. Moreover, splitting data by project also leads to performance degradation, meaning that the generalization ability of existing models for new repositories is limited. Therefore, in order to evaluate models more comprehensively, it is recommended to also evaluate models on datasets split by project.
- By comparing the results of CodeBERT under different multi-PL settings, we find that in general, adding fine-tuning data with more programming languages can improve the overall performance of the model, which indicates that the models can learn from repositories in other programming languages.

Through these findings, we give actionable suggestions on comprehensively evaluating commit message generation models. Then, we discuss future work from four aspects: metrics, models, usage of multi-PL, and different scenarios. To summarize, the major contributions of this paper are as follows:

- We perform a systematic evaluation and elaborate analysis of existing work on commit message generation. We summarize our findings (including the inappropriate use of BLEU metrics, data splitting strategies, and multi-PL combinations etc), which have not received enough attention in the past.
- We develop a large dataset in multi-programming-language, which contains comprehensive information of each commit and is publicly available.¹
- We suggest ways for better evaluating commit message generation models. We also suggest possible research directions that address the limitations of existing models.

Note that this paper is an extension of our conference paper published in ICSME 2021 (Tao et al. 2021). The new contents are as follows:

- We conduct a literature review on commit message generation models.

¹<https://doi.org/10.5281/zenodo.5025758>

Table 1 Searches in databases and their results

Search string	Database	#Results
“commit message” AND (“generating” OR “generation”)	IEEE ^a	10
[All: (“commit message” AND [“generating” OR “generation”])] ^b	ACM ^c	334
(“commit message” AND (“generating” OR “generation”)) WN ALL)	I/C ^d	16
ALL (“commit message” AND (“generating” OR “generation”))	Scopus ^e	230

^a<https://ieeexplore.ieee.org>

^bRaw String: [All: “commit message”] AND [[All: “generating”] OR [All: “generation”]]

^c<https://dl.acm.org>

^dInspec/Compendex(Engineering village), <https://www.engineeringvillage.com>

^e<https://www.scopus.com>

- More recent work, especially CoRec (Wang et al. 2021b), is discussed and experimentally evaluated.
- We add experiments to evaluate the effectiveness of CodeBERT, a recent pre-trained source code model, in commit message generation with multiple programming languages.
- We explore the research questions in more depth and provide more analysis on the experimental results.

2 A Survey of Commit Message Generation Models

In this section, we conduct a systematic review of commit message generation and introduce existing models, experimental datasets, and evaluation metrics related to this task.

2.1 Search Strategy

Our search strategy follows previous guidelines (Petersen et al. 2015). We identified some search strings, which are shown in Table 1.

In total, 583 papers were obtained from these four databases: IEEE,² ACM,³ I/C,⁴ and Scopus.⁵ All of the searched results can be found in Appendix A. We went through all of these papers and found that these studies match our search strings but their relevance varies.

To filter out unqualified or irrelevant studies, we applied our inclusion and exclusion criteria following previous guidelines (Liu et al. 2022; Yang et al. 2020):

- ✓ The paper must be written in English.
- ✓ The paper must propose at least one method addressing the commit message generation and evaluate their performance.
- ✓ The length of the paper must not be less than 6 pages.

²<https://ieeexplore.ieee.org>

³<https://dl.acm.org>

⁴<https://www.engineeringvillage.com>

⁵<https://www.scopus.com>

✓ The paper must be a peer-reviewed full research paper published in a conference proceeding or a journal. Books, standards, keynote records, non-published manuscripts, and grey literature are excluded.

× Conference studies with extended journal versions are discarded.

× The studies that make use of commit messages for other software engineering tasks are ruled out.

Through reading each of the 583 papers, we find that many (433) papers do not satisfy the second criteria as they only mentioned the commit message generation in the related work but did not take commit message generation as the core topic of the paper. Another reason for the exclusion is the third criteria, which excludes 69 papers. Moreover, 39 papers are excluded because they appear repeatedly in multiple databases. 20 papers that use commit messages for other tasks are also filtered (e.g., refactoring, classification, and generating release notes). We also exclude 5 papers that are not written in English and 2 search results that are standards rather than research papers. Moreover, snowballing is also used to get more related work. We carefully read the papers that satisfy the above rules and their references. If one of the references is satisfied with the above rules, we take it into account. Finally, 15 papers are selected for the study after filtering. Inspired by the study (Sorbo et al. 2021), we put the basic information of 15 papers shown in Table 2. Note that the databases shown in Table 1 collect papers that have already been published in peer reviewed conferences or journals. We did not select preprints from Arxiv.org because they may not be peer-reviewed.

2.2 Models

We study representative commit message generation models proposed in recent years. They can be classified into four categories, namely rule-based models, generation-based models, retrieval-based models, and hybrid models.

2.2.1 Rule-Based Models

- **DeltaDoc**, proposed by Buse and Weimer (2010), is an early work to generate commit messages automatically. This algorithm can describe *What* is changed in a method. First, DeltaDoc uses symbolic execution to get path predicates for each statement in both the old and new versions of the code. Second, DeltaDoc identifies all of the added, moved, and changed statements and documents them. Third, this algorithm simplifies their outputs: “iteratively apply lossy summarization transformations until the documentation is sufficiently concise”. Finally, statements are classified in a structured form and made readable. Many change patterns in code diffs are found and utilized to construct rules for commit message generation in this paper.
- **ChangeScribe**, proposed by Cortes-Coy et al. (2014), is another rule-based method. ChangeScribe identifies the type of code diff: addition, deletion, or modification. With these types, it detects the responsibilities and stereotypes of methods using JStereoCode (Moreno and Marcus 2012). After that, it uses commit stereotypes to characterize the code diff and estimates the impact set for the code diff. Finally, ChangeScribe generates commit messages for those code diffs that are selected to reach the developer-defined “impact-value threshold”. This method takes the commit stereotype, type of code diff into account, which is beneficial to generating commit message more accurately and completely.

Table 2 An overview of the related papers

Year	Publication	Model	X-based		Metric			Code		Data
			Rule	Generation	Retrieval	BLEU	Human	ROUGE	METEOR	
2010	ASE	DeltaDoc (Buse and Weiner 2010)	■	□	□	□	■	□	□	□
2014	SCAM	ChangeScribe (Cortes-Coy et al. 2014)	■	□	□	□	■	□	■	(https://github.com/SEMERU-WM/ChangeScribe)
2016	COMPSAC	AutoSumCC (Shen et al. 2016)	■	□	□	□	■	□	□	□
2017	ASE	CritGen (Jiang et al. 2017)	□	■	□	■	■	□	■	(https://jiangl.github.io/commitgen)
2017	ACL	NMT (Loyola et al. 2017)	□	■	□	■	□	□	■	(https://oai.io67kyr/view_only=ad5886e5d114dd795553ba4951b5b99)
2018	ASE	NNGen (Liu et al. 2018)	□	□	■	■	■	□	■	(https://github.com/Thabtn/mngen)
2018	INLG	ContextNMT (Loyola et al. 2018)	□	■	□	■	■	□	■	(https://github.com/epochx/fp-commitgen)
2019	MSR	PrGNCMsg (Liu et al. 2019)	□	■	□	■	□	■	■	(https://zenodo.org/record/2593787)
2019	IJCAI	CoDSum (Xu et al. 2019)	□	■	□	■	□	■	■	(https://github.com/SoftWiser-group/CoDSum)
2020	TSE	ATOM (Liu et al. 2020)	□	■	■	■	■	■	■	(https://zenodo.org/record/407754#_YLUstBBxqjs)
2020	JCST	ChangeDoc (Huang et al. 2020)	□	□	■	■	■	□	■	□
2020	ICSE	CC2Vec (Huang et al. 2020)	□	■	□	■	□	□	■	(https://drive.google.com/file/d/1rPGw87YMNAm2_2baO6b73ynp8sa2)
2021	SANER	QAcom (Wang et al. 2021a)	□	□	■	■	■	■	■	(https://github.com/cqu-isec/QAcom)
2021	Neurecomputing	CoreGen (Nie et al. 2021)	□	■	□	■	□	■	■	(https://github.com/Flitermie/CoreGen)
2021	TOSSEM	CoRec (Wang et al. 2021b)	□	■	■	■	■	■	■	(https://zenodo.org/record/3828107)

- **AutoSumCC**, proposed by Shen et al. (2016), is another method to generate the commit message containing not only *What* but also *Why* information. *What* information is generated according to the method types which are detected by ChangeDistiller (Fluri et al. 2007). *Why* information is generated by “identifying the commit stereotypes and the type of a maintenance task”, which is classified by Swanson (1976). JStereoCode is also used in this paper to identify the type of commit as so in ChangeScribe. Moreover, the rules defined by Dragan et al. (2006) are utilized. Finally, the generated commit messages include both *What* and *Why* information by filling the template.

2.2.2 Generation-Based Models

- **CmtGen**, proposed by Jiang et al. (2017), is an early attempt to adopt Neural Machine Translation techniques in commit message generation. It treats code diffs and commit messages as inputs and outputs, respectively. CmtGen adapts one neural machine translation model Nematus (Sennrich et al. 2017) with the attentional RNN encoder-decoder (Bahdanau et al. 2015). The attention mechanism is introduced to capture long-distance features as CmtGen uses 100 as the maximum input length, which is much longer than that (50) used in Nematus (Sennrich et al. 2017).

The encoder is a bidirectional RNN that reads the source sequence x in bi-direction order. The forward RNN generates a sequence of the hidden states $\vec{h}_1, \dots, \vec{h}_T$ for each h_t at a time step t as (1) and the backward RNN generates a sequence of the hidden states $\overleftarrow{h}_1, \dots, \overleftarrow{h}_T$ as (2). x_t in the source sequence x is encoded to a concatenation as (3).

$$\vec{h}_t = f(x_t, \vec{h}_{t-1}) \quad (1)$$

$$\overleftarrow{h}_t = f(x_t, \overleftarrow{h}_{t+1}) \quad (2)$$

$$h_t = [\vec{h}_t \overleftarrow{h}_t] \quad (3)$$

The decoder defines a probability over the target sentence y by decomposing the joint probability as follows:

$$p(y) = \prod_{t=1}^T p(y_t | \{y_1, y_2, \dots, y_{t-1}\}, c) \quad (4)$$

T is the length of the input sequence, and c is a vector generated from the sequence of the hidden states h_1, \dots, h_T . In this model, code `diffs` are regarded as the input source sequence while the commit message is the output target sequence.

- **NMT**, proposed by Loyola et al. (2017), is an encoder-decoder NMT model. Different from CmtGen which uses Bahdanau attention, it introduces the attention mechanism proposed by Luong et al. (2015). It is compared as a baseline model in ATOM (Liu et al. 2020).
- **CoDiSum**, proposed by Xu et al. (2019), is another encoder-decoder based model. The major difference between CoDiSum and other models is the design of the encoder part. CoDiSum jointly models the structure and the semantics of the code diffs with a multi-layer bidirectional GRU to better learn the representations of the code changes as (5). Moreover, the copying mechanism (See et al. 2017) is used in the decoder to mitigate the out-of-vocabulary (OOV) issue.

$$h_i = \text{BiGRU}(x_i, h_{i-1}) \quad (5)$$

where h_i is the hidden state of the GRU neuron, x_i is the embedding of the i -th token in a code `diff`.

Multi-layer bidirectional GRU is also used to encode the identifier to get code semantics. Each of the identifiers is split to separate words which is denoted as $[z_{i,1}, z_{i,2}, \dots, z_{i,N}]$ where $z_{i,j}$ is the embedding of the j -th separate word in x_i , N is the maximum length of the separate words. The hidden state of the j -th separate word is as,

$$h_{i,j} = \text{BiGRU}(z_{i,j}, h_{i,j-1}) \quad (6)$$

Finally, the overall representation of each token x_i combine code structure and semantics as,

$$h'_i = [h_i \overrightarrow{h_{i,N}} \overleftarrow{h_{i,N}}] \quad (7)$$

where $\overrightarrow{h_{i,N}}$ and $\overleftarrow{h_{i,N}}$ are the hidden states from bi-direction respectively. Copying mechanism (See et al. 2017) is introduced in the decoder part.

- **ContextNMT**, proposed by Loyola et al. (2018), is a method using the source code-docstring relationship to guide the generation. ContextNMT is followed NMT but ContextNMT takes the relationship between source code and natural language (“in the form of intracode documentation”), docstrings, into account. The vectors of code and docstring are concatenated to obtain a better representation than only the vectors of code.
- **PtrGNCSmsg**, proposed by Liu et al. (2019), is an improved attentional RNN encoder-decoder model that incorporates the pointer-generator network (See et al. 2017) to translate code diffs to commit messages. It learns to either copy an OOV word from the code or generate a word from the fixed vocabulary.

The model not only calculates the probabilities p_{vocab} of each word in the vocabulary but also the probability p_{copy} of copying token from the code diff. Moreover, p_{gen} describes whether generate from vocabulary or code diff.

$$p_{vocab}(y_t|x) = f(s_t, h_i) \quad (8)$$

$$c_t = \sum_{T_i=1}^{T_x} a_{t,i} h_i \quad (9)$$

where T_x is the length pf target sentence, a_t is the attention distribution which is calculated as,

$$a_{t,i} = f(h_i, s_t) \quad (10)$$

where h_i is the hidden state and s_t is the decoder state at step t . The target vocabulary probability distribution to predict word y_t for the code diff x at time step t is calculated as,

$$p_{vocab}(y_t|x) = f(s_t, c_t) \quad (11)$$

The weight of the importance of each word x_i is represented by attention distribution $a_{t,i}$ so the probability of copying token y_t in the code diff is:

$$p_{copy}(y_t|x) = \sum_{i: x_i=y_t}^{T_x} a_{t,i} \quad (12)$$

Finally, the probability describing the importance of generating from vocabulary or code diff is used as “a soft switch between the generator mode p_{gen} or the copying mode p_{copy} ” which is calculated as,

$$p_{gen} = f(c_t, s_t, x_t) \quad (13)$$

where x_t represents the decoder input. The overall probability of each word is calculated as,

$$p(y_t|x) = p_{gen} p_{vocab}(y_t|x) + (1 - p_{gen}) p_{copy}(y_t|x) \quad (14)$$

- **CoreGen**, proposed by Nie et al. (2021), is another method to generate commit messages. CoreGen uses two tasks to get a more contextualized code representation: code change prediction and masked code fragment prediction. The Transformer is fed with the old version of code to predict the new version of code. The objective function of the first task is,

$$\mathcal{L}_1 = - \sum_{X \in \mathbb{C}_1} \sum_i \log \mathcal{P}(x_i^{new} | X^{old}; \theta) \quad (15)$$

where x_i^{new} represents the i -th code token in the new version of code sequence X , \mathbb{C}_1 represents code commit subcorpora with explicit code diffs, and θ refers to the parameters of Transformer model. The objective function of the other task is,

$$\mathcal{L}_2 = - \sum_{X \in \mathbb{C}_2} \sum_{i=u}^v \log \mathcal{P}(x_i^k | X^1, \dots, X_{u:v}^k, \dots, X^n; \theta) \quad (16)$$

where x_i^k represents the i -th code token in the masked line X^k , and \mathbb{C}_2 represents code commit subcorpora with implicit code diffs. The Transformer is trained to predict the masked code fragment $x_{u:v}^k$ from its context $\{x_i^k | X^1, \dots, X_{u:v}^k, \dots, X^n\}$. Finally, the overall objective function is,

$$\mathcal{L}_{\text{overall}}(\theta; \mathbb{C}) = \frac{1}{|\mathbb{C}|} (\mathcal{L}_1 + \mathcal{L}_2) \quad (17)$$

where \mathbb{C} represents the combination of \mathbb{C}_1 and \mathbb{C}_2 .

