Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization

Antonio Mastropaolo
antonio.mastropaolo@usi.ch
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

Matteo Ciniselli
matteo.ciniselli@usi.ch
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

Massimiliano Di Penta
dipenta@unisannio.it
Dept. of Engineering,
University of Sannio
Benevento, Italy, IT

Gabriele Bavota
gabriele.bavota@usi.ch
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

ABSTRACT

Several code summarization techniques have been proposed in the literature to automatically document a code snippet or a function. Ideally, software developers should be involved in assessing the quality of the generated summaries. However, in most cases, researchers rely on automatic evaluation metrics such as BLEU, ROUGE, and METEOR. These metrics are all based on the same assumption: The higher the textual similarity between the generated summary and a reference summary written by developers, the higher its quality. However, there are two reasons for which this assumption falls short: (i) reference summaries, *e.g.,* code comments collected by mining software repositories, may be of low quality or even outdated; (ii) generated summaries, while using a different wording than a reference one, could be semantically equivalent to it, thus still being suitable to document the code snippet. In this paper, we perform a thorough empirical investigation on the complementarity of different types of metrics in capturing the quality of a generated summary. Also, we propose to address the limitations of existing metrics by considering a new dimension, capturing the extent to which the generated summary aligns with the semantics of the documented code snippet, independently from the reference summary. To this end, we present a new metric based on contrastive learning to capture said aspect. We empirically show that the inclusion of this novel dimension enables a more effective representation of developers' evaluations regarding the quality of automatically generated summaries.

CCS CONCEPTS

• **Software and its engineering → Documentation**.

KEYWORDS

Code Summarization, Contrastive Learning

1 INTRODUCTION

Program comprehension can take up to 58% of developers' time [90]. Code comments are considered the most important form of documentation in this activity [16]. Despite the undisputed importance of code comments, developers do not always carefully comment code, or update existing comments in response to code changes [75]. This may result in a lack of documentation [16, 72] and/or in outdated code comments [18, 19, 46, 86]. To support developers in such a task, researchers proposed code summarization techniques [4, 6, 26, 29, 32, 37, 41, 52, 53, 64, 67, 73, 73, 87, 88, 93]. These approaches take as input a code component to document (*e.g.,* a code function, or an entire class) and provide as output a natural language summary describing the code. The underlying technique can range from pre-defined templates properly filled via code analysis to the most recent techniques exploiting deep learning models trained on $\langle C, S \rangle$ pairs mined from software repositories, where C represents the code to document and S the original summary (comment) written by developers.

Empirically evaluating the quality of code summaries generated by these approaches is far from trivial. Indeed, assessing the extent to which a natural language text represents a good summary for a code component would require human (developers) judgment. Given the difficulties of running large-scale evaluations with developers, the software engineering community borrowed evaluation metrics from the Natural Language Processing (NLP) field. These include (but are not limited to) BLEU [58], ROUGE [45], and METEOR [8]. These metrics have been originally designed to act as a proxy for the quality of automatically generated text (*e.g.,* a translation) by comparing it with a reference (expected) text: The higher the words' overlap between the generated and the reference text, the higher the assessed quality.

In NLP, the reference text is usually a collection of valid outcomes (*e.g.,* a set of valid translations for an input sentence) and the generated text is compared (in terms of word overlap) with all possible outcomes to obtain a better assessment. Also, the quality of the reference texts used for the evaluation is usually guaranteed by multiple human evaluators.

When adopting such metrics for code summarization, the generated summary is contrasted against a single reference text, usually being the original comment written by developers for the code provided as input, which can be easily mined from software repositories. However, two important issues emerge when using the NLP metrics in this context. First, there is no guarantee that the reference text is of high quality, as also demonstrated by empirical studies documenting quality issues in code comments (see *e.g.,* [18, 46, 86]). Thus, computing a word overlap between the generated and the reference summary may provide misleading indications of the actual quality of the generated summary. Second, the mined comment is only one of the possible ways to summarize the related code. Metrics based on word overlap penalize generated summaries for being different but semantically equivalent to the reference one, thus again not being good proxies for the summary quality.

