SimLLM: Calculating Semantic Similarity in Code Summaries using a Large Language Model-Based Approach

XIN JIN, The Ohio State University, USA
ZHIQIANG LIN, The Ohio State University, USA

Code summaries are pivotal in software engineering, serving to improve code readability, maintainability, and collaboration. While recent advancements in Large Language Models (LLMs) have opened new avenues for automatic code summarization, existing metrics for evaluating summary quality, such as BLEU and BERTScore, have notable limitations. Specifically, these existing metrics either fail to capture the nuances of semantic meaning in summaries or are further limited in understanding domain-specific terminologies and expressions prevalent in code summaries. In this paper, we present SimLLM, a novel LLM-based approach designed to more precisely evaluate the semantic similarity of code summaries. Built upon an autoregressive LLM using a specialized pretraining task on permutated inputs and a pooling-based pairwise similarity measure, SimLLM overcomes the shortcomings of existing metrics. Our empirical evaluations demonstrate that SimLLM not only outperforms existing metrics but also shows a significantly high correlation with human ratings.

CCS Concepts: • **Software and its engineering → Documentation**; • **Computing methodologies → Natural language processing**.

Additional Key Words and Phrases: summary semantic similarity, large language models, automated code summarization

ACM Reference Format:
Xin Jin and Zhiqiang Lin. 2024. SimLLM: Calculating Semantic Similarity in Code Summaries using a Large Language Model-Based Approach. *Proc. ACM Softw. Eng.* 1, FSE, Article 62 (July 2024), 24 pages. https://doi.org/10.1145/3660769

1 INTRODUCTION

The importance of code summaries in software engineering cannot be overstated. These concise textual descriptions serve as essential guideposts for enhancing code readability, assisting in maintenance, and reducing debugging time [Feng et al. 2020; Wang et al. 2021c]. Given their central role, creating informative and precise code summaries is essential, yet it typically demands significant time and cognitive effort. The advances of Large Language Models (LLMs) have opened up promising avenues for automating the task of code summarization [Chakraborty et al. 2022; Feng et al. 2020; Wang et al. 2021c]. However, the increasing reliance on automated tools raises a pertinent question: *How can we rigorously evaluate the quality of machine-generated summaries to ensure they capture the intended code semantics and functionalities?*

To address the limitations of manual evaluations, particularly the labor-intensive nature and potential for human errors, automated metrics for evaluating code summary have gained traction [Haque et al. 2022; Roy et al. 2021]. Several metrics, including BLEU [Papineni et al. 2002],

Authors' addresses: Xin Jin, The Ohio State University, Columbus, USA, jin.967@osu.edu; Zhiqiang Lin, The Ohio State University, Columbus, USA, zlin@cse.ohio-state.edu.

ACM 2994-970X/2024/7-ART62
https://doi.org/10.1145/3660769

Reference Summary	add ***a*** new icon ***to*** ***the*** layout	combines ***two*** int ***lists***	Score Change	**Align to Human**
Candidate Summary	sets ***the*** doc font ***to*** ***a*** copy	combines ***2*** int ***arrays*** into single array		
BLEU Score	0.4286	0.2857	↓	✗
Human Rating	1 (out of 5)	4 (out of 5)	↑	-

(a) High BLEU on Different Semantics (b) Low BLEU on Same Semantics

Reference Summary	show all ***databases*** in hive	reload ***IDE*** to use ***jdk*** 1.7	Score Change	**Align to Human**
Candidate Summary	remove all characters from the ***database***	update ***java*** version by rebooting the ***editor***		
BERTScore	Precision=0.9210, Recall=0.9196, F1=0.9203	Precision=0.8438, Recall=0.8535, F1=0.8486	↓	✗
Human Rating	1 (out of 5)	3 (out of 5)	↑	-

(c) High BERTScore on Different Semantics (d) Low BERTScore on Domain-Specific Terms

Fig. 1. Motivating Examples. BLEU and BERTScore both show poor alignment to human ratings due to either the incapability of understanding summary semantics or the limited knowledge of domain-specific words.

ROUGE [Lin 2004], and METEOR [Banerjee and Lavie 2005], originally devised for machine translation, have been repurposed for code summary evaluations. These metrics quantify summary similarity based on n-gram co-occurrence. More recently, the research community has explored the applicability of contextual word embeddings, as in BERTScore [Zhang et al. 2019] and Sentence-BERT [Reimers and Gurevych 2019], to measure semantic similarities of code summaries [Haque et al. 2022; Roy et al. 2021].

Unfortunately, existing automated metrics often fall short in precisely capturing key semantics and domain-specific lexicon inherent in code summaries. The exact matching-based metrics, *e.g.*, BLEU, ROUGE, and METEOR, exhibit limitations when encountering semantic equivalences among syntactically distinct words and phrases, *e.g.*, "list" and "array", as shown in Figure 1b. These metrics also tend to assign high scores to meaningless but matched words, *e.g.*, "the" and "a" in Figure 1a. Text embedding-based metrics like BERTScore also face challenges, particularly when domain-specific terms such as "JDK" and "Java" or "IDE" and "editor" are involved, resulting in misleading similarity scores as depicted in Figure 1d. These limitations underscore the need for a more refined and specialized metric capable of capturing the precise semantics of code summaries.

In this paper, we propose SimLLM, a novel framework for calculating semantic similarity of code summaries, leveraging the semantic modeling capabilities of LLMs. As an evaluation framework for existing code summary systems, SimLLM takes as input their generated summaries along with the ground-truth summary references and subsequently computes similarity scores as the output. SimLLM employs a methodology based on permutation pretraining to rigorously assess code summary semantics and quantify their similarity, achieving a strong correlation with human ratings. Specifically, rather than resorting to conventional domain adaptation techniques, *e.g.*, supervised fine-tuning, our approach uniquely focuses on pretraining an autoregressive LLM on permuted inputs. This specialized training enables SimLLM to capture the nuanced semantics inherent in concise code summaries effectively, which are represented as semantic embeddings. We further incorporate a measurement component that computes pairwise semantic similarity between summaries by token embeddings and the aggregated summary embeddings. To handle the challenges posed by out-of-vocabulary terms, we introduce a subword tokenizer, ensuring that SimLLM can adapt to a diverse vocabulary found in real-world code summaries.

We have pretrained SimLLM on a code summary corpus containing over 2 million code summaries. To evaluate SimLLM, we construct a new dataset, systematically annotated by four experienced programmers. This dataset covers five programming languages, *i.e.*, Python, JavaScript, Ruby, Go, and PHP, with summaries generated by State-of-the-art LLMs, *i.e.*, ChatGPT [OpenAI 2022], Magicoder [Wei et al. 2023], Llama [Touvron et al. 2023], and CodeT5 [Wang et al. 2021c]. In addition, we also include two public datasets in our evaluations, which consist of Java methods and summaries produced by two code summarization models. We employ Spearman's rank correlation coefficient [Kumar and Abirami 2018] as our evaluation metric. Our evaluation results demonstrate that SimLLM outperforms all the baseline metrics in both datasets with generalizability and robustness to dataset change, achieving an improvement of up to 258.4% in correlation coefficient compared to BLEU. Our findings also reveal that SimLLM's mean pooling- and pairwise cosine-based summary encoding technique offers the best correlation with human evaluations. Moreover, we also identify the strong correlation between SimLLM and some of our baselines (*e.g.*, BERTScore). Our input permutation pretraining reinforces SimLLM's effectiveness as a superior tool for measuring code summary similarity. SimLLM also demonstrates strong robustness to the different software engineering practices (*e.g.*, programming language variance) and code summarization models.

SimLLM brings forth a substantial shift in automated code summarization, offering effective metrics for precise summary semantic quality assessment. For LLMs like ChatGPT and Code Llama [Roziere et al. 2023], SimLLM can facilitate reinforcement learning from automated metric feedback loop [Griffith et al. 2013] that empowers the models to precisely learn code semantics. SimLLM's design can enable continuous performance refinement, leading to the generation of high-quality summaries closely aligned with human judgment. Furthermore, the impact of SimLLM extends beyond improving code summarization systems, offering potential benefits to other tasks that require summary semantic modeling, such as code summary categorization [Shi et al. 2022], due to SimLLM's advanced semantics comprehension capacities. In short, SimLLM paves the way for both next-generation code summarization systems and broader summary-specific applications.

Contributions. Our paper makes the following contributions:

- We propose a novel LLM-based approach, SimLLM, tailored to automatically generate precise summary semantic similarity scores.
- We employ an input permutation-based pretraining paradigm to learn concise code summary semantics, and meanwhile construct a new evaluation dataset that spans various programming languages and latest code summarization models.
- We advance the state-of-the-art in calculating summary similarity by surpassing all prevalent baseline metrics, demonstrating SimLLM's generalizability and robustness, along with additional software engineering insights and impact.

2 RELATED WORK AND MOTIVATIONS

2.1 Related Work

Code Summarization. Code summarization is to automatically generate concise, human-readable summaries representing functionality and intent of code segments [Sharma et al. 2021]. Numerous works [Ahmad et al. 2021; Feng et al. 2020; Guo et al. 2020; Wang et al. 2022, 2021c] have been proposed, which can be classified into three main categories. First, the early efforts [Haiduc et al. 2010; Wong et al. 2015] viewed the task as an information retrieval problem, leveraging keyword extraction from source code to compose term-based summaries. Second, more recent works [Liu et al. 2022; Wang et al. 2021c] have utilized neural machine translation frameworks for code summarization, such as CodeT5 [Wang et al. 2021c], which is based on the powerful T5 model and enhances code features like identifiers. Third, some studies [Guo et al. 2020; Son et al. 2022] capture program-specific features such as control flow and data dependency to enrich code representations. For example, GraphCodeBERT [Guo et al. 2020] focuses on variable-level data flow relationships rather than syntactic structures. Recently, there has been research attention on leveraging generative LLMs to facilitate code summarization tasks [Das et al. 2023; Jin et al. 2023; Sun et al. 2023]. Sun et al. [Sun et al. 2023] evaluate ChatGPT on a large code summarization dataset. Besides source code, Jin et al. [Jin et al. 2023] extend this research area by benchmarking ChatGPT, GPT-4, and Llama models on binary code summarization.