2.2.3 Retrieval-based Models

- **NNGen**, proposed by Liu et al. (2018), leverages the nearest neighbor (NN) algorithm to generate commit messages. Code diffs are represented as vectors in the form of “bag of words” (Mogotsi et al. 2010). To generate a commit message, NNGen calculates the cosine similarity between the target code diff and each code diff in the training set. Then, the top-k code diffs in the training set are selected to compute the BLEU scores between each of them and the target code diff. The one with the largest BLEU score is regarded as the most similar code diff and its commit message will be used as the target commit message. The use of cosine similarity for retrieval can boost efficiency and the use of BLEU for ranking can improve the performance. Therefore, NNGen strikes a balance between effectiveness and efficiency.
- **CC2Vec**, proposed by Hoang et al. (2020), is a neural model that learns a representation of code changes guided by the accompanying commit messages. It aims to represent the semantic intent of the code changes by modeling the hierarchical structure of a code change with an attention mechanism and using various comparison functions to identify the differences between the deleted and added code. The learned vector representations of diffs are used to adapt the NNGen model to retrieve a code diff that is most similar to the input so that the corresponding commit message can be used as the output.
- **ChangeDoc**, proposed by Huang et al. (2020), is a method reusing existing messages in commit history to generate. ChangeDoc retrieves the most similar commits according to the syntax and semantics in the changed code fragments from commits.
- **QAcom**, proposed by Wang et al. (2021a), is a component to filter low-quality commit messages. It makes full use of historical commits to calculate the quality scores. First, it builds a correlation between each word in code changes and each word in commit

messages. This correlation helps QAcom evaluate whether the generated result is under-translation or over-translation. *Under-translation* means code changes are not fully utilized in the commit message and this aspect is estimated by the “precision”. *Over-translation* means that the generated commit message has some unnecessary words and this is estimated by the “recall”. Second, another retrieval-based module calculates the quality score based on the similarities between the code change and the code changes of most similar historical commits. These three scores are combined together to determine whether the generated result is semantically relevant.

2.2.4 Hybrid Model

- **ATOM**, proposed by Liu et al. (2020), is a hybrid model that combines the techniques in generation-based models and retrieval-based models. ATOM is the first model that makes use of the Abstract Syntax Trees (ASTs) of the code diffs for commit message generation. In the generation module, AST paths extracted from ASTs are encoded with a BiLSTM model to represent the code diffs and then the attention mechanism is used in the decoder to generate a sequence as the commit message. In the retrieval module, the code diff that has the largest cosine similarity with the input code diff is retrieved from the training set. Finally, a hybrid ranking module is used to prioritize the commit messages obtained from the generation and retrieval modules.
- **CoRec**, proposed by Wang et al. (2021b), is another hybrid model combining the techniques in generation-based models and retrieval-based models. CoRec is designed to address two issues: generation-based models often generate high-frequency words but ignore low-frequency; exposure bias (Ranzato et al. 2016) which means that the model is not trained for the situation when a previously generated word is wrong. The information retrieval module is used to address the first issue and the decay sampling mechanism makes the model fully exposed and then mitigates the second issue.

2.2.5 Pre-trained Model

Pre-trained models such as CodeBERT can also be used in the task of commit message generation.

- **CodeBERT**, proposed by Feng et al. (2020), is a large bimodal pre-trained model for natural language and programming language. CodeBERT adopted the same model architecture as RoBERTa-base (Liu et al. 2019). It has two objects of training: masked language modeling (Devlin et al. 2019; Liu et al. 2019) and replaced token detection (Clark et al. 2020). It shows good performance after fine-tuning on downstream tasks such as code-to-documentation generation in multiple programming languages. However, commit message generation is not investigated in that paper.

2.3 Datasets

There are several existing datasets for the commit message generation task. Table 3 reports their basic information.

- **CmtGen_{data}**: It is an early commit message generation dataset used in CmtGen and other studies (Liu et al. 2018, 2019; Hoang et al. 2020). It is pre-processed from the commit dataset provided by Jiang and McMillan (2017) which is collected from top-1000 Java GitHub projects. CmtGen_{data} extracts the first sentence from the original

Table 3 Statistics of Datasets

Dataset	Lang.	# Commits	Content
CmtGen_{data}	Java	32,208	⟨Diff, Message⟩
NNGen_{data}	Java	27,144	⟨Diff, Message⟩
CoDiSum_{data}	Java	90,661	⟨Diff, Message⟩
PtrGen_{data}	Java	32,663	⟨Diff, Message⟩
MultiLang_{data}	Java	38,734	⟨Diff, Message, RepoName⟩
	C++	43,868	
	Python	41,023	
	JavaScript	29,821	
ATOM_{data}	Java	197,968	⟨Diff, Message, RepoName, SHA, Timestamp⟩
CoRec_{data}	Java	107,448	⟨Diff, Message⟩
MCMD (Ours)	Java	450,000	⟨Diff, Message, RepoFullName, SHA, Timestamp⟩
	C#	450,000	
	C++	450,000	
	Python	450,000	
	JavaScript	450,000	

Datasets evaluated are marked in bold

commit messages and removes the commits which have ids of issues. Merge and roll-back commits are also removed because existing models are not suitable for most of these commits with too many lines. A Verb-Direct Object filter is also introduced to filter out non-compliant commit messages. After filtering, 32K commits remain in the dataset. The training, validation and test sets contain 26,208, 3,000, and 3,000 commits, respectively.

- **NNGen_{data}**: Liu et al. (2018) find that CmtGen_{data} contains about 16% noisy messages that can be divided into two categories: *bot messages* (generated by bot) and *trivial messages* (written by human but contain little information about code diff). Liu et al. (2018) remove these noises and proposed the cleaned subset of CmtGen_{data}, i.e., NNGen_{data}. The training, validation and test sets contain 22,112 / 2,511 / 2,521 commits, respectively.
- **CoDiSum_{data}**: Based on the dataset in Jiang and McMillan (2017), Xu et al. (2019) remove the commits that contain no source code changes. They also remove punctuations and special symbols in commit messages, and filter commit messages that contain less than three words and duplications. Finally, they obtain 90,661 pairs of ⟨Diff, Message⟩, and they randomly choose 75,000 / 8,000 / 7,661 for training, validation, and testing.
- **PtrGen_{data}**: Liu et al. (2019) collect the top 1,001–2,081 Java projects. They remove the rollback and merged commits, extract the first sentence from messages, and replace the diff signs (+ and −) with special tokens <add> and <delete>. The training, validation and test sets contain 23,623 / 5,051 / 3,989 commits, respectively.
- **MultiLang_{data}**: Loyola et al. (2017) collect pairs of ⟨Diff, Message⟩ from 12 public projects in 4 PLs: Python, JavaScript, C++, and Java, with 12,787 pairs of ⟨Diff,

Message) in each project on average. They randomly choose 80% for training, 20% for validation and testing.

- **ATOM_{data}**: Liu et al. (2020) collect data from 56 Java projects with the largest number of stars. After filtering commits with noisy messages and commits that contain no source code changes, ATOM_{data} contains 197,968 commits. This dataset is designed to provide not only the raw commits but also the extracted functions which are affected in each commit. They randomly choose 81% for training, 10% for validation, and 9% for testing.
- **CoRec_{data}**: Wang et al. (2021b) collect data from 10,000 Java projects with the largest number of stars. It is preprocessed in the same way as CmtGen_{data}. Moreover, CoRec_{data} filters the commits that contain keywords such as “changelog”, “version”, or match patterns such as “modify dockerfile/makefile”. This filtering strategy extends the pattern proposed by Liu et al. (2019). Duplicated commits are also removed. Finally, 107.4K commits are preserved. They randomly choose 90% for training, 5% for validation, and 5% for testing.

2.4 Evaluation Metrics

Several metrics commonly used in NLP tasks such as machine translation, text summarization, and captioning can be adopted for evaluating commit message generation. These metrics include BLEU (Papineni et al. 2002), METEOR (Banerjee and Lavie 2005), ROUGE (Lin 2004), etc. In this study, we focus on BLEU as it is the metric that most of the related works (Jiang et al. 2017; Loyola et al. 2017, 2018; Liu et al. 2018, 2019, 2020; Xu et al. 2019; Huang et al. 2020; Hoang et al. 2020; Wang et al. 2021a, b; Nie et al. 2021) (12/15) use to evaluate the performance of commit message generation models.

BLEU score is used to evaluate the correlation between the generated and the reference sentences in NLP tasks. For the commit message generation, the references are the commit messages written by developers and the generated sentences are the outputs from the models. Different BLEU variants are used in prior work and they could produce different scores for the same generated commit message. In the following, we are going to illustrate three different variants of BLEU used before. The names are not intended to be a standard in the literature, but just for easy reference in this study.

- **B-Moses**: The evaluation described in previous studies (Jiang et al. 2017, Liu et al. 2018, 2019) use the same BLEU script from the open-sourced code in Koehn et al. (2007) which calculates B-Moses. B-Moses is designed for statistical translation, which does not use a smoothing function. It can be calculated as follows:

$$\text{BLEU} = \text{BP} \cdot \exp \left(\sum_{n=1}^N w_n \log p_n \right) \quad (18)$$

where w_n is the weight of n -gram precision p_n , which can be obtained as (20). If not explicitly specified, $N = 4$ and uniform weights $w_n = 1/4$. BP is brevity penalty which is computed as:

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \quad (19)$$

where c is the length of the candidate generation and r is the length of the reference. The n -gram precision p_n can be obtained as,

$$p_n = m_n / l_n \quad (20)$$

where m_n is the number of matched n -grams between the reference and the generation, and l_n is the total number of n -grams in the generation.

- **B-Norm:** B-Norm is a BLEU variant adapted from B-Moses. It is used by Loyola et al. (2017). One difference between B-Norm and B-Moses is that B-Norm converts all characters both in the reference and the generation to lowercase before calculating scores. Therefore, B-Norm is case insensitive. The smoothing method proposed by Lin and Och (2004) is used in B-Norm to smooth the calculation of n -gram precision scores. It adds a constant number (one) to both the numerator and denominator of p_n for $n > 1$.
- **B-CC:** Hoang et al. (2020) use BLEU measure provided by NLTK (Xue 2011) with the smoothing method proposed by Chen and Cherry (2014) in their evaluation. This smoothing method (Chen and Cherry 2014) is inspired by the assumption that matched counts for similar values of n should be similar. The average value of the $n - 1$, n and $n + 1$ -gram matched counts is used as n -gram matched count. m'_0 is defined as $m_1 + 1$. Therefore, m'_n for $n > 0$ is defined as,

$$m'_n = \frac{m'_{n-1} + m_n + m_{n+1}}{3} \quad (21)$$

3 Study Design

3.1 Experimental Models

In this study, we select the models to be evaluated according to the following criteria: a) source code is publicly available, and b) we can confirm the correctness of source code by checking the implementation provided by authors and reproducing results presented in the original paper. CmtGen,⁶ CoDiSum,⁷ NMT,⁸ PtrGNCSmsg,⁹ NNGen,¹⁰ CoRec,¹¹ and CodeBERT¹² satisfy these criteria and thus are selected for in-depth evaluation in our study. For models that hyper-parameter settings are reported in their papers, we use the same hyper-parameters. Otherwise, we tune the hyper-parameters empirically to optimize each model.

For CC2Vec, after inspecting its public source code,¹³ we suspect that the implementation is inconsistent with descriptions in the paper. We find that the scores reported in CC2Vec paper are produced by the code that retrieves the commit message of the most similar code diff in terms of *BLEU score* instead of the vector representation described in the CC2Vec paper. We have tried to modify the code according to the paper, but the result drops significantly in BLEU. We have contacted the authors regarding this issue. Therefore, CC2Vec is not evaluated in this study.

For ATOM, extracting AST paths for code diff is a key step during the pre-processing. However, the tool for this step is not available. We have contacted the authors but it cannot

⁶<https://sjiang1.github.io/commitgen>

⁷<https://github.com/SoftWiser-group/CoDiSum>

⁸<https://github.com/epochx/commitgen>

⁹<https://zenodo.org/record/2542706>

¹⁰<https://github.com/Tbalm/nngen>

¹¹<https://zenodo.org/record/3828107>

¹²<https://github.com/microsoft/CodeBERT>

¹³<https://github.com/CC2Vec/CC2Vec>

be provided for commercial reasons. We have tried to replace the required path extraction tool with JavaExtractor¹⁴ (Alon et al. 2019) for extracting AST paths from the Java function. However, this attempt cannot fully match the results in ATOM paper, therefore, in most experiments except for the one reported in Table 6, ATOM is not evaluated.

For QAcorn, it filters low-quality commit messages. After filtering, fewer commit messages are generated than other models. Therefore, it is not fair to compare QAcorn with other models and we do not evaluate it.