The software engineering research community is well aware of the limitations of these metrics [28, 68]. In response to the second limitation previously discussed (*i.e.,* capturing similarity between summaries using different wording but being semantically equivalent), Haque *et al.* [28] proposed the usage of word/sentence embeddings to properly capture the semantic similarity across summaries, moving from word overlapping to word-similarity measurement. While the authors show that these metrics better correlate with the human judgment of code summary quality as compared to word-overlapping metrics (*e.g.,* BLEU, ROUGE, METEOR), they still suffer of the first limitation we discussed, *i.e., a high similarity to a low-quality reference summary may provide a misleading good assessment of a generated summary.*

In this paper, we argue that an important factor in the assessment of summary quality is currently ignored by the state-of-the-art metrics: "The suitability of the generated summary for the code to document, independently from the original comment written by developers". To provide evidence of that, we present an empirical study that analyzes to what extent, different metrics to assess automatically-generated code summaries correlate, and complement with each other in explaining human assessment of such summaries. More importantly, we check for the complementarity of metrics capturing the new dimension with respect to others. We rely on a dataset by Roy *et al.* [68], featuring more than 5k human evaluations of automatically generated summaries. We relate such evaluations to the quality assessment provided by different families of metrics. To capture the suitability of the generated summary for the code to document we experiment with (i) a simple approach relying on the word overlap between the summary and the code [76]; (ii) a deep learning-based approach exploiting embeddings obtained via a model pre-trained on code [84]; and (iii) SIDE (**S**ummary al**I**gnment to co**D**e s**E**mantics), a new metric leveraging contrastive learning [70] to model the characteristics of suitable and unsuitable code summaries for a given code. Our results show that (i) focusing on assessing the suitability of the generated summary for the documented code ignoring the reference summary allows

to capture orthogonal aspects of summary quality as compared to state-of-the-art metrics, such as BLEU, ROUGE, METEOR, Jaccard similarity, as well as metrics based on word/sentence embeddings; and (ii) SIDE is the metric having the strongest correlation with human judgment of code summary quality. We also show that SIDE can be combined with state-of-the-art metrics to provide a more comprehensive assessment of code summary quality.

2 RELATED WORK

This work relates to studies investigating factors impacting the soundness of evaluations performed to assess code summarization techniques (Section 2.1). Also, as SIDE can be used to assess the quality of a code summary for a given code snippet, we also discuss metrics capturing code comment quality (Section 2.2).

2.1 Evaluating Code Summarization Techniques

LeClair and McMillan [43] discuss the implications of different preprocessing choices made when preparing the datasets used for the evaluation of code summarization techniques, *e.g.,* splitting the training/test datasets by function *vs* by project.

They show that the impact of these choices can be drastic (up to ±33% of performance), and advocate for more standard dataset preparation procedures aimed at increasing results reproducibility.

Stapleton *et al.* [74] present a study involving 45 university students and professional developers, in which participants were asked to perform program comprehension tasks on functions documented by human-written or automatically generated summaries. They found that participants performed significantly better with human-written summaries. Also, they show that the BLEU and ROUGE metrics computed on the automatically generated summaries do not correlate with the participants' performance in comprehending the documented code. This poses questions on the suitability of these metrics to assess the quality of code summaries, at least when they are used to support program comprehension.

Gros *et al.* [23] discuss the basic assumption on which several code summarization techniques are built: The code summarization problem resembles the "natural language translation" problem, meaning that it can be seen as a "code-to-natural language translation". The validity of such an assumption would imply the possibility to reuse metrics successfully applied to assess the performance of automated natural language translations (*e.g.,* BLEU) in the context of code summarization. The empirical findings by Gros *et al.* show, however, that the two problems are substantially different. For example, while there is a strong input-output dependence in natural language translation (*i.e.,* similar input sentences result in similar output translations), this is not necessarily the case for code summarization (*i.e.,* similar functions could be commented in completely different ways).

Roy *et al.* [68] focus on the interpretation of metrics used in code summarization, and in particular on BLEU, ROUGE (in several different variants), METEOR, chrF [60], and BERTScore [95]. They conducted a study with researchers and practitioners asking them to assess the quality of 36 summaries associated with 6 code snippets (a Java method). Each snippet had six summaries associated, one being the reference summary (*i.e.,* the one written by the original

developers) and five resulting from different code summarization techniques [7, 29, 42, 81, 91].

The quality assessment has been performed using an ordinal scale in the range [0, 5] [55] (the higher the better) and focusing on three different aspects of each summary: conciseness, fluency, and content adequacy. Participants were not aware of which summaries were automatically generated and which, instead, represented the reference summary. Overall, they collected 226 surveys, for a total of 6,253 evaluations (not all participants fully completed the survey). By computing the above-listed evaluation metrics for the generated summaries and comparing them with the human assessment, Roy *et al.* found that metric improvements of less than two points do not indicate any meaningful difference in the quality of the summaries generated by two techniques. For higher differences metrics such as METEOR become reliable proxies for differences in summary quality, while others such as BLEU remain unreliable. We reuse the dataset by Roy *et al.* to assess the extent to which SIDE and other metrics can be used to assess generated code summaries.