Automated Code Summary Assessment. Automated metrics evaluate the effectiveness of code summarization models by quantifying the textual similarity between generated and reference summaries [Roy et al. 2021]. These metrics have been inspired by, and often borrowed from, the field of neural machine translation. BLEU [Papineni et al. 2002], ROUGE [Lin 2004], and METEOR [Banerjee and Lavie 2005] are extensively employed for this purpose. Each of these metrics utilizes an exact-matching method to compute word overlaps in the generated and reference summaries [Rankel et al. 2013]. Recently, a shift towards embedding-based metrics, such as BERTScore [Zhang et al. 2019], has been explored [Haque et al. 2022; Roy et al. 2021]. However, both classes of metrics—exact-matching and embedding-based—have their own sets of limitations. While exact-matching metrics focus on syntactical differences, they often fail to capture the semantics of code summaries. In contrast, embedding-based metrics show better correlations to human ratings [Haque et al. 2022] by their contextualized representations generated from well-trained models or pre-computed vectors. However, embedding-based metrics struggle to account for the domain-specific expressions, which are crucial for precisely evaluating code summaries.

Semantic Textual Embeddings. In natural language processing, embeddings serve as numerical text representations within high-dimensional vector space, facilitating computational efficiency and semantic modeling [Levy and Goldberg 2014]. These embeddings operate at different granularities, *e.g.*, words and sentences. At the word level, distributed word embeddings like Word2Vec [Mikolov et al. 2013] and GloVe [Pennington et al. 2014] gained widespread research attention. More recent advancements, however, have come from transformer-based models, such as BERT [Devlin et al. 2018] and GPT [Radford et al. 2019], which produce dynamic embeddings. At the sentence level, SentenceBERT [Reimers and Gurevych 2019] employs BERT encoder, siamese, and triplet networks to produce sentence semantic embeddings. Recent work has explored learning universal sentence representations for various downstream tasks [Conneau and Kiela 2018]. Despite advances, generating effective sentence embeddings remains an open research question [Wu et al. 2018].

2.2 Motivations

Limitations of Exact Matching-based Metrics. Exact matching-based metrics, such as BLEU, METEOR, and ROUGE, struggle to capture semantics present in code summaries. For instance, "array" and "list" both refer to a data structure storing a list of elements and they are keywords defined in Java and Python, respectively. In code summaries, "array" and "list" may be used interchangeably. However, they are treated as distinct by BLEU, yielding in a low similarity score, as illustrated in Figure 1b. Exact matching-based metrics also lack the capability to equate numerals with their written forms, such as "two" with "2" in Figure 1b. Furthermore, it is important to note that not all words within a sentence carry equal semantic significance. English contains numerous words that contribute minimally to sentence semantics, primarily serving as conjunctions. These words are commonly referred to as "stop words" [Silva and Ribeiro 2003]. As depicted in Figure 1a, examples of stop words include "a", "to", and "the". However, BLEU treats such words equally with other words in the summaries, resulting in a high similarity score. Figure 1a showcases BLEU's incorrect measurement based on syntactical same words and the inability to effectively measure the distinct core semantics in the candidate and reference summary pair. These problems, shown in Figure 1a and Figure 1b, also exist in other exact matching-based metrics, *e.g.*, METEOR and ROUGE.

Shortcomings of Embedding-based Metrics. While embedding-based metrics like BERTScore aim to resolve the shortcomings of exact matching by leveraging pretrained language models [Zhang et al. 2019], we found that they have their own limitations. First, these metrics can effectively capture simple semantic variations, such as "database" and "databases" in Figure 1c. However, embedding-based metrics often struggle with domain-specific vocabulary and terminology such as acronyms and synonymous expressions commonly found in code summaries. For example, "jdk" is the abbreviation of "Java Development Kit", which has a close semantic correlation with "java". Similarly, "IDE" stands for integrated development environment, which commonly serves as a code editor for developers. However, BERTScore fails to recognize the semantic relationship between "jdk" and "java" and between "IDE" and "editor", then assigns low similarity scores, as shown in Figure 1d. Similar to BERTScore, embedding-based metrics, such as InferSent [Conneau et al. 2017] and SentenceBERT [Reimers and Gurevych 2019], are typically trained on generic text corpora and lack the specialized semantic understanding capacities and vocabularies necessary for evaluating code summaries accurately [Conneau et al. 2017; Haque et al. 2022].

3 OVERVIEW

Problem Definition. Consider a given code context, denoted by $x \in \mathcal{X}$, which could represent a variety of code components, such as a source code function. A code summarization system, $\mathcal{M} : \mathcal{X} \rightarrow \mathcal{Y}$, generates a natural language summary $y \in \mathcal{Y}$, based on the semantic essence of the input context x. To evaluate the quality of y, we contrast it with a reference summary $\hat{y} \in \mathcal{Y}$ by using a measurement metric $\Gamma : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$. This metric calculates $\Gamma(y, \hat{y})$ to measure the semantic similarity between the machine-generated summaries and reference summaries.

A higher value of $\Gamma(y, \hat{y})$ suggests that the machine-generated summary candidate y is closely aligned with the reference summary $\hat{y}$. More importantly, the relative ranking output by Γ is often more significant than the absolute value of $\Gamma(y, \hat{y})$. In essence, the key concern is how Γ ranks various summary candidates relative to the reference summary. Ideally, if one candidate y_1 is both functionally equivalent to $\hat{y}$ and also preferred by human evaluators over another candidate y_2, a well-designed metric should satisfy $\Gamma(y_1, \hat{y}) > \Gamma(y_2, \hat{y})$.

As an example, in Figure 1, we present the scores of BLEU, BERTScore, and human ratings. Compared to summaries in Figure 1a, the human annotator assigns a higher score to summaries in Figure 1b due to the better essential semantic similarity, while BLEU assigns a lower score. The same issue appears in Figure 1c and Figure 1d for BERTScore. Based on our definition, existing exact matching-based and embedding-based metrics, *e.g.*, BLEU and BERTScore, fail to identify semantic changes, resulting in their poor alignment with human judgment.

Objectives. The overall objective of this paper is to develop a robust framework that can measure the semantic similarity of code summaries, which is closely aligned with human judgement. For this, we draw inspiration from the broader natural language processing (NLP) field, which has successfully built large language models (LLMs) for specialized domains [Wang et al. 2021b]. However, a major hurdle remains: *the challenge of effectively adapting these LLMs to our specific use-case.*

Challenges. Adapting LLMs to domain-specific tasks can be resolved by two common approaches, *i.e.*, fine-tuning existing pretrained LLMs and continuous pretraining [Wang et al. 2021b]. Fine-tuning pretrained models like BERT on *labeled* domain-specific data is a widely-used adaptation technique [Devlin et al. 2018]. However, the unavailability of extensive, publicly accessible, and human-annotated code summary similarity datasets, especially those with sufficient diversity, poses a substantial challenge. Additionally, while continuous pretraining offers an alternative adaptation strategy, this approach risks deteriorating the original model's capabilities [Wang et al. 2021a,b]. Notably, both fine-tuning and continuous pretraining inherit the original model's tokenizer and

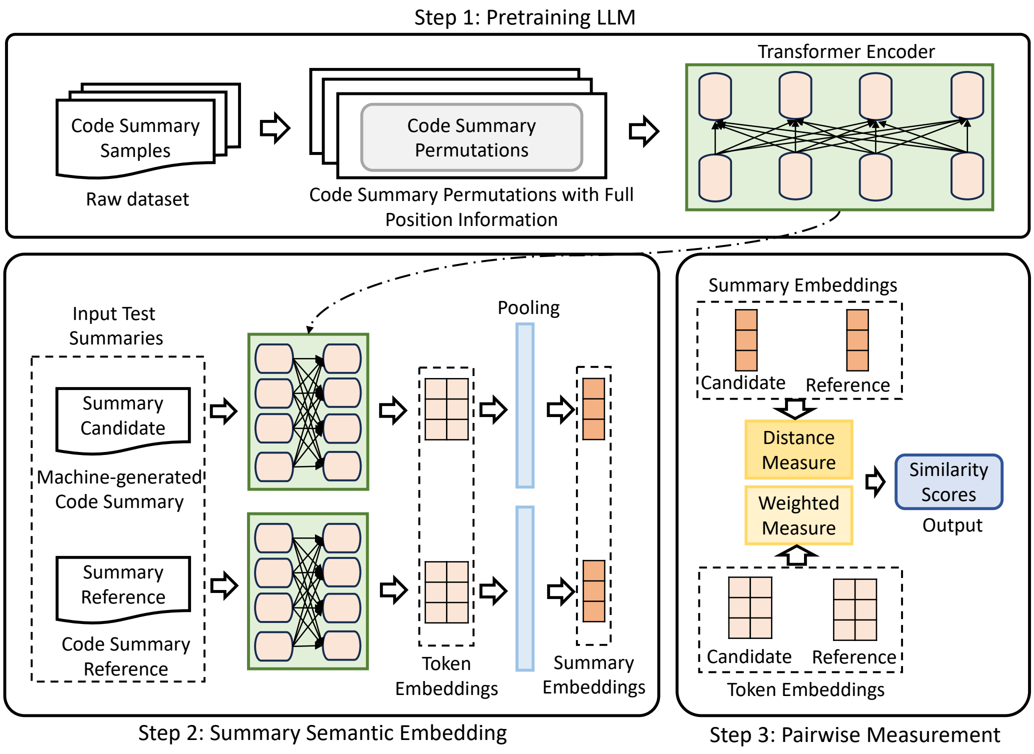


Fig. 2. SimLLM Workflow

vocabulary, thereby failing to adequately address the out-of-vocabulary issue that is especially pertinent to domain-specific words in code summaries.

Insights. Code summaries often comprise words in various morphological forms, including domain-specific acronyms, abbreviations, and occasionally, misspelled words. This characteristic makes it challenging to directly use or adapt existing pretrained LLMs for code summary measurement. A new tokenizer specialized in code summaries should be built to address out-of-vocabulary issues. In addition to morphological terms, common words can undergo semantic shifts within the context of code summarization. For instance, the term “editor”, which commonly denotes a person responsible for reviewing in general English, takes on a different meaning when used in the context of code summaries, where it mostly refers to a programming toolkit. Hence, it becomes imperative to perform summary-specific contextualization and modeling of words and morphological terms. Furthermore, code summaries tend to be concise compared to general corpora, succinctly expressing semantics. As a result, simply training LLMs, *e.g.*, BERT, on code summary data may not yield optimal information utilization (see §4.1 for detailed analysis). Additionally, less significance should be assigned to meaningless words, such as stop words, in summary semantic representations, compared to words that carry substantial semantic meaning.