3.2 Experimental Datasets

3.2.1 Existing Datasets

We select datasets from the existing ones based on their availability and representativeness. In this way, three existing datasets are chosen as highlighted in bold in Table 3: CmtGen_{data}, NNGen_{data}, and CoDiSum_{data}. The reasons are explained as follows.

CmtGen_{data} and NNGen_{data} are Java datasets and are commonly used in previous studies (Jiang and McMillan 2017; Liu et al. 2018, 2019; Hoang et al. 2020). NNGen_{data} is more difficult than CmtGen_{data} because commit messages with certain patterns are filtered in NNGen_{data}. CoDiSum_{data} is another Java dataset with different features compared to CmtGen_{data} and NNGen_{data}. It is a deduplicated dataset, i.e., its training set and test set are not overlapping. CoDiSum_{data} can be used to evaluate the generalization ability of models.

PtrGen_{data} is not used in our study since they are very similar to CmtGen_{data} except that CmtGen_{data} is collected from top-1,000 starred GitHub projects while PtrGen_{data} is from top-1001 to top-2081 projects. The performance difference is reported to be very small on CmtGen_{data} and PtrGen_{data} (Liu et al. 2019). Besides, CmtGen_{data} is used more often in the literature (Jian et al. 2017; Liu et al. 2018, 2019; Hoang et al. 2020) than PtrGen_{data} (Liu et al. 2019).

CoRec_{data} is not used in our study since they are very similar to NNGen except that NNGen is filtered from top-1,000 starred Java GitHub projects while CoRec_{data} is filtered from top-10,000 Java projects.

MultiLang_{data} is not used for two reasons. First, in MultiLang_{data}, commits for each programming language are collected from only three repositories, resulting in small and sparse data. Second, the three collected repositories are not available from the provided link,¹⁵ making it difficult to inspect the data source.

ATOM_{data} is not chosen because the given dataset is incomplete. For example, all commits to the repository retrofit are missing. Recovering the data is not feasible because both the repositories' full names (including owner and repositories' names) and version numbers are not provided.

3.2.2 Our Dataset MCMD

Existing datasets have facilitated the development of commit message generation. However, the available datasets have their limitations: most are in a single programming language (i.e., Java), and the available information is very limited. There is only one dataset MultiLang_{data} (Loyola et al. 2017) in multiple PLs, but it is not usable as explained in Section 3.2.1.

¹⁴<https://github.com/tech-srl/code2seq/tree/master/JavaExtractor>

¹⁵https://osf.io/67kyc/?view_only=ad588fe5d1a14dd795553fb4951b5bf9

To provide a large-scale dataset in multiple PLs and with rich information, we created a new dataset **MCMD**, short for **M**ulti-**p**rogramming-**l**anguage **C**ommit **M**essage **D**ataset. To make this dataset more representative, programming languages are selected based on their popularity. We select the top 5 programming languages which are the most popular according to the PYPL Popularity index:¹⁶ Python, Java, JavaScript, C#, C++. For each language, we collected commits before 2021 from the top 100 starred projects on GitHub. In this step, a total of 3.69M commits were collected. We removed branch merging and rollback commits, and filtered out noisy messages as Liu et al. (2018), to improve the quality of commits in our dataset. About 3.42M commits remain after filtering. To balance the size of data in each programming language so that we can fairly compare the performance of models in the different programming languages in subsequent experiments, we randomly sampled and retained 450,000 commits for each language.

Existing datasets (Jiang et al. 2017; Liu et al. 2018; Xu et al. 2019) contain the information of only code diffs and the commit messages. However, the context of the code diffs can contribute to explaining why this code is added and what role it plays in the software (Sillito et al. 2008). For example, extracting AST paths from the code diffs is beneficial to the commit message generation model (Liu et al. 2020), which requires the dataset to provide enough information to find the complete affected functions around the code changes. However, most of the existing datasets (Jiang et al. 2017; Liu et al. 2018; Xu et al. 2019; Loyola et al. 2017) do not provide information for retrieving related functions. To trace back to the original repository, the *RepoFullname* (including owner and repository's name) and *SHA* of a repository should be recorded. The *RepoFullname* can be used to find the corresponding repository and *SHA* is a unique ID to identify the version of the repository. With sufficient data in our dataset, all of the commits can be traced by accessing the URL: <https://github.com/RepoFullname/commit/SHA> such as <https://github.com/libgdx/libgdx/commit/e13d9466776c37486ca914ff503eabc35136c6ea>, which allows future tools to aggregate additional features.

Moreover, if we want to split the dataset by timestamp, timestamps of commits are necessary. Considering the above demands, our dataset MCMD contains the complete information of commits, including not only code diffs and commit messages, but also *RepoFullname*, *SHA*, and timestamp. We have made MCMD public¹⁷ to benefit future research on commit message generation.

3.3 Research Questions

We have identified the following Research Questions (RQs) and will seek their answers in our evaluation:

RQ1: How do different BLEU variants affect the evaluation of commit message generation?

As described in Section 2.4, most existing works use BLEU as an evaluation metric but the scores in their papers are different BLEU variants. Scores for different BLEU variants can vary a lot for the same sentence as shown in Table 4.

For instance, as Table 4 and Fig. 1 illustrate, the commit message “[fixed] Mesh . setVertices () .” generated by PtrGNCMsg is relatively reasonable for that

¹⁶<https://pypl.github.io/PYPL.html>

¹⁷<https://doi.org/10.5281/zenodo.5025758>

Table 4 Example scores under different metrics

Reference (Ref), Generation(Gen)	BM	BN	BC
Ref: add setup ()	0.00	100.00	22.80
Gen: add setUp ()			
Ref: Fix merge conflicts	0.00	100.00	25.00
Gen: fix merge conflicts			
Ref: BAEL - 3001	0.00	19.64	12.54
Gen: BAEL - 2412: Add a new class			
Ref: Fix typo	0.00	19.64	12.54
Gen: Fix typo in core - validation . adoc			
Ref: Update visualvm to build 908	0.00	19.64	12.54
Gen: [GR - 6405] Update visualvm .			
Ref: [FIXED JENKINS - 12514]	22.31	36.41	41.26
Gen: [FIXED JENKINS - 12514] Fixed a bug in bundled plugins on Windows.			
Ref: [GR - 22084] Add TruffleCreateGraphTime timer .	0.00	19.68	11.51
Gen: Add a timer to the timer .			
Ref: Remove dead code .	0.00	19.07	13.25
Gen: [GR - 19154] Remove unused code .			
Ref: Fix reported leaks	0.00	24.03	8.98
Gen: Fix a bug in SnappyFramedEncoderTest			
Ref: [fixed] Bug in Mesh . setIndices () . had to clear buffer first .	0.00	18.97	16.63
Gen: [fixed] Mesh . setVertices () .			

BM, BN, BC are short for B-Moses, B-Norm, and B-CC, respectively

File Path	gdx/src/com/badlogic/gdx/graphics/Mesh.java
-393,6 +393,7	public void setVertices (int[] vertices, int offset, int count) {
393 393	* @param indices the indices
394 394	*/
395 395	public void setIndices(short[] indices) {
396	this.indices.clear();
396 397	this.indices.put(indices);
397 398	this.indices.limit(indices.length);
398 399	this.indices.position(0);
Reference Commit Message:	
[fixed] Bug in Mesh . setIndices () . had to clear buffer first .	
Generated by PtrGNCSmsg:	
[fixed] Mesh . setVertices () .	

Fig. 1 An example of the code diff and the corresponding commit messages

code changes. Compared with the reference, it has shared tokens and the meaning is partially correct. However, it has different scores for different BLEU variants, as shown in Table 4. The B-Moses score of 0 means that this generation and reference are completely different, which is not true. Another case is the commit message “fix merge conflicts” generated by NMT, which has the same meaning with the reference but it has lower scores for B-Moses (0) and B-CC (25.00) while it has the perfect score for B-Norm (100). These examples are just the “tip of the iceberg”.

RQ1 chooses the most suitable BLEU metric for the task of commit message generation by human evaluation and analyzes why that variant is better than others. Following best practices for human evaluation (van der Lee et al. 2019), three human experts manually labeled the data. All of them have more than 5 years of programming experience and they are majoring in Computer Science. Firstly, we select 100 commit messages randomly from generation results which show large disagreement (the variance among the three BLEU metrics is larger than 30).

Then, we define criteria in three aspects for manual labeling as shown in Table 5 following previous works (Moreno et al. 2013; Panichella et al. 2016) that use different perspectives of human judgment: content adequacy, conciseness, and expressiveness. These

Table 5 The meaning of scores in human evaluation on three metrics

Score	Meaning
Content adequacy	
Is the important information about the code diff reflected in the commit message?	
0	Missing all of the information about the code diff.
1	Is missing some very important information that can hinder the understanding of the code diff.
2	Is missing some information but some of the missing information is not necessary to understand the code diff.
3	Is missing some information but all of the missing information is not necessary to understand the code diff.
4	Is not missing any information.
Conciseness	
Is there extraneous information included in the commit message?	
0	All of the information is unnecessary.
1	Has a lot of unnecessary information.
2	Has some unnecessary information.
3	Has a little unnecessary information.
4	Has no unnecessary information.
Expressiveness	
How readable and understandable is the commit message?	
0	Cannot read and understand.
1	Is hard to read and understand.
2	Is somewhat readable and understandable.
3	Is mostly readable and understandable.
4	Is easy to read and understand.

metrics can help us to demonstrate the correlation between traditional human evaluation and different BLEUs in three views.

The raters give a score between 0 to 4 to measure the semantic similarities between reference and the generated commit message. After labeling, all scores are double-checked by each rater to confirm whether scores from humans are stable and reliable.

To validate the reliability of human scores, we calculate Krippendorff's alpha (Hayes and Krippendorff 2007) and Kendall rank correlation coefficient (Kendall's Tau) (Kendall 1945) values.¹⁸ Before the calculation, these human direct assessments are converted into relative rankings as Direct Assessment Relative Rankings (DaRR) serve as the golden standard for segment-level evaluation (Ma et al. 2019). On the evaluation of content adequacy and conciseness, all of the Krippendorff's alphas are greater than 0.72, and all of the Kendall's Tau values between any two raters are greater than 0.65, which indicates that there is a relatively high degree of agreement between the raters and these human labels are reliable.

Human scores are regarded as the reference and we want to compare the three BLEU variants by their correlations with the reference. Spearman (Myers et al. 2013) and Kendall (Kendall 1945) are selected because the scores are ordinal and satisfy their assumptions. Note that these correlation coefficients are calculated per commit and more details can be found in our repository.

RQ2: How good are the existing models and datasets?

As described in Section 2.3, When evaluating commit message generation models, not only are the evaluation metrics different, but the datasets used are also different. Most of the existing studies (Loyola et al. 2017; Xu et al. 2019; Liu et al. 2020) only experiment on one dataset. Therefore, in RQ2, we conduct a unified evaluation of existing models on existing datasets and study the impact of using different datasets on model evaluation.

Moreover, we evaluate each model's stability and efficiency in this RQ. Specifically, stability refers to whether the test results under different lengths of commit messages are stable. Efficiency refers to the model inference time.

RQ3: Why do we need a new dataset MCMD for evaluating commit message generation?

Since most of the prior works focus on Java datasets, there is a lack of research on other PLs' repositories that have the same need to generate commit messages automatically. In RQ3, we explore the necessity of a new dataset. And rely on the new dataset MCMD, we collected to explore the performance of migrating existing models to other PLs. Similar to previous work (Jiang et al. 2017; Liu et al. 2019, 2020), for the dataset of each language in MCMD, we randomly select 80% data for training, 10% data for validation, and 10% data for testing.

RQ4: What is the impact of different dataset splitting strategies?

The commit message generation models have different usage scenarios, so they need to be evaluated on datasets split by different strategies that simulate different scenarios. For example, the "split by timestamp" strategy simulates the just-in-time (JIT) scenario where the model cannot see future data and can only use past data for training. However, for the datasets used in most previous work, commits are split into training, validation,

¹⁸The details of calculation can be seen at our repository <https://anonymous.4open.science/r/CommitMessageEmpirical>

and testing sets randomly (split by commit), while other situations such as split by project (where training, validation, and testing sets contain commits from disjoint projects) are not considered. In RQ4, we study the impact of different dataset splitting strategies from two aspects.

Split by Timestamp As a previous study on code summarization (LeClair and McMillan 2019) suggests: “Care must be taken to avoid unrealistic scenarios, such as ensuring that the training set consists only of code older than the code in the test set”. The commit message generation task is similar to the code summarization task in this regard, especially in a JIT scenario, the model cannot see future data and can only use past data for training. Therefore, we further conduct experiments on datasets split by timestamp instead of by commit to ensure future commits are not used as training data. In this splitting strategy, we divide each programming language’s dataset for training (80%), validation (10%), and test (10%) in chronological order.

Split by Project According to the study of LeClair and McMillan (2019) on code summarization, splitting dataset by function (in analogy with “commit” in our study) might cause information leakage from test set projects into the training or validation sets and should be avoided. Following previous studies (LeClair and McMillan 2019; Liu et al. 2020), we also evaluate the performance of models based on MCMD split by project to ensure that projects in the training, validation and test sets are disjoint. In this splitting strategy, we divide each PL’s dataset for training (80%), validation (10%), and test (10%) by the repository.

The experiments of splitting by project can reflect the performance of models on new repositories. For a repository that has commits before, the models can be trained by using only other repositories, using the repository itself, or using both of itself and others. Furthermore, to mimic the scenario that we are predicting commit messages on a new project with a trained model, we conduct a series of experiments called *single project experiments*. As illustrated in Fig. 2, the test set for all experiments is the same, and it comes from the target project. The training sets for the three experiments in the single project experiments are: ① data only from the target project for *Within-Project* setting; ② data only from other projects in the same programming language for *Cross-Project* setting; and ③ the union of

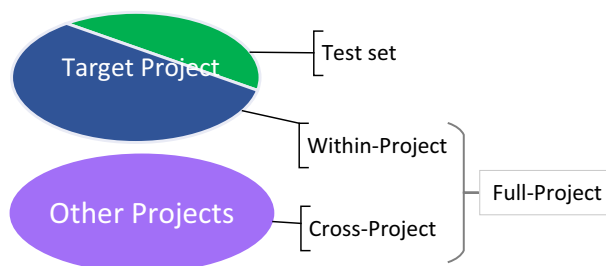


Fig. 2 Illustration of data settings for single project experiments in Table 13. The training data for Within-Project, Cross-Project, and Full-Project settings are blue, purple, and blue+purple parts, respectively. Test data (the green part) is the same for all three settings, taken from the target project

training sets from the target project and other projects for *Full-Project* setting. The experiment on these three settings can provide suggestions for the models' usability on existing repositories.