In a subsequent study, Hu *et al.* [31] confirmed METEOR as the metric having the strongest correlation with the human judgment of code summary quality. Still, the achieved correlation is lower than that observed by human raters. Haque *et al.* [28] criticize the use of word overlap metrics such as BLEU and ROUGE for assessing the quality of automatically generated summaries. They observe that using word overlap ignores the fact that (i) not all words have the same importance in a sentence; and (ii) the usage of synonyms in the generated summary (*e.g.,* using "delete" instead of "cancel" as in the reference summary) penalizes its quality assessment, which is conceptually wrong. Starting from this observation, they propose the usage of "semantic similarity" metrics based on word/sentence embeddings, which can better capture the similarity between the generated and the reference summaries. They then conducted a study involving 30 professional programmers, each of which was asked to evaluate the quality of a single summary associated with 210 Java methods. Their findings show that embedding-based metrics such as SentenceBERT [66] might be more suitable as quality assessment metrics for code summaries as compared to word overlapping metrics such as BLEU. In our study, we also consider all code summary quality metrics used in the work by Haque *et al.*

2.2 Assessing the Quality of Code Comments

Several works focused on the automated identification of inconsistent comments, namely comments that are not aligned with the code they document [48, 77–79, 83]. While these techniques are extremely valuable, they are not suited for the quality assessment of an automatically generated summary for the following reasons. First, their output is usually a binary classification indicating whether the comment is consistent or inconsistent, lacking one of the key characteristics of a metric, such as the possibility to compare and rank instances. Second, they focus on a specific aspect of low-quality comments, namely its inconsistency (*e.g.,* the comment documents a parameter no longer present in the method). However, a "consistent comment" might still be of low quality if it is too verbose, difficult to read, etc. As we will detail in Section 3, SIDE (i) provides a continuous score ranging between [-1, 1], thus allowing comparing and ranking instances; and (ii) being trained via contrastive learning, learns the characteristics of suitable and

unsuitable code summaries for a given code rather than relying on manually-crafted definitions of low-quality comments.

Khamis *et al.* [39] propose JavadocMiner, a tool measuring several metrics which the authors consider important for the quality of Javadoc comments, *e.g.,* readability indexes, number of nouns in the comment, whether the Javadoc documents code entities (*e.g.,* parameters, return types) are not implemented in the documented code, etc. This approach exploits Javadoc-specific heuristics which cannot be generalized to the unstructured comments usually generated by code summarization techniques. Also, the quality-related metrics have been defined by the authors and never empirically evaluated against the developers' perception of comment quality.

Steidl *et al.* [76] propose a machine learning-based approach to classify code comments into seven categories (*i.e.,* copyright, header, member, inline, section, code, and task). Also, they propose a metric named c_coeff to identify low-quality comments. The metric is based on the percentage of words in a comment that is similar to words in the code, where two words are considered similar if they have a Levenshtein distance lower than two (*i.e.,* at most one character must be changed to convert one word into the other). We consider c_coeff in our study as a possible metric to capture the suitability of the generated summary for the documented code, showing that SIDE better captures summary quality as perceived by developers.

3 SIDE

We present SIDE, our novel metric to assess whether a natural language text represents a suitable summary for a given code. First, we provide background information about the DL model on top of which SIDE is based (Section 3.1), and the contrastive learning procedure used to train it (Section 3.2). Then, we describe the dataset used for the model's training (Section 3.3). Finally, Section 3.4 provides the details (*e.g.,* parameters) of the training procedure.

3.1 MPNet in a Nutshell

MPNet (Masked and Permuted Pre-training for Language Understanding) [71] is a Transformer [81] pre-trained model built on top of BERT [17]. Before discussing its architecture, let us briefly introduce the notion of pre-trained models. Pre-trained Transformers achieved state-of-the-art results in several Natural Language Processing (NLP) tasks [5, 17, 24, 44, 47, 63, 71, 92, 92, 94]. The pre-training, together with the self-attention mechanism featured in the Transformer architecture [81], played a major role in these achievements. The idea of pre-training is to provide the model with general knowledge about a language of interest before specializing it for a specific task. For example, let us assume we want to create an English-to-French translator. The model can be pre-trained on a large amount of unlabelled English and French data (*e.g.,* articles extracted from Wikipedia) using a self-supervised training objective, such as masked language modeling (MLM). MLM consists in randomly masking a percentage of the tokens in a given (English or French) sentence asking the model to guess them. For example, applying MLM to a sentence $\langle T_1, T_2, T_3, T_4, T_5 \rangle$ we could obtain $\langle T_1, M, T_3, M, T_5 \rangle$ (*i.e.,* T_2 and T_4 have been masked). The input for the model is the masked sentence, while the expected output are T_2 and T_4 as replacements for the two masked tokens. The idea is that thanks to MLM, the model starts acquiring knowledge about the

languages' structure, preparing it to be specialized (fine-tuned) for the task of interest, namely language translation.