SimLLM Overview. Based on these insights, we propose SimLLM, to advance automated code summary similarity calculation, as illustrated in Figure 2. In step 1, SimLLM’s LLM model is autoregressively pretrained from scratch on permutations of tokenized code summaries with full positional information, leveraging a transformer encoder architecture (§4.1). This pretraining approach combines autoregressive language modeling and input permutation. After pretraining, steps 2 and 3 generate semantic similarity between summary candidates, produced by existing code summarization systems, and ground-truth summary references. Specifically, in step 2, SimLLM generates numerical token embeddings for input code summaries. These embeddings are then aggregated into summary-level representations via pooling methods (§4.2). In step 3, pairwise similarity is calculated using semantic distance functions (§4.3), with adjustments like weighted token embeddings. Additionally, a custom subword-level tokenizer is designed to mitigate out-of-vocabulary issues (see §5.1). SimLLM outputs scores ranging from -1 to 1, with positive values indicating the degree of similarity and negative values reflecting the extent of dissimilarity.

Assumptions. To use SimLLM, we assume that the input code summaries are in natural languages. Importantly, SimLLM is designed to be independent of the summary's language, offering versatility across different linguistic contexts. Despite its language-agnostic capabilities, our pretraining and evaluations primarily engage with English-language summary corpora. This focus is driven by English's widespread use and the relative ease of accessing English-based datasets.

Formal Hypotheses and Statistical Tests. To robustly assess SimLLM, we propose and test the null hypothesis (H_0) that *there is no association between human-generated assessments and SimLLM-produced similarity scores*. In other words, it suggests that any observed correlation in the ranks of these assessments and similarity scores can be attributed to chance rather than a real, underlying relationship. This hypothesis is tested against the alternative hypothesis (H_1), which proposes that there is a non-zero correlation between the variables. If the p-values [Zwillinger and Kokoska 1999] obtained from our statistical tests are less than the chosen significance level (0.001 in this paper), the null hypothesis is rejected, indicating that there is evidence of a statistically significant alignment between the human-generated assessments and SimLLM's similarity scores.

4 DETAILED DESIGN

4.1 Pretraining

Unsupervised language models have proven to be remarkably effective in many real-world tasks [Devlin et al. 2018; Jin and Wang 2023; Su et al. 2024; Yang et al. 2019]. As the frontrunners of these models, the autoregressive ones effectively estimate the probability distribution of text corpora [Yu et al. 2019]. Specifically, an autoregressive model decomposes the probability of a text sequence $S = (w_1, w_2, \ldots, w_n)$ into either a forward or backward conditional product, denoted by $p(S) = \prod_{i=1}^{n} p(w_i|w_1, w_2, \ldots, w_{i-1})$ and $p(S) = \prod_{i=1}^{n} p(w_i|w_{i+1}, w_{i+2}, \ldots, w_n)$, respectively. However, such a design can only capture unidirectional context. Addressing this limitation, autoencoder-based approaches like BERT deviate from explicit density estimation to focus on data reconstruction [Devlin et al. 2018]. BERT employs masked language modeling in which a portion of the input tokens is randomly selected and replaced with special `[MASK]` tokens. Nevertheless, BERT's architecture makes the incorrect assumption that masked tokens are statistically independent [Song et al. 2020; Yang et al. 2019], thereby failing to model the dependency of these tokens. To rectify this, permutation language modeling, as exemplified by XLNet, introduces a more nuanced understanding of conditional distributions over permuted inputs [Yang et al. 2019]. In particular, XLNet computes autoregressive probabilities over all permutations, thereby efficiently modeling the bidirectional conditional probabilities and token dependency. However, even XLNet's strategy has limitations; it cannot fully capture the context of an *entire* sentence, losing complete positional information, during its autoregressive pre-training [Song et al. 2020].

Considering our task of calculating semantic similarity in code summaries, it's important to note that such summaries are usually shorter and denser in semantic content than typical natural language inputs. As a result, any loss of information, such as the lack of masked token dependency in BERT or the absence of complete positional information in XLNet, could significantly degrade SimLLM's ability to model semantics accurately. Inspired by MPNet [Song et al. 2020], SimLLM addresses these issues by pretraining autoregressive large language models on permuted inputs while preserving full positional information. This design choice offers two main advantages: firstly, it improves input utilization by 8.8% compared to approaches that only use masked tokens [Song et al. 2020]; secondly, it merges the strengths of both masked and permutation-based language models, thereby equipping SimLLM with the capability to effectively learn concise and domain-specific function summaries. In the following, we present our detailed methodology, including position-enhanced input permutation and pretraining objectives.

Position-enhanced Input Permutation. Let's consider an original sequence of tokens $S = \{w_1, w_2, \dots, w_n\}$. Following the same permutation technique as XLNet, we generate a randomly permuted order $o = \{o_1, o_2, ..., o_n\}$ to rearrange the sequence into $S_o = \{w_{o_1}, w_{o_2}, \dots, w_{o_n}\}$. If we directly pretrain autoregressive language models on such a permutated sequence, it results in the same problem of incomplete positional information as in XLNet. Following the permutation augmentation approach [Song et al. 2020], we tackle this issue by introducing unpredictable tokens that carry the same positional information as the ones that are missing.

Specifically, we partition S_o into two distinct subsequences: a non-target subsequence $S_{o_{\leq c}}$ and a target subsequence $S_{o_{>c}}$. Tokens in the non-target subsequence are not predicted during training but serve as the context for predicting the tokens in the target subsequence. In a standard implementation of XLNet, the conditional probability for predicting each token in the target subsequence is given by $p(S_o) = \prod_{i=c+1}^{n} p(w_{o_i} | w_{o_{<i}})$. However, this approach constrains each target token $w_{o_{>c}}$ to only consider its preceding tokens, lacking full positional awareness. To address this limitation, we construct an extended sequence $S'_o = \{w_{o_{\leq c}}, d_{o_{>c}}, w_{o_{>c}}\}$, where $d_{o_{>c}}$ denotes a series of dummy tokens that have the same length and positional information as $w_{o_{>c}}$. $d_{o_{>c}}$ compensates the missing positional information and its tokens are unpredictable during training, similar to $w_{o_{\leq c}}$. This design choice empowers SimLLM to access the complete positional information within the input sequence, as opposed to the partial positional information observed in XLNet. To clarify the process, let's consider an example: Suppose $S = \{w_1, w_2, w_3, w_4\}$ is permuted with the order $o = \{1, 3, 2, 4\}$, resulting in $S_o = \{w_1, w_3, w_2, w_4\}$. With $c = 2$, we divide S_o into $S_{o_{\leq c}} = \{w_1, w_3\}$ and $S_{o_{>c}} = \{w_2, w_4\}$. We then insert dummy tokens, yielding $S'_o = \{w_1, w_3, \text{[DUMMY]}, \text{[DUMMY]}, w_2, w_4\}$ with the position sequence $P'_o = \{p_1, p_3, p_2, p_4, p_2, p_4\}$. Here, the prediction of w_2 and w_4 can leverage the complete position context $\{p_1, p_3, p_2, p_4\}$, *e.g.*, the conditional probability $p(w_2 | w_1, w_3, \text{[DUMMY]}, \text{[DUMMY]})$. It is worth noting that, in this input permutation design, the sequence after permutation can appear unnatural to human developers. But SimLLM is based on the transformer model and its self-attention mechanism is not sensitive to the absolute input order of those tokens, when each token is associated with its correct position [Song et al. 2020]. In our design, we strictly followed this token-position association rule to align tokens and their positions. Therefore, SimLLM can model contextualized token semantics in the autoregressive training.

Pretraining Objective. We train SimLLM to maximize the following likelihood estimation, which is defined over all permutations O_n of a sequence of length n:

$$\max_{\theta} \mathbb{E}_{o \sim O_n} \left[\sum_{i=c+1}^{n} log \left(p_\theta \left(w_{o_i} | w_{o_{<i}}, d_{o_{>i}} \right) \right) \right] \tag{1}$$

$$p_\theta \left(w_{o_i} | w_{o_{<i}}, m_{o_{>c}} \right) = \frac{exp \left(h_\theta \left(w_{o_{<i}}, d_{o_{>i}}, o_i \right)^\top e(w_{o_i}) \right)}{\sum_{w'} exp \left(h_\theta \left(w_{o_{<i}}, d_{o_{>i}}, o_i \right)^\top e(w') \right)} \tag{2}$$

Here, $e(w)$ is the embedding of w, o_i is i-th token's positional index in permutated sequence, and $h_\theta(\cdot)$ denotes the model-generated hidden representations, which incorporate complete positional information. In this work, such representations are generated using two-stream self-attention layers [Yang et al. 2019] for efficient autoregressive prediction.

4.2 Summary Semantics Embedding

Token Semantics Embedding. After pretraining our model on a substantial code summary corpus (see implementation details of pretraining in §5.2), we use it to encode semantics of test summaries, *i.e.*, summary candidates and summary references. Specifically, unlike the pretraining stage where input sequences are permuted and dummy tokens are inserted, during inference we only tokenize the code summaries without permutation (see tokenizer details in §5.1). Our model, therefore, produces high-quality contextual embeddings for individual tokens without requiring artificial tokens like dummy tokens. In SimLLM, we take the hidden states from its last layer as token semantic embeddings. Our token semantic embeddings present three advantages over traditional static word2vec [Mikolov et al. 2013; Pennington et al. 2014] and dynamic embeddings [Devlin et al. 2018]. First, they generate context-sensitive embeddings for identical words appearing in differing sentences. Second, tailored for code summaries, our model outperforms general-purpose embeddings trained on datasets like Common Crawl [Luccioni and Viviano 2021]. Third, our model adeptly manages summary-specific terminologies and abbreviations, effectively mitigating out-of-vocabulary issues commonly encountered in other models.