RQ5: What is the effectiveness of the pre-trained model in commit message generation?

In this RQ, we evaluate the effectiveness of the pre-trained model, especially CodeBERT (Feng et al. 2020), in commit message generation. CodeBERT shows good performance on downstream tasks such as code-to-documentation generation for multiple programming languages. However, commit message generation is not investigated in the original CodeBERT study. Benefiting from our multi-PL dataset, we select the data in different combinations of PLs (such as Java and Python repositories) as the training set to fine-tune CodeBERT and then test the fine-tuned model on the data in different PLs (such as Java and C++ repositories). Following the fine-tuning steps described in the code-to-documentation generation task of CodeBERT, we treat code diff as code and commit messages as documentation.

4 Results and Findings

4.1 How Do Different BLEU Variants Affect the Evaluation of Commit Message Generation? (RQ1)

To answer this question, we evaluate commit message generation models using different BLEU variants and compare BLEU scores with the results from a human evaluation.

Table 6 Models performance

Model		CmtGen	CoDiSum	NMT	PtrGNCSmsg	NNGen	CoRec	ATOM
Data	Metric							
CmtGen _{data}	BM	34.03	0.00	32.09	35.41	38.54	41.14	/
	BN	31.11	6.88	26.66	29.86	34.74	36.93	/
	BC	26.00	0.49	21.51	24.82	29.44	30.74	/
NNGen _{data}	BM	15.08	0.00	7.46	9.69	16.41	19.44	/
	BN	21.60	8.03	13.82	18.96	23.07	25.38	/
	BC	15.48	0.86	8.28	10.39	16.77	18.29	/
CoDiSum _{data}	BM	1.33	1.74	1.32	0.81	3.04	3.00	/
	BN	9.37	15.45	9.93	12.71	9.07	10.62	/
	BC	4.17	5.72	3.81	4.77	5.27	5.15	/
MCMJava	BM	6.29	2.00	9.17	8.25	13.30	11.39	7.47
	BN	12.39	14.00	13.39	15.33	17.81	16.09	16.42
	BC	7.67	5.37	10.24	11.70	14.46	12.68	9.29

BM, BN, BC are short for B-Moses, B-Norm, and B-CC, respectively

Table 7 Correlation between metrics and human evaluations on content adequacy and conciseness

Metric	B-Moses	B-Norm	B-CC
Content adequacy			
Spearman	0.2193*	0.6462*	0.5612*
Kendall	0.1896*	0.4878*	0.4137*
Conciseness			
Spearman	0.1303	0.6830*	0.4277*
Kendall	0.1117	0.5236*	0.3045*
Expressiveness			
Spearman	0.0269	0.4136*	0.2327*
Kendall	0.0231	0.3224*	0.1739*

*The correlation between them is significant (p-value < 0.05)

The largest correlation coefficient under each metric is marked in bold

4.1.1 Experiments Under Different BLEU Variants

As shown in Table 6, rankings of models are inconsistent under different metrics.¹⁹ For example, on CoDiSum_{data}, CoDiSum is the best model when measured by B-Norm or B-CC, while NNGen is the best model when measured by B-Moses. On CmtGen_{data}, if the B-Moses metric is used, PtrGNCMsg is better than CmtGen, but an opposite conclusion will be drawn if B-Norm is used. Similar inconsistencies can also be observed in Table 10.

4.1.2 Human Evaluation

Table 7 shows the correlation scores between three BLEU variants and human evaluation, under two correlation metrics: Spearman (Myers et al. 2013) and Kendall (1945). We can see that B-Norm is most correlated with human judgment, especially on content accuracy and conciseness. After manually investigating a great number of commit messages in the test set and comparing the design of three BLEU variants, we find two possible reasons: smoothing and case sensitivity.

B-Moses does not perform smoothing when calculating each p_n while B-Norm and B-CC do so. However, more than 17.19% commit messages have less than five tokens in MCMD. Therefore, 4-gram precision p_4 (shown in (20)) of these commit messages is close to zero, leading the geometric mean of n-gram precision scores to be zero even if there are many 1-gram, 2-gram, or 3-gram matches. Without smoothing, a short commit message that is identical to the reference will get a near-zero B-Moses score, which is unreasonable. As many commit messages are short, we believe that B-Moses is not very suitable for evaluating commit message generation.

B-Norm is not case sensitive while B-Moses and B-CC are. As shown in Table 4, the generated message “fix merge conflicts” has the same meaning as the reference “Fix merge conflicts”. The only difference is the case of “Fix” and “fix”. Besides, other words such as “add” and “Add” also have the same meaning. Scores of B-Moses and B-CC for commit messages that differ only in case tend to be lower. The lower scores are unreasonable since these messages have exactly the same meaning as the references.

¹⁹ All the generated commit messages are available in our repository.

Summary: For the evaluation of commit message generation models, using different metrics may lead to different conclusions. B-Norm, which uses a smoothing method and is case insensitive, is more in line with human judgments.

4.2 How Good are the Existing Models and Datasets? (RQ2)

Based on the experimental results shown in Table 6, we have the following findings:

- The scores of the same model on different datasets can vary a lot. For example, the B-Norm score of the NNGen model on CmtGen_{data} is 34.74. When evaluated on CoDiSum_{data}, the score drops to 9.07. Hence, we should consider more datasets in the evaluation: good performance on one dataset does not mean we can observe similar performance on another dataset.
- The scores on CmtGen_{data} are higher than scores on NNGen_{data}. NNGen_{data} is a subset of CmtGen_{data}, in which noisy messages are filtered out as described in Section 2.3. To investigate the role of noise data in model training and evaluation, we conduct ablation experiments on CmtGen and the results are shown in Table 8. We split the test set of CmtGen_{data} into 2 parts: one only contains noisy messages, and the other is the rest (i.e., the test set of NNGen_{data}). We can see that the scores on the test set of noisy messages are much higher than that of NNGen_{data}, indicating that noisy messages are easy to generate. However, these messages (e.g., branch merging messages) are often bot-generated and do not need to be predicted by a model. Therefore, what really needs to be compared is the performance on the NNGen_{data} test set. We further investigated whether excluding noisy data in model training will improve its performance. As shown in Table 8, training on NNGen_{data} (noise-free data) has a higher score than CmtGen_{data}, which indicates that it is better not to include noisy data in model training.
- When experimenting on CmtGen_{data} and NNGen_{data}, NNGen has the highest scores under all metrics. But NNGen does not perform the best on CoDiSum_{data}. We speculate that this is because NNGen is retrieval-based and it is easy for it to achieve high scores on datasets with duplicated data. After checking, we find that in the test set of NNGen_{data}, there are 16.02% duplicated commit messages and 5.16% duplicated (Diff, Message) pairs from the training set. And in the test set of CmtGen_{data}, there are 29.13% duplicated commit messages and 4.67% of duplicated (Diff, Message) pairs. In contrast, CoDiSum_{data} is a deduplicated dataset as described in Section 3.2.1. As a retrieval-based model, NNGen obtains a high score by leveraging the duplication in the dataset.

Based on the experimental results in Fig. 3, we find that most generation-based models show better performance when the commit message is shorter than 20. Moreover, when

Table 8 Noisy data ablation study

	Testing Training	NNGen _{data}			Noisy data		
		BM	BN	BC	BM	BN	BC
BM, BN, BC are short for B-Moses, B-Norm, and B-CC. Model evaluated is CmtGen	CmtGen _{data}	11.38	19.09	12.86	97.44	94.43	95.00
	NNGen _{data}	15.08	21.60	15.48	/	/	/

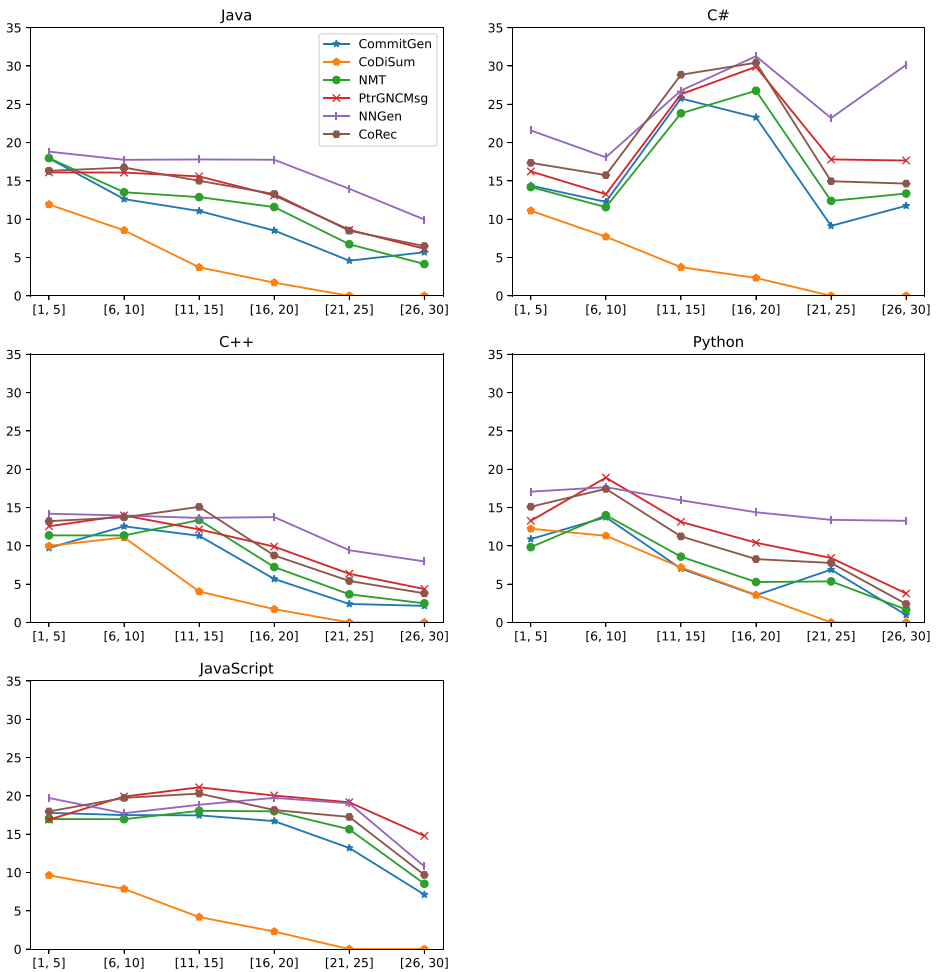


Fig. 3 The performance of commit messages of different length intervals in the test set (5 PLs of MCMD). The horizontal axis represents the length range of the commit messages, and the vertical axis represents the B-Norm scores

the commit message is longer than 20, the retrieval-based model, NNGen, shows better performance in general.

This finding reveals the advantage of retrieval-based models on the longer length of commit messages.

During our experiments, we find that most of the models need to be trained for more than 10 hours on two Tesla V100 GPUs. Moreover, as shown in Table 9, the inference time of these models vary from 3 minutes to 17 hours. In general, the retrieval-based models require more inference time than the generation-based models. If the dataset is smaller, this gap would be smaller.

Table 9 The inferencing time of different models on MCMD, numbers are displayed in minutes

Model	MCMD _{Java}	MCMD _{C#}	MCMD _{C++}	MCMD _{Py}	MCMD _{JS}	Avg.
CmtGen ^a	702	843	721	846	1060	834.4
NNGen	94	229	176	84	87	134
CoDiSum	35	61	38	47	45	45.20
NMT	6	3	3	3	3	3.60
CoRec	26	29	28	21	30	27
PtrGNCMsg	275	247	321	239	209	258.2

^aOur experiments for CmtGen are based on CPU because the dependent library “Theano” for GPU is stopped developing^b and we cannot experiment with them using GPU

^b<https://groups.google.com/g/theano-users/c/7Poq8BZutBY/m/rNCIfvAEAwAJ>

Summary: More datasets can be used for more comprehensive evaluation since the good performance of a model on one dataset does not mean good performance on other datasets. Removing noisy data (commits with the bot and trivial messages) during model training can improve performance. The duplication of commit data makes the performance of retrieval-based models such as NNGen better.

4.3 Why Do We Need a New Dataset MCMD for Evaluating Commit Message Generation? (RQ3)

From the results shown in Tables 10 and 6, we have the following findings:

- Most of the existing datasets only retain (Diff, Message) information and cannot be used to evaluate models that require more information. For example, as shown in Table 6, only our MCMD dataset can be used to evaluate ATOM. This is because ATOM needs to know the complete code of the modified functions in order to extract the AST information. However, this information is unavailable in existing public datasets.²⁰ Compared to existing datasets, our dataset MCMD provides complete information for each commit. For example, the provided (RepoFullname, SHA) information can be used to obtain the complete functions for AST extraction. Please note that the ATOM results we presented here are only to illustrate the necessity of a richly informative dataset. As part of the ATOM code is not disclosed (as described in Section 3.1), ATOM is out of the scope of this study. We believe future research that requires the use of other commit information can benefit from the complete information provided by MCMD.
- Extremely low scores are observed for CoDiSum on CmtGen_{data} and NNGen_{data}. The reason might be that the size of CmtGen_{data} and NNGen_{data} is not large enough to support the model’s training after filtering the data. CoDiSum is designed to extract additional structure information from code changes in “.java” files. Therefore, code changes that are not related to “.java” files are filtered. After filtering by CoDiSum, there are only hundreds of commits left, which are inadequate for its training. With the large-scale dataset MCMD, we have tens of thousands of commits after filtering to

²⁰Although the ATOM_{data} can be used for evaluating ATOM, it is not publicly available as described in Section 3.2.1.

support the training of CoDiSum. Considering that some filtering steps reduce the size of the original dataset for the model's training, using a larger dataset can reduce the negative impact of insufficient training during the evaluation.

- The multiple-programming-language nature of the MCMD dataset makes it possible to study the migration of commit message generation models to other PLs. From Table 10, we find that the ranking of models on the Java dataset cannot be preserved when migrating to other languages and the best model for different languages may vary. Overall, the retrieval-based model NNGen performs the best, with an average B-Norm score of 17.82. But no model can consistently outperform others.

Summary: A large-scale, multi-language, and information-rich dataset is needed to comprehensively evaluate commit message generation models. Overall, NNGen performs the best, but no model can consistently outperform other models in all PLs. Therefore, when selecting commit message generation models for a new language, we suggest testing multiple models in that language to select the best one.

4.4 What is the Impact of Different Dataset Splitting Strategies? (RQ4)

We analyze the impact of different dataset splitting strategies including splitting by timestamp and splitting by project.

4.4.1 Split by Timestamp

Table 11 shows experimental results on datasets split by timestamp. Compared to Table 10, the performance of all models on all datasets drops consistently, and the BLEU scores of all models drop by 17.88–51.71% on average. This shows that it is more difficult to predict future commit messages based on past data training.

Although the retrieval-based model NNGen shows the best performance in the split by commit setting as shown in Table 10, the results of splitting by timestamp are different. The performance degradation of NNGen is greater than other models, and PtrGNCMsg performs the best. Therefore, in the JIT application scenario, it is not suitable to use datasets split by commit to evaluate models. Moreover, PtrGNCMsg which is based on generation and pointer-generator mechanism has better generalization ability in the JIT scenario.

4.4.2 Split by Project

Table 12 shows experimental results on datasets split by project. Compared to Table 10, the performance of all models on all datasets drops consistently, by 26.93% to 73.41%. This indicates that the split-by-project scenario is much more difficult than the split-by-commit, and models need to have better generalization ability when applied to new projects. We also find that on datasets split by project, PtrGNCMsg shows the best performance.

Furthermore, to emulate the scenario in which we need to generate commit messages for a new project, we conduct a series of *single project experiments* as described in Section 3.3 with the NMT model. From the results shown in Table 13, we can find that the performance of NMT with Cross-Project training is poor, which is consistent with the previous split-by-project conclusion. Within-Project training is much better than Cross-Project, and the performance of the NMT model can be further improved through Full-Project training.

Table 10 Model performance on our dataset MCMD

Model		CmtGen	CoDiSum	NMT	PtrGNCMsg	CoRec	NNGen
Data	Metric						
MCMD _{Java}	BM	6.29	2.00	9.17	8.25	11.39	13.30
	BN	12.39	14.00	13.39	15.33	16.09	17.81
	BC	7.67	5.37	10.24	11.70	12.68	14.46
	R1	13.06	16.84	15.47	19.09	18.95	21.49
	R2	6.23	4.43	7.78	8.32	10.19	11.50
	RL	12.94	16.51	15.33	18.64	18.66	20.87
	ME	14.16	11.34	16.00	19.13	19.58	22.11
MCMD _{C#}	BM	16.87	1.78	20.18	18.92	24.31	22.79
	BN	18.14	12.73	17.32	19.72	22.23	22.92
	BC	14.84	4.74	16.35	17.18	20.62	20.73
	R1	19.41	15.07	20.11	22.32	25.07	25.23
	R2	14.64	3.50	14.74	14.30	18.56	17.50
	RL	19.32	14.78	20.02	21.99	24.87	24.80
	ME	20.18	10.01	19.80	22.33	25.38	26.22
MCMD _{C++}	BM	7.08	3.99	8.37	5.91	10.14	9.57
	BN	11.58	12.46	11.56	13.07	13.80	13.69
	BC	8.83	5.71	9.63	9.97	11.58	10.89
	R1	13.58	15.17	14.11	17.53	16.80	16.81
	R2	8.32	4.54	8.17	7.69	9.79	8.26
	RL	13.53	14.94	14.04	17.08	16.62	16.25
	ME	14.61	10.20	14.75	16.86	17.42	17.18
MCMD _{Py}	BM	5.81	2.63	7.64	8.62	10.03	11.66
	BN	11.10	14.63	11.53	15.99	15.13	16.64
	BC	7.78	5.74	9.56	11.87	12.09	13.19
	R1	13.11	19.45	14.56	21.39	19.14	20.16
	R2	7.47	5.94	8.22	9.91	10.61	10.33
	RL	13.01	18.91	14.41	20.76	18.81	19.44
	ME	15.17	12.08	16.4	21.18	20.29	20.84
MCMD _{JS}	BM	11.66	1.51	15.00	15.15	16.81	14.05
	BN	17.40	11.24	17.08	19.58	19.84	18.03
	BC	11.81	4.31	14.77	16.50	16.91	14.72
	R1	20.12	13.89	20.78	25.11	23.68	21.97
	R2	12.66	2.94	13.07	14.45	15.32	11.61
	RL	19.94	13.53	20.54	24.60	23.36	21.27
	ME	20.50	9.51	21.13	24.61	23.84	22.46
Average	BM	9.54	2.38	12.07	11.37	14.54	14.27
	BN	14.12	13.01	14.18	16.74	17.42	17.82
	BC	10.19	5.18	12.11	13.44	14.77	14.80
	R1	15.86	16.08	17.01	21.09	20.73	21.13
	R2	9.86	4.27	10.40	10.93	12.89	11.84
	RL	15.75	15.73	16.87	20.61	20.46	20.53
	ME	16.92	10.63	17.62	20.82	21.30	21.76

BM, BN, BC are short for B-Moses, B-Norm, and B-CC, R1, R2, RL are short for ROUGE-1, ROUGE-2, ROUGE-L, ME is short for METEOR, respectively

The highest score of the model under each metric for each dataset is marked in bold

Table 11 The performance on MCMD split by timestamp under B-Norm

Model Data	CmtGen	CoDiSum	NMT	NNGen	CoRec	PtrGNCMsg
MCMD _{Java}	8.08	12.71	9.50	10.73	12.94	13.30
	↓34.75%	↓9.21%	↓29.05%	↓39.78%	↓19.58%	↓13.21%
MCMD _{C#}	4.53	4.85	5.15	7.83	9.16	9.38
	↓75.04%	↓61.90%	↓70.27%	↓65.81%	↓58.79%	↓52.45%
MCMD _{C++}	7.08	12.24	8.53	9.30	11.73	10.94
	↓38.85%	↓1.77%	↓26.21%	↓32.05%	↓15.00%	↓16.29%
MCMD _{Py}	5.50	12.46	7.31	9.36	11.07	13.21
	↓50.49%	↓14.83%	↓36.60%	↓43.78%	↓26.83%	↓17.38%
MCMD _{JS}	8.91	11.17	11.58	12.07	15.94	18.07
	↓48.77%	↓0.62%	↓32.20%	↓33.04%	↓19.66%	↓7.73%
Average	6.82	11.17	8.41	9.86	12.17	12.98
	↓51.71%	↓17.88%	↓40.65%	↓44.67%	↓30.15%	↓22.45%

The score drop rates compared to Table 10 are shown in gray

The highest score of the model under each dataset is marked in bold

Summary: The dataset splitting strategies have a significant impact on the evaluation of commit message generation models. Under the split-by-timestamp or split-by-project strategies, the evaluation scores of models are significantly lower than that of split-by-commit, and the PtrGNCMsg model is overall the best. Moreover, to achieve the best performance, it is recommended to train models with data from both the target project and other projects.

Table 12 The performance on MCMD split by repository under B-Norm

Model Data	CmtGen	CoDiSum	NMT	NNGen	CoRec	PtrGNCMsg
MCMD _{Java}	5.20	10.23	7.94	5.67	6.76	7.92
	↓58.03%	↓26.93%	↓40.70%	↓68.16%	↓57.99%	↓48.34%
MCMD _{C#}	4.82	8.43	5.95	9.89	7.12	8.08
	↓73.41%	↓33.78%	↓65.65%	↓56.85%	↓67.97%	↓59.03%
MCMD _{C++}	4.47	2.87	5.73	3.90	5.17	6.28
	↓61.37%	↓76.97%	↓50.43%	↓71.51%	↓62.54%	↓51.95%
MCMD _{Py}	7.61	9.23	5.29	4.66	7.12	8.79
	↓31.46%	↓36.91%	↓54.12%	↓72.00%	↓52.94%	↓45.03%
MCMD _{JS}	7.05	8.02	7.39	5.72	9.78	11.98
	↓59.50%	↓28.65%	↓56.73%	↓68.28%	↓50.71%	↓38.82%
Average	5.83	7.76	6.46	5.97	7.19	8.61
	↓58.71%	↓40.39%	↓54.43%	↓66.50%	↓58.72%	↓48.57%

The score drop rates compared to Table 10 are shown in gray

The highest score of the model under each dataset is marked in bold

Table 13 The performance of NMT on MCMD under *single project experiments*

	MCMD _{Java}	MCMD _{C#}	MCMD _{C++}	MCMD _{Py}	MCMD _{JS}	Avg.
Cross-Proj.	6.27	5.03	2.52	1.61	2.40	3.57
Within-Proj.	9.35	28.82	8.47	7.28	18.33	14.45
Full-Proj.	10.28	40.04	8.45	7.41	18.92	17.02

The highest score of the setting under each dataset is marked in bold

Table 14 CodeBERT on MCMD (fine-tuning with different multi-PL combinations)

Fine-Tune \ Test						Avg.
	Java	Py	JS	C++	C#	
Java	18.98	7.74	8.93	6.70	6.11	9.69
Py	<u>7.06</u>	19.39	11.04	7.36	6.85	10.34
JS	<u>6.92</u>	8.12	23.01	6.82	6.52	10.28
C++	<u>7.08</u>	9.45	11.22	17.09	7.26	10.42
C#	<u>6.82</u>	7.65	9.35	6.81	24.03	10.93
Java, Py	18.89	19.28	11.25	7.71	6.87	12.80
Java, JS	18.86	9.27	22.98	7.39	6.87	13.07
Java, C++	18.56	9.20	11.90	16.98	7.48	12.82
Java, C#	18.76	8.44	10.28	7.49	23.97	13.79
Py, JS	7.90	19.47	23.13	7.88	7.97	13.27
Py, C++	7.39	17.56	11.58	16.04	7.59	12.03
Py, C#	7.66	19.33	11.22	7.69	24.27	14.03
JS, C++	7.05	9.29	17.38	12.57	7.37	10.73
JS, C#	7.28	8.70	23.12	7.48	24.32	14.18
C++, C#	7.64	9.63	11.51	17.02	23.65	13.89
Java, Py, JS	18.77	19.34	23.17	7.95	7.39	15.32
Java, Py, C++	18.86	19.26	12.53	17.34	8.05	15.21
Java, Py, C#	18.67	19.23	11.36	7.83	23.84	16.19
Java, JS, C++	19.00	9.69	23.30	17.35	7.90	15.45
Java, JS, C#	18.91	9.07	23.20	7.91	24.31	16.68
Java, C++, C#	18.58	9.02	11.91	17.09	23.87	16.09
Py, JS, C++	7.40	14.53	17.41	12.98	8.36	12.14
Py, JS, C#	8.33	19.56	23.31	8.17	24.42	16.76
Py, C++, C#	7.83	16.39	12.03	14.47	21.24	14.39
JS, C++, C#	7.44	9.81	18.83	14.01	20.59	14.14
Java, Py, JS, C++	18.88	19.38	23.28	17.48	8.23	17.45
Java, Py, JS, C#	18.93	19.41	23.39	8.26	24.15	18.83
Java, Py, C++, C#	15.80	16.42	12.05	14.48	21.12	15.97
Java, JS, C++, C#	18.58	9.53	23.15	17.23	23.95	18.49
Py, JS, C++, C#	8.29	19.43	23.19	17.52	24.10	18.51
Java, Py, JS, C++, C#	19.10	19.58	23.37	17.61	24.29	20.79

All scores are in B-Norm. Scores in other metrics can be found in Appendix B.4

The highest score on the test under each fine-tuning setting is marked in bold

The highest score on the test which is fine-tuned in only one PL is marked in underlined

4.5 What is the Effectiveness of the Pre-trained Model in Commit Message Generation? (RQ5)

We evaluate CodeBERT, a recent pre-trained source code model, on our MCMD dataset. We experiment with different combinations of the five PLs, including Java, Python (Py), JavaScript (JS), C++, and C#. Repositories in different PL combinations are included in the fine-tuning data. The fine-tuned model is then tested on each of the five PLs. The experimental results are shown in Table 14. This table can be divided into five groups. Each line in the table represents one PL combination setting. The six scores in each row represent the performance tested on each of the five programming languages and their average performance. Based on the results, we have the following findings:²¹

- In each group, the highest score of each column is achieved by a combination containing the target programming language. Moreover, the performance of model fine-tuning on the dataset with the target PL has nearly two times more superiority than others. For example, in the second column of the first group, when evaluating the five models individually fine-tuned in five PLs (Java, Py, JS, C++, and C#), the model fine-tuned in Java dataset achieves best (18.98) than others (6.82–7.08). These differences indicate that it is better to fine-tune the model with repositories in the same programming language as the target programming language.
- For a target programming language, using repositories in that language to perform fine-tuning can lead to similar results as using repositories in multiple programming languages. For example, fine-tuning with only Java's repositories can achieve a B-Norm of 18.98, while fine-tuning with all PL's repositories can achieve a slightly better result of 19.10. However, it is more time-consuming to fine-tune with repositories in multiple PLs because the number of commits increases.
- The overall performance of the model is improved after the model is fine-tuned with more PLs. As shown in the Average (Avg.) column of Table 14, the average B-Norm scores across PLs with single PL fine-tuning range from 9.69 to 10.93, and the average scores with four-PLs fine-tuning range from 17.45 to 18.83. This finding holds true for each PL combination. For example, the B-Norm value of 9.69 indicates the overall performance of CodeBERT after fine-tuned with Java data, and 17.45 indicates the overall performance of CodeBERT after fine-tuned with Java, Python, JavaScript, and C++ data.

Summary: When the training data used for fine-tuning contains the target PL, the effect is better than that without the target PL. For a certain target PL, adding training data from more other PLs to fine-tune the model can improve the performance a bit, but the gain is limited. For the overall performance of all languages, adding more PLs during training can improve performance.

²¹Note that although Table 14 is the result under the B-Norm metric, our findings still hold on other metrics (with results shown in Appendix B.4) as well.

5 Discussion

5.1 Future Research Directions

As described in Section 4.1, evaluation metrics are important for the evaluation of commit message generation models, and using different metrics may lead to different conclusions. We have only studied the metrics that have been used in this task. A possible future direction is to study whether other metrics used in some NLP tasks are better than B-Norm or even design a new metric for commit message generation task.