Summary Semantics Embedding. Upon obtaining the token embeddings, our objective shifts to quantifying the semantics of the whole summaries. Achieving this goal presents a multitude of challenges. First of all, the code summaries exhibit variability in sequence lengths, resulting in token embedding matrices of disparate dimensions. This length diversity complicates computational efforts to compare one summary's embeddings with another. To address these challenges, we generate semantic embeddings for entire summaries by pooling the token embedding matrices, and employ three pooling methods to generate fixed-length summary embeddings:

- **CLS Pooling**: [CLS] token is adept at capturing sentence-level semantics, introduced by BERT [Devlin et al. 2018]. We prepend [CLS] tokens to input summaries and generate [CLS] embeddings.
- **Mean Pooling**: Mathematically defined as, given token embedding $x \in \mathcal{X}^{\mathbb{R}}$ and the number of tokens in the summary $|\mathcal{X}^{\mathbb{R}}|$:

$$emb_{mean} = \frac{1}{|\mathcal{X}^{\mathbb{R}}|} \cdot \sum_{x \in \mathcal{X}^{\mathbb{R}}} x \tag{3}$$

- **Maximum Pooling**: Expressed formally as:

$$emb_{max} = \max_{x \in \mathcal{X}^{\mathbb{R}}} x \tag{4}$$

These pooling methods serve the crucial purpose of producing fixed-length representations of sentences, an essential requirement for our downstream task of similarity measurement. Utilizing fixed-length representations is computationally more efficient. Additionally, this approach helps mitigate the impact of noisy tokens within the summaries. To dive deeper into the advantages of these pooling methods, we additionally investigate a measurement approach by directly performing calculations on the token embeddings for comparison, which is further illustrated in §4.3.

4.3 Pairwise Summary Similarity Measurement

Recall the primary objective of our work (see §3) is to compute a similarity score, denoted by $\Gamma(y, \hat{y})$, between summary pairs $\{y, \hat{y}\}$, in which $\Gamma : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$ is the pairwise similarity

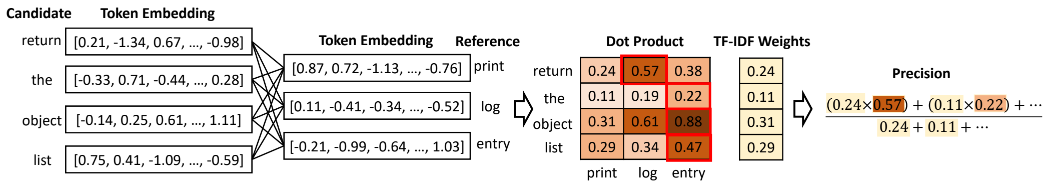

Candidate
Token Embedding
return [0.21, -1.34, 0.67, ..., -0.98]
the [-0.33, 0.71, -0.44, ..., 0.28]
object [-0.14, 0.25, 0.61, ..., 1.11]
list [0.75, 0.41, -1.09, ..., -0.59]
Token Embedding
Reference
[0.87, 0.72, -1.13, ..., -0.76] print
[0.11, -0.41, -0.34, ..., -0.52] log
[-0.21, -0.99, -0.64, ..., 1.03] entry
Dot Product
TF-IDF Weights
return 0.24 0.57 0.38
the 0.11 0.19 0.22
object 0.31 0.61 0.88
list 0.29 0.34 0.47
print log entry
0.24 0.11 0.31 0.29
Precision
(0.24×0.57) + (0.11×0.22) + ⋯
0.24 + 0.11 + ⋯

Fig. 3. Illustration of calculating precision for summary similarity based on token embeddings

measurement function. We examine two types of measurement functions: those based on summary-level embeddings and those directly on token-level embeddings.

Measurement on Summary Embeddings. To capture semantic similarity, we compute the distance between summary pairs in the embedding space. Specifically, we utilize cosine similarity [El-Kassas et al. 2021], a commonly used metric in natural language processing, as outlined below:

$$Sim(emb_y, emb_{\hat{y}}) = \frac{emb_y \cdot emb_{\hat{y}}}{\|emb_y\| \times \|emb_{\hat{y}}\|} \tag{5}$$

Cosine similarity transforms the problem of distance measurement into an angle comparison in the vector space. Additionally, its scale-invariant property makes it particularly suited for our task where summary lengths can vary but may still have similar meanings.

We also use inverse Euclidean distance [El-Kassas et al. 2021] as another summary-level similarity metric:

$$Sim(emb_y, emb_{\hat{y}}) = \frac{1}{1 + \sqrt{\sum_{i=1}^{|emb|}(emb_y^{(i)} - emb_{\hat{y}}^{(i)})^2}} \tag{6}$$

Measurement on Token Embeddings. As mentioned in §4.2, we also investigate token-level similarity measures for summary pairs. Inspired by greedy search methods [Zhang et al. 2019; Zhou et al. 2023], we utilize precision, recall, and F1 score to calculate weighted summary similarity scores by their semantic significance. Given a candidate summary y, its reference $\hat{y}$, and their token embeddings x_y and $x_{\hat{y}}$ generated by SimLLM, precision (P), recall (R), and F1 score (F1) are calculated as follows:

$$P = \frac{\sum_{w_i \in y} \mu(w_i) \times \max_{\hat{w}_j \in \hat{y}} x_y^{(i)} x_{\hat{y}}^{(j)\top}}{N \times \sum_{w_i \in y} \mu(w_i)}, \quad R = \frac{\sum_{\hat{w}_j \in \hat{y}} \mu(\hat{w}_j) \times \max_{w_i \in y} x_y^{(i)} x_{\hat{y}}^{(j)\top}}{N \times \sum_{\hat{w}_j \in \hat{y}} \mu(\hat{w}_j)}, \quad \text{F1} = 2 \times \frac{P \times R}{P + R} \tag{7}$$

where $x_y^{(i)} x_{\hat{y}}^{(j)\top}$ is the dot product. $\mu(w)$ is the TF-IDF [Robertson 2004] weight of token w, computed as:

$$\mu(w) = \text{tf}(w) \times \log\left(\frac{N}{DF_w}\right) \tag{8}$$

$tf(w)$ denotes the frequency of w in the candidate summary. N is the total number of reference summaries, and DF_w is the count of reference summaries that include the token w. This weighting scheme for tokens stems from our empirical observation that different words contribute variably to the semantics of the summary, as elaborated in §2.2. To illustrate the procedure for computing similarity scores using token embeddings, we offer a detailed example for calculating precision in Figure 3. Specifically, given SimLLM-generated token embeddings of summary pairs $\{\hat{y}, y\}$, we compute dot products and then calculate precision scores using a weighted summation, accounting for TF-IDF weights. In this example, the stop word “the” is automatically assigned a diminished weight relative to other tokens. The methodology for computing recall follows a similar procedure.

5 IMPLEMENTATION

We have implemented SimLLM upon PyTorch [Paszke et al. 2019], Scikit-learn [Pedregosa et al. 2011], and SciPy [Virtanen et al. 2020], and CUDA 11.6 for GPU acceleration. Our pretraining module is built upon Fairseq [Ott et al. 2019], roBERTa [Liu et al. 2019], and MPNet [Song et al. 2020]. Our computational environment for both the pretraining and inference phases comprises a Dell server, furnished with a 48-core AMD EPYC 7643 processor clocked at 2.3 GHz, running the RHEL 8.6 operating system, equipped with 921 GB of RAM, and 4 NVIDIA A100 GPUs, each with 80 GB of VRAM. Overall, pretraining SimLLM requires 53 GPU hours. After pretraining, SimLLM’s inference time per code summary pair is 0.0089 seconds on a single CPU and 0.0003 seconds on a GPU, respectively.

5.1 Preprocessing and Tokenization

Preprocessing. We noticed that code summaries contain textual noise, posing the risk of SimLLM learning noisy language features and hindering its robustness and generalizability. To address this, we employ a four-step preprocessing method. Initially, the summary text is extracted from comment blocks. Next, we standardize the summaries by eliminating extraneous spaces, normalizing decimal separators, and streamlining punctuation marks. Afterwards, characters outside the printable ASCII range and control characters are replaced with spaces as well. Lastly, we convert all the English words in the summaries to lowercase.

Tokenization. Tokenization is a pivotal step in addressing out-of-vocabulary issues for SimLLM, transforming unstructured summary text into a machine-friendly structured format. The common tokenization approaches are at either character-level [Mielke et al. 2021] or word-level [Jin et al. 2022], presenting limitations. Character-level tokenization segment summaries into individual characters, losing original word semantics. Word-level tokenization fails in handling code summaries that feature morphologically complex or infrequent terms, affecting both training efficiency and model comprehension during inference. To avoid these issues, we opt for the byte-pair encoding (BPE) algorithm [Sennrich et al. 2015], which operates at the subword level. It adopts a hybrid character- and word-level tokenization, significantly compressing the vocabulary size and enabling efficient computation. For vocabulary construction, our BPE tokenizer statistically identifies and amalgamates frequently occurring byte pairs in an iterative manner until a predefined vocabulary size is reached. We implemented this tokenization module upon fastBPE [Ott and et. al 2023], setting the vocabulary size to 40K for our pretraining corpus (see corpus detailed in §5.2). The vocabulary and tokenizer are constructed once during pretraining and directly used in inference. When using our BPE tokenizer, we found that *it never introduces any out-of-vocabulary tokens* in our subsequent evaluations.

5.2 Pretraining

Pretraining Dataset. Our pretraining of SimLLM leverages the CodeSearchNet dataset [Husain et al. 2019], which is widely used in source code summarization research [Feng et al. 2020; Wang et al. 2021c]. This dataset contains a corpus of over 2 million comment-code pairs curated from open-source repositories, spaning six distinct programming languages: Python, Javascript, Ruby, Go, Java, and PHP. We extracted function summaries from 2,070,536 unique functions, serving as the basis for our pretraining dataset. Due to the resource-intensive and computationally expansive nature of pretraining, conducting hyperparameter tuning and evaluation with a separate validation set is unfeasible. Instead, we split the dataset into training and test sets at a 9:1 ratio, utilizing the test set primarily for performance monitoring.

Model Architecture. SimLLM inherits its architectural foundation from the well-known roBERTa model [Liu et al. 2019]. The model consists of 12 transformer layers, each consists of 12 self-attention heads. We configure the embedding dimension to be 768 and the hidden layer size of the feed-forward networks within the self-attention layers to be 3072. To mitigate the vanishing gradient problem and enhance gradient flow, we incorporate the GeLU activation function. Additionally, we set a dropout rate of 0.1. In total, SimLLM encapsulates approximately 110 million parameters.