Another future direction is to leverage more context information of repositories in the design of commit message generation models. In addition to the AST information that ATOM has studied, there is other information that could be explored, such as programming languages' features, contributors, history changes on the target code, associated bug reports, etc. In this regard, our dataset MCMD, which is information-rich and can be traced back to original repositories to extract all information mentioned above, can be very helpful for future research. We have made our dataset public.

Moreover, as discussed in Sections 4.3 and 4.4, there are several aspects that deserve more attention when evaluating commit message generation models in the future, including multiple PLs, dataset splitting strategies, etc. More specifically, in the settings of split-by-timestamp and split-by-project, the performance of existing models needs to be further improved. Furthermore, the overall best splitting strategy may not be optimal for some specific projects. In our splitting strategy, we treat all projects equally, which means that Tables 10, 11 and 12 are the overall experimental results for all projects. Some specific projects may have different characteristics so the evaluation of models on these projects needs to apply different splitting strategies. It can be a potentially valuable direction to explore how different splitting strategies may affect the performance of models on these projects.

As shown in Section 4.5, pre-trained language models such as CodeBERT show great potential but are also full of challenges. Therefore, we discuss possible future research directions on pre-trained models for commit message generation:

- One research direction is how to make better use of the training data from different PLs. As the results in Section 4.5 show, the overall performance of CodeBERT can be improved after fine-tuning the model with more PLs. However, due to the limited PLs and combinations we experimented with, we felt there would be more space for exploration. Therefore, it is a worthy research direction to study the integration methods of multi-PL data for pre-trained models to further improve the performance of commit message generation.
- Another direction is to choose a suitable code pre-trained model to fine-tune or even design new pre-trained models dedicated to code diff.
 1. Our experimental results in Section 4.5 preliminarily verify the effectiveness of applying code pre-trained model for commit message generation. The findings of our experiments on CodeBERT are yet to be further studied in other code pre-trained models such as GraphCodeBERT (Guo et al. 2021), CuBERT (Kanade et al. 2020), CodeT5 (Wang et al. 2021), PLBART (Ahmad et al. 2021), and CODE-MVP (Wang et al. 2022).
 2. Furthermore, we suspect that directly applying code pre-trained models may not be the most suitable approach, as there are some natural data-level differences between *code* and *code diff*. For example, they have different structures and formats. Designing new pre-train tasks for code diff and training new pre-trained models that are specifically for code diff may help generate better commit messages.

- It is also worth investigating whether some of the new findings in the natural language pre-training area apply to *code* and *code diff* pre-training. For example, in the field of natural language processing, researchers find that domain similarity affects the quality of pre-trained word embeddings (Lample et al. 2018; Conneau et al. 2020). If these findings of natural languages also apply to programming languages, we can leverage them to study how to use data from more similar programming languages for pre-training to further improve code or code diff representations.

5.2 Possible Ways to Improve?

Our findings suggest that commit message generation still has a long way to go. We now show that the performance of commit message generation models could be improved through small changes. Note that we do not aim to design a full-scale model here. Our purpose is to show that there is still ampler room for further improving existing models.

As the experimental results in Table 10 suggest, a retrieval-based model is a simple yet effective approach to generating commit messages. However, NNGen only uses the “bag of words” (which is 1-gram) as the retrieval index, which is basic and can be optimized. We can use a simple and different representation. Inspired by the design of BLEU concerning the precision of n-gram matches, we change the representation of diff tokens from 1-gram to n-gram. NNGen-Gram4 in Table 15 means that we represent all of the tokens including 1,2,3,4-gram rather than 1-gram only. Besides the representation, the retrieval method is also changed in our attempt. As described in Section 4.1, adding a smoothing function can affect the BLEU score. The original BLEU metric used by NNGen may not be the most suitable option for retrieval. Therefore, we try to add the smoothing function (shown as NNGen-Smooth in Table 15) to the retrieval method. We compare the results of our two attempts NNGen-Gram4 and NNGen-Smooth with NNGen on NNGen_{data}, as shown in Table 15. The two variants achieve consistently higher scores than the original NNGen and NNGen-Smooth-Gram4, the combination of them further improves the performance.

5.3 Threats to Validity

We have identified the main threats to validity as follows:

- *Data Quality*. The quality of the data could be a threat to validity. To mitigate this threat, we have used multiple filtering rules following previous work (Liu et al. 2019) to obtain a set of relatively good-quality commit messages. But there might still exist low-quality data. It is possible to further improve our dataset MCMD to obtain higher quality.
- *Human Labeling Bias*. Our manual annotation of the quality of commit message may be biased, and interrater reliability could be a threat to validity: bias may exist in the scores assigned to the same sentence by different raters. We attempted to mitigate this

Table 15 Retrieval index experiments on NNGen_{data}

	Model	B-Moses	B-Norm	B-CC
	NNGen	16.41	23.07	16.77
	NNGen-Smooth	16.47	23.12	16.80
	NNGen-Gram4	17.26	23.92	17.60
	NNGen-Smooth-Gram4	17.48	24.03	17.63

The highest score of the model under each metric is marked in bold

threat by: (1) making clear scoring rules as shown in Table 5 before actual scoring, and (2) having discussions on disagreement cases so that the standard deviations among all raters are small.

- *Data Sampling.* 100 sampled commit messages are used in the human evaluation. These samples are randomly selected from the messages with the variance among the three BLEU variants larger than 30 because the aim is to find which BLEU variant correlates with human scores the most, especially when there is a large variance between the BLEU scores. This sampling could be a threat to validity. Larger-scale human evaluation can be conducted to alleviate this threat.
- *Programming Languages.* There are many programming languages with various characteristics. Although we have expanded the language diversity in MCMD to 5 popular PLs, it is still not exhaustive. Caution is needed when applying our findings to other PLs.
- *Replication.* There could exist potential errors in our implementations and experiments. To mitigate this threat, we reuse the existing implementations of the models from the original authors when possible. The new implementation in our experiments (such as data collection, scripts for comparison experiments, and the implementation of the improved NNGen) are double-checked by multiple experts to ensure the correctness.

6 Conclusion

In this paper, we conduct an in-depth analysis of the datasets and models for the commit message generation task. We have investigated several aspects, including: evaluation metrics, multiple programming languages, and dataset splitting strategies, etc. Our study points out that all these aspects have a large impact on the evaluation. We believe that the results and findings in our study can be of great help for practitioners and researchers working on this interesting area.

Our source code and data are available at <https://anonymous.4open.science/r/CommitMessageEmpirical>.

Appendix A: Searched Results

All of the searched results can be available at our repository <https://anonymous.4open.science/r/CommitMessageEmpirical/survey>. “Searched_Results.csv” contains 583 papers’ titles searched from four databases. Other “.html” files record web page results when searching from different databases. For example, “IEEE.html” is the search result page when searching from <https://ieeexplore.ieee.org>.

Appendix B: More Experimental Results

B.1 Models performance on ROUGE and METEOR

As shown in Table 16, rankings of models are consistent mostly under ROUGE-1, ROUGE-2, ROUGE-L and METEOR. We also calculated the correlation between each of them. We find that 44 of the 48 Spearman’s Rank correlation coefficients are significantly higher than 0.78, which means these metrics are generally consistent across evaluations.

Table 16 Models performance on ROUGE and METEOR

Model		CmtGen	CoDiSum	NMT	PtrGNCSmsg	NNGen	CoRec	ATOM
Data	Metric							
CmtGen _{data}	ROUGE-1	33.56	11.64	28.46	34.99	38.55	40.40	/
	ROUGE-2	24.54	0.03	20.22	23.96	28.57	29.91	/
	ROUGE-L	33.30	11.64	28.30	34.77	38.21	40.12	/
	METEOR	33.28	4.57	27.87	31.79	37.49	38.54	/
NNGen _{data}	ROUGE-1	24.57	11.32	15.92	23.47	27.43	29.11	/
	ROUGE-2	14.51	0.45	7.04	9.41	15.73	17.20	/
	ROUGE-L	24.29	11.32	15.74	23.12	27.00	28.71	/
	METEOR	24.35	4.82	15.43	19.19	26.56	27.53	/
CoDiSum _{data}	ROUGE-1	10.73	19.00	11.24	16.43	10.79	12.52	/
	ROUGE-2	2.27	4.66	2.26	3.58	3.31	3.33	/
	ROUGE-L	10.59	18.62	11.11	16.06	10.53	12.27	/
	METEOR	9.83	12.30	9.61	12.16	10.62	11.30	/
MCMD _{Java}	ROUGE-1	13.06	16.84	15.47	19.09	21.49	18.95	19.10
	ROUGE-2	6.23	4.43	7.78	8.32	11.50	10.19	9.58
	ROUGE-L	12.94	16.51	15.33	18.64	20.87	18.66	18.60
	METEOR	14.16	11.34	16.00	19.13	22.11	19.58	20.80

The highest score of the model under each metric for each dataset is marked in bold

B.2 Models Performance on MCMD Split by Timestamp on ROUGE and METEOR

Table 17 shows experimental results on MCMD split by timestamp. Compared to Table 10, the performance of all models on all PLs of MCMD drops consistently. This shows that it is more difficult to predict future commit messages based on past data training.

Table 17 The performance on MCMD split by timestamp

Data	Metric	CmtGen	CoDiSum	NMT	NNGen	CoRec	PtrGNCSmsg
MCMD _{Java}	ROUGE-1	8.80	14.94	11.21	11.97	14.55	16.15
	ROUGE-2	4.03	4.09	4.75	4.19	7.17	5.93
	ROUGE-L	8.74	14.72	11.13	11.57	14.37	15.71
	METEOR	9.23	10.17	13.86	14.34	16.61	17.55
MCMD _{C#}	ROUGE-1	7.07	5.52	8.43	8.86	11.43	11.78
	ROUGE-2	3.86	0.32	2.35	3.50	5.97	4.36
	ROUGE-L	7.04	5.50	8.42	8.67	11.32	11.55
	METEOR	5.97	3.31	7.71	10.45	11.58	12.18
MCMD _{C++}	ROUGE-1	9.84	15.10	10.62	10.83	14.46	13.93
	ROUGE-2	6.21	4.80	5.80	4.49	8.04	5.34
	ROUGE-L	9.80	14.78	10.60	10.53	14.33	13.50
	METEOR	8.96	10.29	11.10	12.29	15.40	14.33

Table 17 (continued)

Data	Metric	CmtGen	CoDiSum	NMT	NNGen	CoRec	PtrGNCMsg
MCMD _{Py}	ROUGE-1	7.62	16.66	8.45	10.11	13.07	17.70
	ROUGE-2	4.64	4.48	5.06	2.88	5.69	6.54
	ROUGE-L	7.60	16.19	8.42	9.73	12.84	17.02
	METEOR	6.71	10.06	11.17	13.87	16.28	20.01
MCMD _{JS}	ROUGE-1	12.25	12.74	14.58	13.54	18.87	23.74
	ROUGE-2	6.21	1.97	7.87	5.65	10.81	12.58
	ROUGE-L	12.05	12.43	14.44	13.07	18.57	23.10
	METEOR	11.13	7.81	15.54	16.89	20.75	23.98
Average	ROUGE-1	9.12	12.99	10.66	11.06	14.48	16.66
	ROUGE-2	4.99	3.13	5.17	4.14	7.54	6.95
	ROUGE-L	9.05	12.72	10.60	10.71	14.29	16.18
	METEOR	8.40	8.33	11.88	13.57	16.12	17.61

The highest score of the model under each metric for each dataset is marked in bold

B.3 Models Performance on MCMD Split by Project on ROUGE and METEOR

Table 18 shows experimental results on MCMD split by project. Compared to Table 10, the performance of all models on all PLs of MCMD drops in general, which indicates that the split-by-project scenario is much more difficult than the split-by-commit.

Table 18 The performance on MCMD split by project

Data	Metric	CmtGen	CoDiSum	NMT	NNGen	CoRec	PtrGNCMsg
MCMD _{Java}	ROUGE-1	7.54	12.65	11.41	5.78	8.11	10.97
	ROUGE-2	3.21	2.75	7.86	2.08	3.85	3.75
	ROUGE-L	7.42	12.47	11.37	5.63	8.02	10.67
	METEOR	10.62	8.29	10.51	6.48	7.53	9.41
MCMD _{C#}	ROUGE-1	5.39	9.74	8.22	10.39	9.21	10.17
	ROUGE-2	2.18	1.96	4.25	6.92	4.60	3.93
	ROUGE-L	5.29	9.57	8.19	10.30	9.12	9.99
	METEOR	6.95	6.44	7.92	11.05	9.11	9.63
MCMD _{C++}	ROUGE-1	4.90	3.67	7.71	3.89	6.68	8.90
	ROUGE-2	3.12	1.14	5.45	0.68	3.13	2.43
	ROUGE-L	4.89	3.60	7.70	3.79	6.62	8.61
	METEOR	5.15	2.61	7.36	4.91	6.93	8.03
MCMD _{Py}	ROUGE-1	7.90	12.24	8.58	4.80	9.38	13.46
	ROUGE-2	4.26	3.59	5.45	0.88	4.56	4.85
	ROUGE-L	7.88	11.92	8.53	4.69	9.25	13.03
	METEOR	9.31	7.35	7.99	5.40	9.07	10.93
MCMD _{JS}	ROUGE-1	9.16	10.77	11.10	6.35	12.45	16.64
	ROUGE-2	5.88	3.78	7.60	1.54	6.48	6.82
	ROUGE-L	9.06	10.61	11.01	6.13	12.27	16.12

Table 18 (continued)

Data	Metric	CmtGen	CoDiSum	NMT	NNGen	CoRec	PtrGNCSg
Average	METEOR	10.46	7.54	11.14	7.55	12.72	15.77
	ROUGE-1	6.98	9.81	9.40	6.24	9.17	12.03
	ROUGE-2	3.73	2.64	6.12	2.42	4.52	4.36
	ROUGE-L	6.91	9.63	9.36	6.11	9.06	11.68
	METEOR	8.50	6.45	8.98	7.08	9.07	10.75

The highest score of the model under each metric for each dataset is marked in bold

B.4 CodeBERT Performance on MCMD (Fine-Tuning with Different Multi-PL Combinations) on ROUGE, METEOR, B-Moses, and B-CC

Tables 19, 20, 21, 22, 23, and 24 shows the experimental results on ROUGE-1, ROUGE-2, ROUGE-L, METEOR, B-Moses and B-CC respectively. These scores reflect the performance of CodeBERT fine-tuned with different combinations of the five PLs. The findings described in Section 4.5 still can be found from these tables.