Pretraining Setup. We set the input length as 512 and the per-update batch size as 32. Employing a gradient accumulation strategy with an update frequency of 16, our effective batch size becomes $32 \times 16 = 512$, aligning with established best practices [You et al. 2019]. For this effective batch size, we set the learning rate as 10^{-4}. We also adopt a warmup mechanism, commencing at 10^{-7} and escalating to the target rate over the first 10,000 updates. We employ the Adam optimizer, configured with hyperparameters $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 10^{-6}$, and apply a weight decay of 10^{-2}. The training phase caps at 50,000 updates, completing in approximately 53 hours on a single A100 GPU.

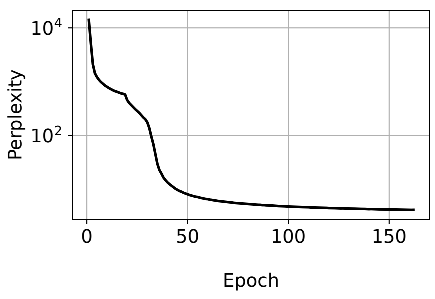


Fig. 4. Test Set Perplexity Evolution Across Epochs.

Dataset Shuffling. To induce variance during pretraining, we shuffle the dataset, thereby randomizing the sample orders for each epoch. This strategy serves a twofold purpose: it not only obstructs the model's tendency to memorize specific sequences, thereby prohibiting overfitting, but also aids in the efficient convergence of the model by avoiding fixed, repetitive sequences of examples.

Pretraining Metric Monitoring. For pretraining assessment, we use the prevalent metric perplexity [Chen et al. 1998] to measure the model's uncertainty regarding its predictions. We validate the efficacy of SimLLM's pretraining by the perplexity of our test set. As shown in Figure 4, SimLLM exhibits a marked reduction in perplexity, converging after 100 epochs. This signifies that SimLLM has likely developed a robust comprehension of code summary semantics. It is worth noting that we preserve the best model, determined by testing perplexity, as a precaution against overfitting.

6 EVALUATIONS

Our evaluation is structured around the following research questions:

- **RQ1:** How well do SimLLM generated similarity scores correlate with human ratings, compared to existing summary similarity metrics? Is SimLLM robust on different datasets?
- **RQ2**: Which pairwise measurement method of SimLLM (§4.3) exhibits superior performance?
- **RQ3:** How does SimLLM correlate with other metrics, *i.e.*, existing summary similarity metrics?
- **RQ4:** How does SimLLM's pretraining specifically enhance its performance?
- **RQ5:** How do software engineering practices (*e.g.*, programming language-specific variance) and code summary generation models impact SimLLM's performance?

6.1 Evaluation Setup

Gold Standard. Human-generated assessments remain the gold standard for evaluating code summaries, as they ultimately target a human audience [Roy et al. 2021]. We argue, in line with prior research [Haque et al. 2022; Roy et al. 2021; Shen et al. 2021], that an effective summary evaluation metric should be closely aligned with human judgment. Therefore, we evaluate SimLLM to study whether it can precisely capture the semantic similarity between candidate and reference summaries, as rated by human annotators.

Evaluation Datasets. To evaluate SimLLM, we leverage three datasets, including two public datasets and our own dataset, which cover six programming languages and six code summarization models. Specifically, the public datasets include the Haque dataset [Haque et al. 2022] and the Shen dataset [Shen et al. 2021]. The Haque dataset consists of 220 distinct pairs of Java method summaries, with summaries generated by the Attendgru model [LeClair et al. 2019]. These Java methods are sampled from a large Java source dataset curated from UCI source code corpora [LeClair and McMillan 2019]. The Shen dataset has 7,747 Java method summaries, with Java methods from Github and summaries generated by the Seq2seq model [Shen et al. 2021].

While the public datasets are centered on Java methods, we expand the scope to build our own evaluation dataset to include code in Python, JavaScript, Ruby, Go, and PHP (*i.e.*, the five most popular programming languages [Roziere et al. 2023; Yang et al. 2022]). This dataset comprises 430 functions with ground truth summaries, randomly sampled from The Stack dataset [Kocetkov et al. 2022], across the five programming languages. To obtain model-generated summaries, we employed four state-of-the-art code summarization models: ChatGPT [OpenAI 2022], Magicoder [Wei et al. 2023], Llama [Touvron et al. 2023], and CodeT5 [Wang et al. 2021c], adhering to recognized code summarization practices [Wang et al. 2023]. These models rank the top in code summarization and understanding benchmarks [Cod 2024; Eva 2024]. To collect the gold standard, *i.e.*, human ratings of the ground truth and model-generated summaries, we perform an annotation study. Our approach of this study draws on established methods for assessing code summarization quality [Haque et al. 2022]. Essentially, we ask annotators to evaluate summaries based on semantic similarity. For this, we have recruited four participants, who have over 5.2 years of programming experience on average, to participate in our study. They were presented with a question assessing the semantic similarity of two function summaries (one reference and one candidate) and asked to rate summary similarity using a five-point scale: “Strongly Agree”, “Agree”, “Neutral”, “Disagree”, and “Strongly Disagree”. These responses correspond to scores of 5 to 1, respectively.

To address inter-annotator disagreements [Bobicev and Sokolova 2017], often stemming from individual biases, we implement three strategies in our study. First, we consistently refine the annotation guidelines and organize meetings with annotators to clarify these guidelines. These sessions lead to the addition of more detailed examples and the clarification of rules, reducing potential misunderstandings. Second, we adopt a majority voting system to aggregate annotators' individual ratings to final ratings, a method aligned with recognized annotation standards [Sabou et al. 2014]. Third, for instances where majority voting does not resolve biases, we hold discussions to address and clarify any ambiguities. To gauge the level of agreement among annotators, we calculate Krippendorff's α scores, a statistical measure of consensus for multi-coder annotation tasks [Krippendorff 2011]. Our approaches have improved Krippendorff's α from 0.205 to 0.436, indicating a significant reduction in annotator disagreement.

Evaluation Metrics. We employ Spearman's rank correlation coefficient [Kumar and Abirami 2018] as our evaluation metric to assess the extent to which SimLLM-generated similarity scores align with the gold standard, *i.e.*, human ratings. This metric is widely used for capturing the strength and direction of the monotonic relationship between two variables without assuming any specific linear relationship. It is notably suitable for our evaluations as it can deal with variables with distinct ranges, *e.g.*, human scores in the Shen dataset range from 1 to 5, while SimLLM's output cosine similarity scores span from -1 to 1. The coefficient's value spans from -1 to 1. A positive value signifies a positive correlation, with higher values indicating stronger correlations. To evaluate the null hypothesis (H_0) introduced in §3, we calculate the p-value of Spearman's rank correlation coefficient [Zwillinger and Kokoska 1999] to determine whether to reject H_0. Consistent with previous research [Haque et al. 2022], we establish the p-value threshold at 0.001.

Baselines. we compare SimLLM against a comprehensive set of existing code summary similarity evaluation metrics, *i.e.*, exact matching-based metrics and embedding-based metrics:

(I) Exact Matching-based Baselines. We utilize the prevalent BLEU, METEOR, and ROUGE-L metrics as baselines, assessing the extent of word overlaps between predicted and reference outputs. For BLEU, we report the average score across n-grams, where n ranges from 1 to 4. Regarding METEOR and ROUGE-L, we compute their F1 scores, derived from precision and recall.

(II) Embedding-based Baselines. We incorporate InferSent, USE, BERTScore, and SentenceBERT as our baseline methods. InferSent computes similarities with pre-trained word vectors [Conneau et al. 2017]. USE employs the Universal Sentence Encoder to generate context-aware word embeddings [Cer et al. 2018]. BERTScore exploits a pre-trained BERT to generate embeddings to evaluate summary similarity [Zhang et al. 2019]. SentenceBERT utilizes the BERT model in a siamese triplet network architecture to generate summary embeddings [Reimers and Gurevych 2019]. For BERTScore and SentenceBERT, we employ `roberta-base` and `stsb-roberta-base` base models, respectively, and report the corresponding F1 scores in our evaluations. These two base models possess the same number of transformer layers as SimLLM, enabling a fair comparison.

While BERTScore and SentenceBERT leverage the same transformer encoder model as SimLLM to grasp summary semantics, we enhance their capabilities by training them on the same pretraining dataset of SimLLM (see §5.2). This step aims to assess whether the advanced performance of SimLLM primarily stems from its pretraining design, instead of the domain-specific dataset. Specifically, as stated in §3, there are two popular domain-adaptation approaches: finetuning and continuous pretraining. Finetuning demands a substantial labeled dataset for summary semantic similarity, which is unavailable to us. Instead, continuous pretraining is more foundational and self-supervised, which involves training the existing model *without* requiring any domain-specific annotations. In this paper, we undertake continuous pretraining on the pretrained base models of BERTScore and SentenceBERT, using the masked language modeling objective. In this process, we use the same parameters as those in SimLLM's pretraining (see §5.2). We observe that the models' loss stabilizes after 29,000 updates. In our subsequent evaluations, we refer to continuously pre-trained BERTScore and SentenceBERT models as BERTScore$_{Pre}$ and SentenceBERT$_{Pre}$.

6.2 RQ1: Overall Effectiveness

To answer **RQ1**, we calculate Spearman's rank correlation coefficient of SimLLM on our evaluation datasets. The results are presented in Figure 5, where we report SimLLM's performance in the cosine similarity measure on summary embeddings (we also report other pairwise measures of SimLLM in §6.3). We have carried out five repeated experiments to minimize random factors, where we observed no variations in the calculated correlation coefficient scores across the experiments. *Remarkably, SimLLM outperforms all baselines across three datasets and demonstrates the strongest correlation with human ratings.* For example, it surpasses BLEU and METEOR by margins of 258.4% and 7.4%, respectively. Moreover, on our newly introduced dataset, we observed the significant

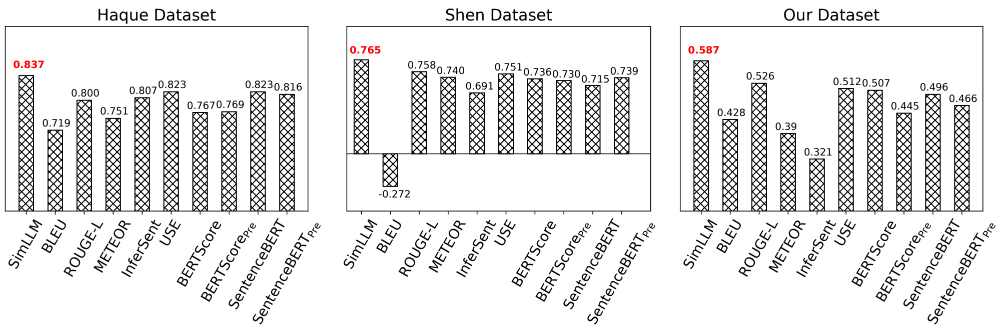


Fig. 5. Spearman's Rank Correlation Coefficient of SimLLM and Baseline Metrics on Our Evaluation Datasets.

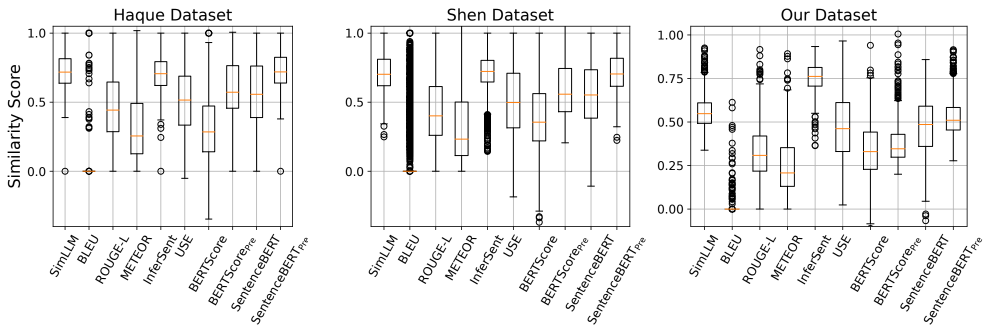


Fig. 6. Distribution of Similarity Scores Generated by Metrics on Our Evaluation Datasets

outperformance of SimLLM over all baselines, *e.g.*, 15.8% and 18.3% better coefficient compared to BERTScore and SentenceBERT.

More importantly, *SimLLM maintains its generalizable and robust performance even confronted with dataset variation as shown in Figure 5*. To validate this robustness, we compute p-value for statistical tests. The average p-value for SimLLM's Spearman's rank correlation coefficient across the datasets stands at 1.57×10^{-7}, which is significantly lower than our predetermined threshold (*e.g.*, 0.001). As stated in §3, this p-value leads to the rejection of the null hypothesis H_0, suggesting strong alignment between SimLLM 's results and human ratings. However, our baseline metrics show unstable performance across datasets. For instance, BLEU has a negative correlation with human ratings on the Shen dataset while having a positive correlation on Haque and our datasets. BLEU's coefficients change dramatically, undermining its reliability as a metric. We also examine the distributions of similarity scores generated by SimLLM and its baseline metrics, as presented in Figure 6. A surprising observation concerns the notably poor similarity score generated by BLEU. That is, most of the summary pairs in evaluation datasets received a BLEU score of zero, leading to a distribution of scores heavily skewed toward zero (*e.g.*, the orange line shows BLEU's median scores of zero), which indicates its failure to precisely compute semantic similarity.

In contrast, as shown in Figure 6, *SimLLM consistently assigns higher median similarity scores compared to most baselines metrics such as BLEU, USE, and SentenceBERT.* We attribute this behavior to SimLLM's domain-specific pretraining, which focuses on the particular context of code summaries. Meanwhile, we observe that the continuous pre-training helps BERTScore$_{Pre}$ and SentenceBERT$_{Pre}$ obtain higher semantic similarity scores in Figure 6. However, as depicted in Figure 5, this improvement does not translate into enhanced Spearman's rank correlation coefficients,

Table 1. Spearman's Rank Correlation Coefficients of SimLLM's Summary Embedding-based and Token Embedding-based Pairwise Similarity Measures on Our Evaluation Datasets.

Dataset	**Summary Embedding-based**				**Token Embedding-based**		
	Distance Function	**CLS**	**Mean**	**Max**	**Precision**	**Recall**	**F1**
Haque	Cosine	0.803	**0.836**	0.765	0.618	0.780	0.763
	Euclidean	0.802	**0.825**	0.800			
Shen	Cosine	0.752	**0.765**	0.706	0.611	0.740	0.732
	Euclidean	0.751	**0.753**	0.746			
Ours	Cosine	0.435	**0.587**	0.375	0.398	0.473	0.473
	Euclidean	0.435	**0.542**	0.429			

with the exception of SentenceBERT's performance on the Shen dataset. Our manual analysis indicates that BERTScore$_{Pre}$ and SentenceBERT$_{Pre}$ exhibit inconsistent performance when comparing summary pairs against human ratings. This finding is consistent with prior research [Wang et al. 2021a,b] on the efficacy of domain adaptation for existing models, suggesting that such adaptation may compromise the original model's capabilities.

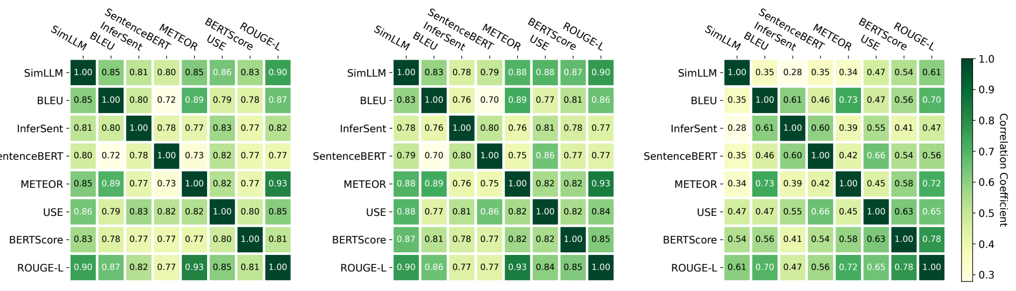
6.3 RQ2: Optimal Pairwise Similarity Measurement

To answer **RQ2**, we calculate the correlation coefficients of both the summary embedding-based and token embedding-based semantic similarity measures versus human ratings. In Table 1, we first present the performance of summary embedding-based metrics of SimLLM on our evaluation datasets, including six combinations based on two distance functions and three pooling methods. For SimLLM, the mean pooling offers a more faithful representation of summary semantics as opposed to CLS and max pooling techniques. And the cosine function is better than the Euclidean function in measuring the semantic distance in the embedding space. Next, in Table 1, we also present the correlation coefficients for SimLLM's token embedding-based semantic similarity measures, *i.e.*, precision, recall, and F1 score. Remarkably, within SimLLM, measures of recall conspicuously outperform those of precision, thereby resulting in F1 scores that lag behind recall scores.

Upon comparing the performance of all measures in Table 1, we conclude that *SimLLM's summary embedding-based similarity measure based on cosine function and the mean pooling method aligns the best with human ratings*. We hypothesize that this better performance of SimLLM stems from its alignment with the cognitive process of how humans comprehend code summaries. This process typically starts with understanding individual tokens and then integrates these token semantics into entire summaries. Additionally, we observe that SimLLM can automatically discern meaningless words, such as stop words, producing nearly identical similarity scores with and without these words when utilizing the cosine function and mean pooling. We attribute this capability to its dynamic semantic modeling of words, which learns embeddings of words under various contexts.

6.4 RQ3: Comparative Correlation with Other Metrics

To further understand SimLLM's effectiveness, we extend our analysis to its correlation with established automatic similarity metrics, *i.e.*, our baselines. This investigation serves two purposes: firstly, it pinpoints shared characteristics among SimLLM and other metrics; secondly, a strong correlation between metrics suggests they could be substituted for each other. Note that, we exclude BERTScore$_{Pre}$ and SentenceBERT$_{Pre}$ in this analysis as these versions are improved versions, trained by us, and are not recognized as standard, *i.e.*, well-established metrics. Figure 7 showcases the cross-metric correlations between SimLLM and our baseline metrics.



(a) Haque Dataset (b) Shen Dataset (c) Our Dataset

Fig. 7. Spearman's Rank Correlation Coefficient among SimLLM and Baseline Metrics

The exact match-based metrics exhibit strong correlations with each other, a finding that aligns with expectations given their underlying calculation mechanisms. For instance, METEOR and ROUGE-L align the most with BLEU. Moreover, InferSent displays unique characteristics, as evidenced by its low average correlation coefficients with others. It employs static word vectors to compute summary similarity [Conneau et al. 2017]. This approach distinguishes it from both exact match-based metrics and those based on dynamic embeddings (*i.e.*, BERTScore, SentenceBERT, and SimLLM).

Additionally, *SimLLM demonstrates strong correlations with several baselines, including METEOR, USE, BERTScore, and ROUGE-L*. For example, the correlation coefficients between SimLLM and ROUGE-L, and USE are 0.903 and 0.864 on the Haque dataset, respectively. ROUGE-L emphasizes the sequence overlaps, thus capturing structural and content similarities, rather than mere single-word overlaps [Lin 2004]. USE, on the other hand, generates context-aware embeddings that are aggregated into summary encodings via a sum operation [Cer et al. 2018]. Similarly, SimLLM calculates semantic similarity by aggregating token embeddings across the input sequence, enabling it to capture the semantics of long sequences, instead of individual words. SimLLM shares characteristics with both ROUGE-L and USE, blending aspects of structural focus with context-aware encoding.

By analyzing correlations with human ratings and other metrics in Figure 5 and Figure 7, *we posit that BLEU can be replaced by METEOR/ROUGE-L for code summary similarity evaluation.* BLEU's poor performance in capturing summary semantics, as evidenced by its mostly zero scores in Figure 6, indicates its shortcomings in evaluating short summaries, which has been a long-standing critique of BLEU [Clark et al. 2019; Radford et al. 2019]. For evaluations requiring understanding dynamic summary semantics, we contend that SimLLM presents a more suitable option compared to metrics that employ a similar embedding-based approach, such as USE and BERTScore.

6.5 RQ4: Exploratory Study on the Impact of Pretraining

To answer **RQ4**, we compare SimLLM's correlation with human ratings in pretrained and non-pretrained settings. In the non-pretrained setup of SimLLM, we initialize the model parameters using random values. This involves setting up the weights for the embedding, linear, and self-attention layers by drawing from a normal distribution, and initializing the biases of these layers to zero. This non-pretrained configuration allows SimLLM to operate purely based on its transformer architecture, devoid of any advantages conferred by pretraining. For the pretrained setting of SimLLM, we compute the summary similarity scores at five-epoch intervals throughout the pretraining phase. We then calculate correlation coefficients with human ratings using summary similarity scores generated for both pretrained and non-pretrained settings. Figure 8 presents these coefficients over pretraining epochs, where we use another dashed line to denote the performance of the non-pretrained configuration.