Table 19 CodeBERT on MCMD (fine-tuning with different multi-PL combinations)

Fine-Tune \ Test						
	Java	Py	JS	C++	C#	Avg.
Java	22.74	10.70	12.27	9.40	7.28	12.48
Py	10.31	25.28	15.98	11.38	9.24	14.44
JS	9.72	11.89	28.14	10.36	8.91	13.80
C++	9.23	13.59	15.83	21.62	9.17	13.89
C#	8.83	10.58	13.26	10.14	27.31	14.02
Java, Py	22.47	25.02	15.94	11.27	8.93	16.72
Java, JS	22.41	13.71	28.05	11.16	8.88	16.84
Java, C++	21.97	13.51	16.44	21.42	9.51	16.57
Java, C#	22.27	12.25	14.31	11.20	27.21	17.45
Py, JS	11.40	25.33	28.28	12.23	10.71	17.59
Py, C++	9.59	23.30	16.31	20.53	9.59	15.86
Py, C#	10.50	25.01	16.13	11.76	27.50	18.18
JS, C++	8.71	13.03	21.61	16.15	9.00	13.70
JS, C#	10.33	12.90	28.18	11.60	27.67	18.14
C++, C#	10.13	13.59	15.94	21.45	26.85	17.59
Java, Py, JS	22.47	25.20	28.33	11.89	9.76	19.53
Java, Py, C++	22.53	25.16	17.61	22.04	10.43	19.56
Java, Py, C#	22.25	25.01	16.34	12.01	27.13	20.55
Java, JS, C++	22.72	14.37	28.54	22.03	10.16	19.57
Java, JS, C#	22.57	13.64	28.40	12.08	27.58	20.85
Java, C++, C#	22.09	13.23	16.51	21.59	27.10	20.11
Py, JS, C++	8.95	19.51	21.71	16.25	9.66	15.21
Py, JS, C#	11.41	25.42	28.48	12.41	27.70	21.08
Py, C++, C#	10.09	21.84	16.36	18.59	24.45	18.27
JS, C++, C#	9.41	13.54	23.56	18.06	23.71	17.66
Java, Py, JS, C++	22.60	25.25	28.57	22.21	10.70	21.86

Table 19 (continued)

Test Fine-Tune						
	Java	Py	JS	C++	C#	Avg.
Java, Py, JS, C#	22.72	25.33	28.59	12.67	27.48	23.36
Java, Py, C++, C#	19.23	21.82	16.45	18.62	24.50	20.12
Java, JS, C++, C#	22.12	14.02	28.34	21.85	27.22	22.71
Py, JS, C++, C#	11.29	25.33	28.41	22.22	27.46	22.94
Java, Py, JS, C++, C#	22.89	25.51	28.62	22.43	27.70	25.43

All scores are in ROUGE-1

The highest score on the test under each fine-tuning setting is marked in bold

Table 20 CodeBERT on MCMD (fine-tuning with different multi-PL combinations)

Test Fine-Tune						
	Java	Py	JS	C++	C#	Avg.
Java	11.95	3.48	4.64	3.05	2.32	5.09
Py	3.49	13.48	6.94	3.80	3.20	6.18
JS	2.82	3.98	16.89	3.19	2.57	5.89
C++	3.07	5.30	6.71	11.71	3.39	6.03
C#	3.08	3.70	5.45	3.65	19.13	7.00
Java, Py	11.94	13.45	6.92	3.78	3.29	7.87
Java, JS	11.93	4.96	16.96	3.82	3.05	8.14
Java, C++	11.70	5.52	7.50	11.80	3.68	8.04
Java, C#	11.84	4.50	6.02	3.97	18.94	9.05
Py, JS	3.64	13.53	17.08	4.12	3.79	8.43
Py, C++	3.65	12.55	7.49	11.21	3.98	7.78
Py, C#	3.42	13.48	7.01	4.22	19.08	9.44
JS, C++	3.16	5.25	12.93	8.74	3.72	6.76
JS, C#	3.37	4.84	17.07	4.11	19.24	9.73
C++, C#	3.85	5.45	7.14	11.70	18.79	9.38
Java, Py, JS	11.98	13.61	17.18	4.09	3.57	10.08
Java, Py, C++	12.05	13.63	7.98	12.10	4.20	9.99
Java, Py, C#	11.83	13.46	6.97	4.11	18.81	11.04
Java, JS, C++	12.11	5.71	17.27	12.02	3.75	10.17
Java, JS, C#	12.05	5.15	17.20	4.27	19.17	11.57
Java, C++, C#	11.72	5.08	7.51	11.86	18.86	11.00
Py, JS, C++	3.35	9.64	12.62	8.46	4.10	7.63
Py, JS, C#	3.69	13.69	17.19	4.41	19.27	11.65
Py, C++, C#	3.84	11.29	7.65	9.62	16.91	9.86
JS, C++, C#	3.58	5.46	13.98	9.59	16.51	9.83
Java, Py, JS, C++	11.89	13.67	17.21	12.19	4.15	11.82
Java, Py, JS, C#	12.09	13.75	17.33	4.46	19.06	13.34
Java, Py, C++, C#	10.21	11.17	7.63	9.70	17.00	11.14
Java, JS, C++, C#	11.85	5.48	17.27	12.09	18.98	13.14
Py, JS, C++, C#	3.94	13.56	17.19	12.10	19.03	13.17
Java, Py, JS, C++, C#	12.16	13.73	17.37	12.29	19.20	14.95

All scores are in ROUGE-2

The highest score on the test under each fine-tuning setting is marked in bold

Table 21 CodeBERT on MCMD (fine-tuning with different multi-PL combinations)

Fine-Tune \ Test						
	Java	Py	JS	C++	C#	Avg.
Java	22.34	10.38	11.97	9.14	7.16	12.20
Py	10.02	24.63	15.60	11.08	9.08	14.08
JS	9.43	11.57	27.68	10.11	8.76	13.51
C++	9.01	13.23	15.46	21.25	9.02	13.59
C#	8.65	10.34	13.01	9.93	27.03	13.79
Java, Py	22.06	24.36	15.49	10.96	8.76	16.33
Java, JS	22.01	13.27	27.56	10.81	8.72	16.47
Java, C++	21.60	13.14	16.06	21.04	9.35	16.24
Java, C#	21.89	11.88	13.95	10.88	26.91	17.10
Py, JS	11.00	24.64	27.77	11.85	10.51	17.15
Py, C++	9.30	22.69	15.92	20.15	9.42	15.50
Py, C#	10.18	24.35	15.70	11.45	27.20	17.78
JS, C++	8.45	12.65	21.22	15.82	8.84	13.40
JS, C#	10.01	12.50	27.69	11.28	27.35	17.77
C++, C#	9.87	13.24	15.58	21.07	26.56	17.26
Java, Py, JS	22.05	24.52	27.82	11.51	9.59	19.10
Java, Py, C++	22.12	24.48	17.12	21.61	10.25	19.11
Java, Py, C#	21.87	24.36	15.86	11.64	26.82	20.11
Java, JS, C++	22.29	13.90	28.02	21.57	9.95	19.15
Java, JS, C#	22.16	13.20	27.89	11.67	27.26	20.44
Java, C++, C#	21.70	12.84	16.10	21.21	26.79	19.73
Py, JS, C++	8.66	18.90	21.30	15.88	9.49	14.85
Py, JS, C#	11.02	24.70	27.96	12.02	27.38	20.62
Py, C++, C#	9.77	21.14	15.94	18.14	24.15	17.83
JS, C++, C#	9.12	13.13	23.12	17.61	23.44	17.28
Java, Py, JS, C++	22.15	24.54	28.04	21.76	10.48	21.39
Java, Py, JS, C#	22.28	24.65	28.08	12.23	27.16	22.88
Java, Py, C++, C#	18.83	21.09	15.99	18.16	24.19	19.65
Java, JS, C++, C#	21.71	13.60	27.84	21.39	26.90	22.29
Py, JS, C++, C#	10.91	24.62	27.88	21.73	27.14	22.46
Java, Py, JS, C++, C#	22.43	24.79	28.07	21.97	27.36	24.93

All scores are in ROUGE-L

The highest score on the test under each fine-tuning setting is marked in bold

Table 22 CodeBERT on MCMD (fine-tuning with different multi-PL combinations)

Fine-Tune \ Test	Java	Py	JS	C++	C#	Avg.
Java	20.99	7.96	10.04	7.02	6.37	10.48
Py	7.79	22.69	12.31	7.64	7.42	11.57
JS	7.47	7.91	26.35	7.00	6.93	11.13
C++	6.98	9.03	12.59	19.02	7.47	11.02
C#	7.70	7.38	10.08	7.42	26.01	11.72
Java, Py	20.88	22.64	12.77	7.77	7.35	14.28
Java, JS	20.82	9.05	26.31	7.56	7.25	14.20
Java, C++	20.35	9.18	13.46	18.90	7.98	13.98
Java, C#	20.71	8.37	11.43	7.83	25.83	14.83
Py, JS	8.79	22.80	26.51	8.04	8.32	14.89
Py, C++	7.41	19.59	12.68	17.46	7.80	12.99
Py, C#	8.53	22.70	12.51	8.14	26.21	15.62
JS, C++	6.94	8.47	18.86	12.72	7.59	10.92
JS, C#	8.15	8.63	26.40	7.68	26.28	15.43
C++, C#	8.10	9.37	13.01	19.00	25.68	15.03
Java, Py, JS	20.82	22.71	26.53	8.12	7.66	17.17
Java, Py, C++	20.83	22.70	14.48	19.63	8.69	17.26
Java, Py, C#	20.54	22.49	12.67	8.04	25.77	17.90
Java, JS, C++	21.06	9.65	26.72	19.55	8.43	17.08
Java, JS, C#	20.91	8.90	26.65	8.12	26.25	18.16
Java, C++, C#	20.52	8.90	13.35	19.19	25.88	17.57
Py, JS, C++	7.20	14.62	18.50	12.89	8.14	12.27
Py, JS, C#	9.32	22.92	26.66	8.44	26.42	18.75
Py, C++, C#	7.92	18.13	13.06	15.13	22.27	15.30
JS, C++, C#	7.35	8.81	20.26	14.42	21.46	14.46
Java, Py, JS, C++	20.86	22.76	26.69	19.74	8.83	19.77
Java, Py, JS, C#	20.93	22.96	26.79	8.72	26.09	21.10
Java, Py, C++, C#	16.58	18.10	12.96	15.09	22.44	17.03
Java, JS, C++, C#	20.62	9.47	26.53	19.50	25.95	20.41
Py, JS, C++, C#	9.01	22.79	26.62	19.65	26.08	20.83
Java, Py, JS, C++, C#	21.21	22.98	26.82	19.79	26.27	23.41

All scores are in METEOR

The highest score on the test under each fine-tuning setting is marked in bold

Table 23 CodeBERT on MCMD (fine-tuning with different multi-PL combinations)

Fine-Tune \ Test						
	Java	Py	JS	C++	C#	Avg.
Java	10.79	2.10	3.73	1.98	1.60	4.04
Py	2.32	10.55	5.05	1.67	1.65	4.25
JS	1.93	2.02	17.68	1.67	1.48	4.96
C++	2.04	2.75	5.63	10.58	1.94	4.59
C#	2.21	1.73	3.41	1.97	24.25	6.71
Java, Py	10.81	10.66	5.58	1.97	2.12	6.23
Java, JS	10.75	2.56	17.77	2.05	2.06	7.04
Java, C++	10.54	2.90	6.32	10.58	2.49	6.57
Java, C#	10.78	2.27	4.59	2.16	24.03	8.77
Py, JS	2.46	10.87	17.89	1.85	2.18	7.05
Py, C++	2.50	9.56	6.06	10.00	2.57	6.14
Py, C#	2.46	10.84	5.24	2.02	24.20	8.95
JS, C++	2.41	2.95	12.57	7.31	2.73	5.59
JS, C#	2.42	2.44	17.87	1.95	24.28	9.79
C++, C#	3.08	2.97	5.89	10.70	23.86	9.30
Java, Py, JS	10.83	10.84	17.86	1.95	2.08	8.71
Java, Py, C++	10.81	10.86	6.83	10.98	2.89	8.47
Java, Py, C#	10.70	10.74	5.38	1.96	23.78	10.51
Java, JS, C++	10.94	2.95	18.06	10.92	2.66	9.11
Java, JS, C#	10.92	2.53	18.06	2.17	24.33	11.60
Java, C++, C#	10.76	2.67	6.22	10.84	23.99	10.90
Py, JS, C++	2.55	6.62	12.10	7.60	3.11	6.40
Py, JS, C#	2.64	11.00	17.96	2.22	24.41	11.65
Py, C++, C#	2.89	8.53	6.32	8.49	21.42	9.53
JS, C++, C#	2.66	3.15	13.47	8.22	21.03	9.71
Java, Py, JS, C++	10.81	10.97	18.02	11.18	3.03	10.80
Java, Py, JS, C#	10.90	11.02	18.17	2.35	24.09	13.31
Java, Py, C++, C#	8.57	8.34	6.14	8.42	21.60	10.61
Java, JS, C++, C#	10.88	2.89	18.07	11.08	24.13	13.41
Py, JS, C++, C#	2.98	10.90	18.02	11.04	24.03	13.39
Java, Py, JS, C++, C#	11.03	11.10	18.13	11.14	24.19	15.12

All scores are in B-Moses

The highest score on the test under each fine-tuning setting is marked in bold

Table 24 CodeBERT on MCMD (fine-tuning with different multi-PL combinations)