During the initial 20 epochs, SimLLM demonstrates a poor alignment with human ratings. This low coefficient in the coefficient is attributed to SimLLM's underfitting at first, due to the vast size of the pretraining dataset. That is, during the warmup stage of SimLLM's pretraining (see §5.2), the model navigates its parameter space to identify local minima, employing a large learning rate. Therefore, we hypothesize that underfitting leads to the optimization of parameter gradients in directions contrary to the minima [Kingma and Ba 2014]. However, after 25 epochs, SimLLM in its pretrained configuration consistently surpasses the performance of its non-pretrained version. This upward trajectory suggests that *SimLLM gradually obtains the capacity of learning code summary semantics, facilitated by the autoregressive language modeling objective as defined in §4.1.* Eventually, we note that the correlation coefficient stabilizes around the 90 to 100 epoch range.

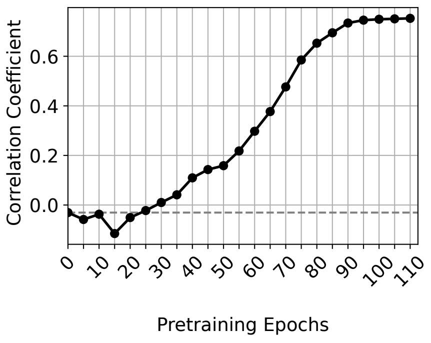


Fig. 8. Test Correlation Coefficient across Epochs

6.6 RQ5: Impact of Software Engineering Practices and Summarization Models

For **RQ5**, we initially examine SimLLM's performance across code written in different programming languages. Given that the Haque and Shen datasets are limited to Java code, our analysis concentrates on SimLLM's effectiveness within our annotated evaluation set. Here, SimLLM exhibits the different Spearman's rank correlation coefficients for Php, Javascript, Go, Ruby, and Python samples, respectively. This variation in performance is attributed to the quality of the ground truth code summaries, namely, developer-written comments. For instance, the average token length of Python comments in our dataset is 15.75, in contrast to Php's average of 9.57. A closer examination of these comments reveals that Python comments often contain supplementary details. For example, the comment for the `_detect_screen_size` function in our dataset is "*Attempt to work out the number of lines on the screen. This is called by page(). It can raise an error (e.g., when run in the test suite)*", which provides extra context about its caller and test suite, including information that introduces inconsistency in the predicted summary. This noisy and inconsistent information decreases the performance of both SimLLM and our baselines. Nevertheless, *SimLLM shows remarkable robustness to summary quality variations across different programming languages, outperforming all baselines.* For example, ROUGE-L, the best baseline metric that we identified in §6.2, shows a coefficient degradation of 44.3% from Php to Python samples. In contrast, SimLLM maintains a decrement of only 19.6%, highlighting its strong robustness across different coding contexts.

Similarly, our analysis identifies differences in how code summarization models produce summaries. The average token lengths for summaries generated by ChatGPT, Llama, Magicoder, and CodeT5 are 31.0, 20.8, 25.6, and 8.6, respectively. Consistent with findings from prior code summarization research [Jin et al. 2023], we find that ChatGPT tends to produce verbose summaries, often detailing code logic block by block rather than succinctly conveying the core semantics of the code, *e.g.*, ChatGPT's average token count of 31.0 significantly exceeds the ground truth average of 14.08 tokens. Corresponding to these lengths, SimLLM produces higher scores for shorter summaries. Moreover, we find ChatGPT often generates summaries with extra details, in the form of "*This function takes ... arguments, then ..., and returns ...*".

7 DISCUSSION

7.1 Software Engineering Insights and Impact

Unreliability of BLEU for Code Summary Semantic Assessment. While BLEU, the exact match-based metric, is commonly used in assessing code summary quality, it fails to understand semantics in summaries and our evaluations have shown its poor alignment with human judgments. For example, it mostly assigns zero similarity scores to code summaries in our datasets (see findings in 6.2). We propose that SimLLM, METEOR, and ROUGE-L can serve as effective alternatives to BLEU (details in 6.4). In our evaluations, SimLLM and ROUGE-L have much better correlation coefficients with human ratings than BLEU. Our perspective aligns with the longstanding criticism of BLEU in various research communities [Clark et al. 2019; Haque et al. 2022; Radford et al. 2019].

The Need of Code Summary-specific LLM Design. Applying pretrained models like BERTScore and SentenceBERT directly to code summary measurement, or enhancing them through continuous pretraining for code summary evaluation, encounters challenges due to the frequent use of code-specific abbreviations and terms. Our evaluations reveal that these metrics cannot align with human judgments compared with SimLLM, as shown in 6.2. Code summaries typically encapsulate semantics concisely, requiring models that can accurately retain this essence—a capability not met by these metrics. Moreover, as shown in Figure 5, the continuous pretraining of BERTScore and SentenceBERT has not improved their alignment with human evaluations, consistent with findings of recent LLM studies [Wang et al. 2021a,b]. We argue that applying LLMs to other summary-related tasks, *e.g.*, code summary categorization [Shi et al. 2022], also requires summary-specific LLM designs like SimLLM.

Prioritizing Concise and Effective Code Summarization is Paramount. In §6.6, we have noted significant variance in code comment quality across programming languages, with Python code, in particular, being verbose and often containing additional information not reflected in the code body. This practice not only poses challenges for software developers in understanding code but also affects summary assessment metrics to accurately evaluate code summarization frameworks. We advocate for emphasizing the conciseness of code comments in software development practices. Additionally, we also find that some code summarization models, such as ChatGPT, provide superficial explanations rather than capturing the core semantics of the code. Therefore, we argue for enhancing the capabilities of code summarization models to prioritize their effectiveness.

Software Engineering Impact of SimLLM. The introduction of SimLLM marks a significant advancement in automated code summarization by providing effective metrics for precisely assessing the semantic quality of generated summaries. For LLMs, such as ChatGPT and Code Llama, SimLLM could support the reinforcement learning process through feedback loops from automated metrics, enabling them to accurately grasp code semantics. The SimLLM's design allows for ongoing improvements, contributing to the creation of high-quality code summaries that closely reflect human expertise. Beyond enhancing code summarization systems, SimLLM's profound understanding of semantics offers advantages for other tasks requiring semantic modeling of summaries, like code summary categorization [Shi et al. 2022]. Essentially, SimLLM lays the groundwork for future code summarization frameworks and a range of applications focused on summary-specific tasks.

7.2 Limitations

Pretraining and Evaluation Dataset. Our pretraining and evaluation datasets originate from CodeSearchNet, the Stack dataset, and public repositories, covering code and related comments from six programming languages. Commenting practices and templates vary significantly across different programming languages, which may limit SimLLM's effectiveness for code comments outside of these six languages. However, SimLLM's design is agnostic to programming languages and code repositories. More importantly, SimLLM has been exposed to over 2 million real-world summary examples during its pretraining phase, and our evaluations showcase its broad generalizability. That said, we believe that training SimLLM with a larger dataset covering more diverse programs can be an interesting future research.

Summary Format. In our study, we concentrate on evaluating reference and candidate summaries presented in natural languages. However, summaries can also be represented through alternative means such as system calls and logs [Hao et al. 2023; Pan et al. 2023], which typically lack the structured nature of human languages. Although natural language summaries are the standard output for automated code summarization research, and our baseline metrics are also primarily designed for this format, we acknowledge that evaluating code summaries in other formats remains a future research problem.

Model Size and Computational Resource. SimLLM requires a higher computational overhead for pretraining, compared to our non-parametric baselines, *e.g.*, BLEU and METEOR. However, pretraining is a one-off effort, after which the pretrained models can be reused multiple times. For inference, SimLLM's computational requirements are comparable to those of BERTScore and SentenceBERT as their base models are the same in model sizes. Furthermore, the computational demands of code summarization, the primary application area for SimLLM, typically involve much larger models such as CodeT5 (100 times larger than SimLLM). This indicates that potential users of SimLLM likely possess the necessary computational infrastructure. Moreover, we have evaluated SimLLM on both CPU and GPU platforms, demonstrating efficiency with 0.0089 seconds on CPU and 0.0003 seconds on GPU for inference per sample. Further optimizing SimLLM through model compression techniques [Cheng et al. 2017] presents an interesting direction for future work.

8 CONCLUSION

In this paper, we present SimLLM, a novel approach for calculating semantic similarities of code summaries. By pretraining the large language model on permuted input, SimLLM showcases better alignment with human ratings compared to our baseline metrics, with a better comprehension of code summaries. Our analyses also reveal key software engineering insights in SimLLM's performance across programming languages and code summarization models. While SimLLM surpasses current metrics in performance, we recognize possible constraints, including the diversity and representativeness of the pretraining and evaluation datasets, the other summary formats, and the demands for computational resources. Evaluating other summary formats, more datasets, and new summarization models can be interesting for future research.

DATA AVAILABILITY

We have made the artifact, model, and datasets of SimLLM publicly available [Jin and Lin 2024].

ACKNOWLEDGMENTS

We thank Linxi Jiang, Steven Hu, and Boya Zhang for their help with our dataset preparation. We also appreciate the insightful comments from the anonymous reviewers. This research was supported by DARPA award N6600120C4020, ARO award W911NF2110081, and NSF award CNS-2112471. The opinions, findings, conclusions, or suggestions expressed are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

2024. CodeXGLUE. https://microsoft.github.io/CodeXGLUE/. Accessed: 2024-02-20.

2024. EvalPlus Leaderboard. https://evalplus.github.io/leaderboard.html. Accessed: 2024-02-20.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021). https://doi.org/10.48550/arXiv.2103.06333

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 65–72.

Victoria Bobicev and Marina Sokolova. 2017. Inter-annotator agreement in sentiment analysis: machine learning perspective. In *International Conference Recent Advances in Natural Language Processing*. 97–102. https://doi.org/10.26615/978-954-452-049-6_015

Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, et al. 2018. Universal sentence encoder. *arXiv preprint arXiv:1803.11175* (2018). https://doi.org/10.48550/arXiv.1803.11175

Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 2022. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 18–30. https://doi.org/10.1145/3540250.3549162

Stanley F Chen, Douglas Beeferman, and Roni Rosenfeld. 1998. Evaluation metrics for language models. (1998).

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282* (2017). https://doi.org/10.48550/arXiv.1710.09282

Elizabeth Clark, Asli Celikyilmaz, and Noah A Smith. 2019. Sentence mover's similarity: Automatic evaluation for multi-sentence texts. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2748–2760. https://doi.org/10.18653/v1/P19-1264

Alexis Conneau and Douwe Kiela. 2018. Senteval: An evaluation toolkit for universal sentence representations. *arXiv preprint arXiv:1803.05449* (2018). https://doi.org/10.48550/arXiv.1803.05449

Alexis Conneau, Douwe Kiela, Holger Schwenk, Loic Barrault, and Antoine Bordes. 2017. Supervised learning of universal sentence representations from natural language inference data. *arXiv preprint arXiv:1705.02364* (2017). https://doi.org/10.48550/arXiv.1705.02364

Souvick Das, Novarun Deb, Agostino Cortesi, and Nabendu Chaki. 2023. CoDescribe: An Intelligent Code Analyst for Enhancing Productivity and Software Quality. In *International Symposium on Applied Computing for Software and Smart Systems*. Springer, 161–181. https://doi.org/10.1016/j.softx.2024.101677

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018). https://doi.org/10.48550/arXiv.1810.04805

Wafaa S El-Kassas, Cherif R Salama, Ahmed A Rafea, and Hoda K Mohamed. 2021. Automatic text summarization: A comprehensive survey. *Expert systems with applications* 165 (2021), 113679. https://doi.org/10.1016/j.eswa.2020.113679

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020). https://doi.org/10.48550/arXiv.2002.08155

Shane Griffith, Kaushik Subramanian, Jonathan Scholz, Charles L Isbell, and Andrea L Thomaz. 2013. Policy shaping: Integrating human feedback with reinforcement learning. *Advances in neural information processing systems* 26 (2013).

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020). https://doi.org/10.48550/arXiv.2009.08366

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working conference on reverse engineering*. IEEE, 35–44. https://doi.org/10.1109/WCRE.2010.13

Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. 2023. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 3262–3278. https://doi.org/10.1109/SP46215.2023.10179298

Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. 2022. Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 36–47. https://doi.org/10.1145/3524610.3527909

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019). https://doi.org/10.48550/arXiv.1909.09436

Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. 2023. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. *arXiv preprint arXiv:2312.09601* (2023). https://doi.org/10.48550/arXiv.2312.09601

Xin Jin and Zhiqiang Lin. 2024. *SimLLM artifact*. https://doi.org/10.5281/zenodo.11095396

Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1631–1645. https://doi.org/10.1145/3548606.3560612

Xin Jin and Yuchen Wang. 2023. Understand legal documents with contextualized large language models. *arXiv preprint arXiv:2303.12135* (2023). https://doi.org/10.48550/arXiv.2303.12135

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014). https://doi.org/10.48550/arXiv.1412.6980

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *Preprint* (2022). https://doi.org/10.48550/arXiv.2211.15533

Klaus Krippendorff. 2011. Computing Krippendorff's alpha-reliability. (2011).

Ashok Kumar and S Abirami. 2018. Aspect-based opinion ranking framework for product reviews using a Spearman's rank correlation coefficient method. *Information Sciences* 460 (2018), 23–41. https://doi.org/10.1016/j.ins.2018.05.003

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 795–806. https://doi.org/10.48550/arXiv.1902.01954

Alexander LeClair and Collin McMillan. 2019. Recommendations for datasets for source code summarization. *arXiv preprint arXiv:1904.02660* (2019). https://doi.org/10.48550/arXiv.1904.02660

Omer Levy and Yoav Goldberg. 2014. Neural word embedding as implicit matrix factorization. *Advances in neural information processing systems* 27 (2014).

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

Shangqing Liu, Yanzhou Li, Xiaofei Xie, and Yang Liu. 2022. Commitbart: A large pre-trained model for github commits. *arXiv preprint arXiv:2208.08100* (2022). https://doi.org/10.48550/arXiv.2208.08100

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019). https://doi.org/10.48550/arXiv.1907.11692

Alexandra Luccioni and Joseph Viviano. 2021. What's in the box? an analysis of undesirable content in the Common Crawl corpus. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. 182–189. https://doi.org/10.18653/v1/2021.acl-short.24

Sabrina J Mielke, Zaid Alyafeai, Elizabeth Salesky, Colin Raffel, Manan Dey, Matthias Gallé, Arun Raja, Chenglei Si, Wilson Y Lee, Benoît Sagot, et al. 2021. Between words and characters: A brief history of open-vocabulary modeling and tokenization in NLP. *arXiv preprint arXiv:2112.10508* (2021). https://doi.org/10.48550/arXiv.2112.10508

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013). https://doi.org/10.48550/arXiv.1301.3781

OpenAI. 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt. Accessed: 2024-02-20.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038* (2019). https://doi.org/10.48550/arXiv.1904.01038

Myle Ott and et. al. 2023. Fast BPE. https://github.com/glample/fastBPE. 09-12-2023.

Yu Pan, Zhichao Xu, Levi Taiji Li, Yunhe Yang, and Mu Zhang. 2023. Automated generation of security-centric descriptions for smart contract bytecode. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1244–1256. https://doi.org/10.1145/3597926.3598132

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543. https://doi.org/10.3115/v1/D14-1162

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

Peter A Rankel, John Conroy, Hoa Trang Dang, and Ani Nenkova. 2013. A decade of automatic content evaluation of news summaries: Reassessing the state of the art. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 131–136.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019). https://doi.org/10.48550/arXiv.1908.10084

Stephen Robertson. 2004. Understanding inverse document frequency: on theoretical arguments for IDF. *Journal of documentation* 60, 5 (2004), 503–520.

Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1105–1116. https://doi.org/10.1145/3468264.3468588

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023). https://doi.org/10.48550/arXiv.2308.12950

Marta Sabou, Kalina Bontcheva, Leon Derczynski, and Arno Scharl. 2014. Corpus annotation through crowdsourcing: Towards best practice guidelines.. In *LREC*. Citeseer, 859–866.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015). https://doi.org/10.48550/arXiv.1508.07909

Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, Indira Vats, Hadi Moazen, and Federica Sarro. 2021. A survey on machine learning techniques for source code analysis. *arXiv preprint arXiv:2110.09610* (2021). https://doi.org/10.48550/arXiv.2110.09610

Juanjuan Shen, Yu Zhou, Yongchao Wang, Xiang Chen, Tingting Han, and Taolue Chen. 2021. Evaluating Code Summarization with Improved Correlation with Human Assessment. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 990–1001. https://doi.org/10.1109/QRS54544.2021.00108

Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? on the importance of data preprocessing for code summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 107–119. https://doi.org/10.1145/3540250.3549145

Catarina Silva and Bernardete Ribeiro. 2003. The importance of stop word removal on recall values in text categorization. In *Proceedings of the International Joint Conference on Neural Networks, 2003.*, Vol. 3. IEEE, 1661–1666. https://doi.org/10.1109/IJCNN.2003.1223656

Jikyoeng Son, Joonghyuk Hahn, HyeonTae Seo, and Yo-Sub Han. 2022. Boosting code summarization by embedding code structures. In *Proceedings of the 29th International Conference on Computational Linguistics*. 5966–5977.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2020. Mpnet: Masked and permuted pre-training for language understanding. *Advances in Neural Information Processing Systems* 33 (2020), 16857–16867. https://doi.org/10.48550/arXiv.2004.09297

Jing Su, Chufeng Jiang, Xin Jin, Yuxin Qiao, Tingsong Xiao, Hongda Ma, Rong Wei, Zhi Jing, Jiajun Xu, and Junhong Lin. 2024. Large Language Models for Forecasting and Anomaly Detection: A Systematic Literature Review. *arXiv preprint arXiv:2402.10350* (2024). https://doi.org/10.48550/arXiv.2402.10350

Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic Code Summarization via ChatGPT: How Far Are We? *arXiv preprint arXiv:2305.12865* (2023). https://doi.org/10.48550/arXiv.2305.12865

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023). https://doi.org/10.48550/arXiv.2307.09288

Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. 2023. Prompt Tuning in Code Intelligence: An Experimental Evaluation. *IEEE Transactions on Software Engineering* (2023). https://doi.org/10.1109/TSE.2023.3313881

Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022. Bridging pre-trained models and downstream tasks for source code understanding. In *Proceedings of the 44th International Conference on Software Engineering*. 287–298. https://doi.org/10.48550/arXiv.2112.02268

Kexin Wang, Nils Reimers, and Iryna Gurevych. 2021a. Tsdae: Using transformer-based sequential denoising auto-encoder for unsupervised sentence embedding learning. *arXiv preprint arXiv:2104.06979* (2021). https://doi.org/10.48550/arXiv.2104.06979

Kexin Wang, Nandan Thakur, Nils Reimers, and Iryna Gurevych. 2021b. Gpl: Generative pseudo labeling for unsupervised domain adaptation of dense retrieval. *arXiv preprint arXiv:2112.07577* (2021). https://doi.org/10.48550/arXiv.2112.07577

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021c. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021). https://doi.org/10.48550/arXiv.2109.00859

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120* (2023). https://doi.org/10.48550/arXiv.2312.02120

Edmund Wong, Taiyue Liu, and Lin Tan. 2015. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 380–389. https://doi.org/10.1109/SANER.2015.7081848

Wei Wu, Houfeng Wang, Tianyu Liu, and Shuming Ma. 2018. Phrase-level self-attention networks for universal sentence encoding. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 3729–3738. https://doi.org/10.18653/v1/D18-1408

Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A survey on deep learning for software engineering. *ACM Computing Surveys (CSUR)* 54, 10s (2022), 1–73. https://doi.org/10.1145/3505243

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems* 32 (2019). https://doi.org/10.48550/arXiv.1906.08237

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2019. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962* (2019). https://doi.org/10.48550/arXiv.1904.00962

Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. 2019. A review of recurrent neural networks: LSTM cells and network architectures. *Neural computation* 31, 7 (2019), 1235–1270.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675* (2019). https://doi.org/10.48550/arXiv.1904.09675

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. Codebertscore: Evaluating code generation with pretrained models of code. *arXiv preprint arXiv:2302.05527* (2023). https://doi.org/10.48550/arXiv.2302.05527

Daniel Zwillinger and Stephen Kokoska. 1999. *CRC standard probability and statistics tables and formulae*. Crc Press.

Received 2023-09-28; accepted 2024-04-16