Fine-Tune \ Test						
	Java	Py	JS	C++	C#	Avg.
Java	13.02	3.55	4.86	3.23	2.92	5.51
Py	3.39	12.97	6.41	3.04	2.92	5.75
JS	3.14	3.30	17.83	2.84	2.72	5.97
C++	3.26	4.09	6.55	12.02	3.39	5.86
C#	3.61	3.04	4.83	3.25	20.28	7.00
Java, Py	13.04	12.98	6.78	3.23	3.33	7.87
Java, JS	13.00	3.94	17.89	3.29	3.24	8.27
Java, C++	12.76	4.19	7.25	11.98	3.80	8.00
Java, C#	12.97	3.65	5.81	3.42	20.12	9.19
Py, JS	3.75	13.16	17.99	3.18	3.49	8.31
Py, C++	3.70	11.60	7.25	11.25	3.95	7.55
Py, C#	3.78	13.11	6.55	3.35	20.30	9.42
JS, C++	3.52	4.15	13.20	8.23	3.96	6.61
JS, C#	3.45	3.78	18.00	3.20	20.41	9.77
C++, C#	4.05	4.31	7.00	12.00	19.94	9.46
Java, Py, JS	13.02	13.15	18.03	3.35	3.39	10.19
Java, Py, C++	13.05	13.14	7.89	12.35	4.13	10.11
Java, Py, C#	12.84	12.98	6.54	3.30	19.91	11.11
Java, JS, C++	13.18	4.32	18.15	12.28	3.89	10.36
Java, JS, C#	13.11	3.91	18.13	3.45	20.38	11.80
Java, C++, C#	12.87	3.97	7.19	12.17	20.05	11.25
Py, JS, C++	3.75	8.24	12.88	8.39	4.58	7.57
Py, JS, C#	4.04	13.26	18.10	3.51	20.47	11.88
Py, C++, C#	4.10	10.34	7.59	9.47	17.56	9.81
JS, C++, C#	3.76	4.40	14.11	9.19	17.09	9.71
Java, Py, JS, C++	12.99	13.23	18.09	12.50	4.24	12.21
Java, Py, JS, C#	13.07	13.28	18.23	3.66	20.22	13.69
Java, Py, C++, C#	10.54	10.22	7.47	9.43	17.55	11.04
Java, JS, C++, C#	12.97	4.25	18.10	12.39	20.19	13.58
Py, JS, C++, C#	4.17	13.13	18.11	12.38	20.14	13.58
Java, Py, JS, C++, C#	13.24	13.35	18.26	12.51	20.33	15.54

All scores are in B-CC

The highest score on the test under each fine-tuning setting is marked in bold

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

References

- Ahmad WU, Chakraborty S, Ray B, Chang K (2021) Unified pre-training for program understanding and generation. In: NAACL-HLT. Association for Computational Linguistics, pp 2655–2668
- Alon U, Brody S, Levy O, Yahav E (2019) code2seq: generating sequences from structured representations of code. In: ICLR

- Bahdanau D, Cho K, Bengio Y (2015) Neural machine translation by jointly learning to align and translate. In: ICLR
- Banerjee S, Lavie A (2005) METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In: IEEValuation@ACL. Association for Computational Linguistics, pp 65–72
- Barnett JG, Gathuru CK, Soldano LS, McIntosh S (2016) The relationship between commit message detail and defect proneness in java projects on github. In: MSR. ACM, pp 496–499
- Buse RPL, Weimer W (2010) Automatically documenting program changes. In: ASE. ACM, pp 33–42
- Chen B, Cherry C (2014) A systematic comparison of smoothing techniques for sentence-level BLEU. In: WMT@ACL. The Association for Computer Linguistics, pp 362–367
- Clark K, Luong M, Le QV, Manning CD (2020) ELECTRA: pre-training text encoders as discriminators rather than generators. In: ICLR. OpenReview.net
- Conneau A, Wu S, Li H, Zettlemoyer L, Stoyanov V (2020) Emerging cross-lingual structure in pretrained language models. In: ACL. Association for Computational Linguistics, pp 6022–6034
- Cortes-Coy LF, Vásquez ML, Aponte J, Poshvanyk D (2014) On automatically generating commit messages via summarization of source code changes. In: SCAM. IEEE Computer Society, pp 275–284
- Devlin J, Chang M, Lee K, Toutanova K (2019) BERT: pre-training of deep bidirectional transformers for language understanding. In: NAACL-HLT (1). Association for computational linguistics, pp 4171–4186
- Dragan N, Collard ML, Maletic JI (2006) Reverse engineering method stereotypes. In: ICSM. IEEE Computer Society, pp 24–34
- Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) Codebert: a pre-trained model for programming and natural languages. In: EMNLP (Findings), findings of ACL, vol EMNLP 2020. Association for Computational Linguistics, pp 1536–1547
- Fluri B, Würsch M, Pinzger M, Gall HC (2007) Change distilling: tree differencing for fine-grained source code change extraction. IEEE Trans Software Eng 33(11):725–743
- Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M, Deng SK, Clement CB, Drain D, Sundaresan N, Yin J, Jiang D, Zhou M (2021) Graphcodebert: pre-training code representations with data flow. In: ICLR. OpenReview.net
- Hayes AF, Krippendorff K (2007) Answering the call for a standard reliability measure for coding data. Commun Methods Meas 1(1):77–89
- Hindle A, Germán DM, Godfrey MW, Holt RC (2009) Automatic classification of large changes into maintenance categories. In: ICPC. IEEE Computer Society, pp 30–39
- Hoang T, Kang HJ, Lo D, Lawall J (2020) Ce2vec: distributed representations of code changes. In: ICSE. ACM, pp 518–529
- Huang Y, Jia N, Zhou H, Chen X, Zheng Z, Tang M (2020) Learning human-written commit messages to document code changes. J Comput Sci Technol 35(6):1258–1277
- Jiang S (2019) Boosting neural commit message generation with code semantic analysis. In: ASE. IEEE, pp 1280–1282
- Jiang S, Armaly A, McMillan C (2017) Automatically generating commit messages from diffs using neural machine translation. In: ASE
- Jiang S, McMillan C (2017) Towards automatic generation of short summaries of commits. In: Proceedings of the 25th international conference on program comprehension, ICPC 2017, Buenos Aires, Argentina, May 22–23, 2017
- Kanade A, Maniatis P, Balakrishnan G, Shi K (2020) Pre-trained contextual embedding of source code. Preprint. <https://openreview.net/attachment?id=rygoURNYvS&name=original.pdf>
- Kendall MG (1945) The treatment of ties in ranking problems. Biometrika 33(3):239–251
- Koehn P, Huang H, Birch A, Callison-Burch C, Federico M, Bertoldi N, Cowan B, Shen W, Moran C, Zens R, Dyer C, Bojar O, Constantin A, Herbst E (2007) Moses: open source toolkit for statistical machine translation. In: ACL. The Association for Computational Linguistics
- Lample G, Conneau A, Ranzato M, Denoyer L, Jégou H (2018) Word translation without parallel data. In: ICLR (Poster). Openreview.net
- LeClair A, McMillan C (2019) Recommendations for datasets for source code summarization. In: NAACL-HLT (1). Association for Computational Linguistics, pp 3931–3937
- Lin C (2004) ROUGE: a package for automatic evaluation of summaries. In: Text summarization branches out, pp 74–81
- Lin C, Och FJ (2004) Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In: ACL. ACL, pp 605–612
- Liu C, Xia X, Lo D, Gao C, Yang X, Grundy JC (2022) Opportunities and challenges in code search tools. ACM Comput Surv 54(9):196:1–196:40
- Liu Q, Liu Z, Zhu H, Fan H, Du B, Qian Y (2019) Generating commit messages from diffs using pointer-generator network. In: MSR. IEEE/ACM, pp 299–309


- Liu S, Gao C, Chen S, Nie LY, Liu Y (2020) ATOM: commit message generation based on abstract syntax tree and hybrid ranking. *TSE* PP:1–1
- Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, Stoyanov V (2019) Roberta: a robustly optimized BERT pretraining approach. [arXiv:1907.11692](https://arxiv.org/abs/1907.11692)
- Liu Z, Xia X, Hassan AE, Lo D, Xing Z, Wang X (2018) Neural-machine-translation-based commit message generation: how far are we? In: ASE. ACM, pp 373–384
- Liu Z, Xia X, Treude C, Lo D, Li S (2019) Automatic generation of pull request descriptions. In: ASE. IEEE, pp 176–188
- Loyola P, Marrese-taylor E, Balazs JA, Matsuo Y, Satoh F (2018) Content aware source code change description generation. In: INLG. Association for Computational Linguistics, pp 119–128
- Loyola P, Marrese-Taylor E, Matsuo Y (2017) A neural architecture for generating natural language descriptions from source code changes. In: ACL (2). Association for Computational Linguistics, pp 287–292
- Luong T, Pham H, Manning CD (2015) Effective approaches to attention-based neural machine translation. In: EMNLP, pp 1412–1421
- Ma Q, Wei J, Bojar O, Graham Y (2019) Results of the WMT19 metrics shared task: segment-level and strong MT systems pose big challenges. In: WMT (2). Association for Computational Linguistics, pp 62–90
- Mogotsi IC, Manning CD, Raghavan P, Schütze H (2010) Introduction to information retrieval - Cambridge University Press, Cambridge, England, 2008, 482 pp, ISBN: 978-0-521-86571-5. *Inf Retr* 13(2):192–195
- Moreno L, Aponte J, Sridhara G, Marcus A, Pollock LL, Vijay-Shanker K (2013) Automatic generation of natural language summaries for java classes. In: ICPC. IEEE Computer Society, pp 23–32
- Moreno L, Marcus A (2012) Jstereocode: automatically identifying method and class stereotypes in java code. In: ASE. ACM, pp 358–361
- Myers JL, Well AD, Lorch RF Jr (2013) Research design and statistical analysis. Routledge
- Nie LY, Gao C, Zhong Z, Lam W, Liu Y, Xu Z (2021) Coregen: contextualized code representation learning for commit message generation. *Neurocomputing* 459:97–107
- Panichella S, Panichella A, Beller M, Zaidman A, Gall HC (2016) The impact of test case summaries on bug fixing performance: an empirical investigation. In: ICSE. ACM, pp 547–558
- Papineni K, Roukos S, Ward T, Zhu W (2002) Bleu: a method for automatic evaluation of machine translation. In: ACL. ACL, pp 311–318
- Petersen K, Vakkalanka S, Kuzniarz L (2015) Guidelines for conducting systematic mapping studies in software engineering: an update. *Inf Softw Technol* 64:1–18
- Ranzato M, Chopra S, Auli M, Zaremba W (2016) Sequence level training with recurrent neural networks. In: ICLR (Poster)
- Rebai S, Kessentini M, Alizadeh V, Sghaier OB, Kazman R (2020) Recommending refactorings via commit message analysis. *Inf Softw Technol* 126:106332
- See A, Liu PJ, Manning CD (2017) Get to the point: summarization with pointer-generator networks. In: ACL (1). Association for Computational Linguistics, pp 1073–1083
- Sennrich R, Firat O, Cho K, Birch A, Haddow B, Hirschler J, Junczys-Dowmunt M, Läubli S, Barone AVM, Mokry J, Nadejde M (2017) Nematus: a toolkit for neural machine translation. In: EACL (Software demonstrations). Association for Computational Linguistics, pp 65–68
- Shen J, Sun X, Li B, Yang H, Hu J (2016) On automatic summarization of what and why information in source code changes. In: COMPSAC. IEEE Computer Society, pp 103–112
- Sillito J, Murphy GC, Volder KD (2008) Asking and answering questions during a programming change task. *IEEE Trans Software Eng* 34(4):434–451
- Sorbo AD, Visaggio CA, Penta MD, Canfora G, Panichella S (2021) An nlp-based tool for software artifacts analysis. In: ICSME. IEEE, pp 569–573
- Swanson EB (1976) The dimensions of maintenance. In: ICSE. IEEE Computer Society, pp 492–497
- Tao W, Wang Y, Shi E, Du L, Han S, Zhang H, Zhang D, Zhang W (2021) On the evaluation of commit message generation models: an experimental study. In: ICSME. IEEE, pp 126–136
- van der Lee C, Gatt A, van Miltenburg E, Wubben S, Krahmer E (2019) Best practices for the human evaluation of automatically generated text. In: Proceedings of the 12th international conference on natural language generation. INLG
- Vásquez ML, Cortes-Coy LF, Aponte J, Poshyvanyk D (2015) Changelogscribe: a tool for automatically generating commit messages. In: ICSE (2). IEEE Computer Society, pp 709–712
- Wang B, Yan M, Liu Z, Xu L, Xia X, Zhang X, Yang D (2021a) Quality assurance for automated commit message generation. In: SANER. IEEE, pp 260–271
- Wang H, Xia X, Lo D, He Q, Wang X, Grundy J (2021b) Context-aware retrieval-based deep commit message generation. *ACM Trans Softw Eng Methodol* 30(4):56:1–56:30

- Wang X, Wang Y, Wan Y, Wang J, Zhou P, Li L, Wu H, Liu J (2022) CODE-MVP: learning to represent source code from multiple views with contrastive pre-training. In: NAACL-HLT. Association For computational Linguistics
- Wang Y, Wang W, Joty SR, Hoi SCH (2021) Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: EMNLP (1). Association for Computational Linguistics, pp 8696–8708
- Xu S, Yao Y, Xu F, Gu T, Tong H, Lu J (2019) Commit message generation for source code changes. In: IJCAI, pp 3975–3981. ijcai.org
- Xue N (2011) Steven bird, Evan Klein and Edward Loper. Natural Language Processing with Python. O'Reilly Media, Inc 2009. ISBN: 978-0-596-51649-9. Nat Lang Eng 17(3):419–424
- Yang Y, Xia X, Lo D, Grundy JC (2020) A survey on deep learning for software engineering. ACM Comput Surv

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Affiliations

Wei Tao¹ · Yanlin Wang²  · Ensheng Shi³ · Lun Du⁴ · Shi Han⁴ · Hongyu Zhang⁵ · Dongmei Zhang⁴ · Wenqiang Zhang¹

Wei Tao
wtao18@fudan.edu.cn

Ensheng Shi
s1530129650@stu.xjtu.edu.cn

Lun Du
lun.du@microsoft.com

Shi Han
shihan@microsoft.com

Hongyu Zhang
hongyu.zhang@newcastle.edu.au

Dongmei Zhang
dongmeiz@microsoft.com

Wenqiang Zhang
wqzhang@fudan.edu.cn

¹ Fudan University, Shanghai, China

² School of Software Engineering, Sun Yat-sen University, Zhuhai, China

³ Xi'an Jiaotong University, Xi'an, China

⁴ Microsoft Research Asia, Beijing, China

⁵ The University of Newcastle, Callaghan, Australia