The MLM pre-training objective has been adopted by several Transformer architectures, such as BERT [17]. However, MLM suffers from a limitation: It ignores the dependencies among the masked tokens, possibly limiting the learning of complex semantic relationships. To overcome this limitation, Yang *et al.* [92] proposed in XLNet the usage of permuted language modeling (PLM) during pre-training. This forces the model to learn long-range relations between tokens by guessing the correct positioning of the tokens in the whole sentence. While the technical details can be found in the paper introducing the technique [71], the basic idea is that at pre-training time the model is provided with a permuted sentence featuring masked tokens and, on top of that, with original positioning information, *i.e.,* what was the original position of the tokens in the sentence before permutation. MPNet managed to achieve new state-of-the-art results in several works from the NLP community [20, 33, 51, 56], including those relying on contrastive learning as training procedure [57, 85], and it is the pre-trained model we specialize for the task of classifying a textual summary as suitable or not for a given code.

In terms of architecture, MPNet builds upon the $BERT_{base}$ model, which comprises 12 transformer layers with a hidden size of 768, 12 attention heads, and a total of 110M trainable parameters. MPNet was pre-trained using the same corpora exploited for the training of RoBERTa [47], which includes datasets such as Wikipedia and BooksCorpus [97], OpenWebText [21], CC-News [1], and Stories [3], summing up for a total of 160GB of textual data.

3.2 Contrastive Learning

Contrastive learning [70] allows DL models to learn an embedding space where similar sample pairs (*i.e.,* pairs sharing specific features) are clustered together while dissimilar pairs are set apart. In our context, we use contrastive learning to discriminate suitable *vs* unsuitable summaries for a given source code snippet. To this aim, we need to show to the model both *positive samples* (source code associated with suitable summaries) and *negative samples* (source code associated with unsuitable summaries).

Several contrastive representation learning losses have been proposed in the literature [25, 35, 40, 70]. We employ the *triplet loss* [70], which has been shown to better encode the positive/negative samples as compared to other contrastive losses [12]. The triplet loss function has been proposed by Schroff *et al.* [70] and introduces the concept of "anchor". Given an anchor x, a positive (x^+) and a negative (x^-) sample is selected, with the triplet loss which during training minimizes the distance between the x and x^+, while maximizing the distance between x and x^-. In our case, the anchor is the code to document, with a suitable summary representing x^+ and an unsuitable summary representing x^-. In the following, we introduce the dataset used to fine-tune MPNet for the task of interest, explaining how we generate positive and negative samples.

3.3 Fine-tuning Dataset

We need to collect code instances paired with "suitable" and "unsuitable" code summaries. In our evaluation of SIDE (Section 4) we

will exploit a dataset from the literature featuring developers' evaluation of code summaries for Java methods [68]. Thus, we started by collecting Java methods accompanied by a textual summary.

We exploited the CodeSearchNet dataset [36], featuring ~6M functions from various programming languages, including Java, and a subset of Java methods is accompanied by a Javadoc description. Lu *et al.* [50] observed that the original CodeSearchNet dataset featured instances potentially being problematic. Therefore, they created a curated version of the dataset excluding methods that cannot be parsed, and those paired with a Javadoc (usually, the method's summary) having its first sentence shorter than 3 or longer than 256 tokens, and featuring special tokens (such as ⟨*img*⟩), or not being written in English. Their refined Java subsection of the CodeSearchNet dataset comprises 181,061 pairs of ⟨*method*, *summary*⟩ which have been already split into three subsets: 164,923 training, 5,183 validation, and 10,955 testing. The *summary* here is the first sentence of the Javadoc extracted as the first paragraph of the documentation (*i.e.,* the one delimited with the first period) [50]. To increase the confidence in the quality of the exploited dataset and verify whether the sentences automatically extracted from the documentation actually represent summaries of the method, two of the authors independently inspected 100 randomly selected samples from the dataset, classifying their associated documentation as "code summary" or "other". The guideline was to classify it as a code summary if it summarizes the intent of the method. After solving 2 cases of disagreement, 95 of the inspected documentations were classified as actual summaries.

The starting assumption is that the original summary written by the developers is a positive sample when paired with its associated method. This makes for 164,923 positive samples in our dataset. The same number of negative samples can be easily created by associating each method with a randomly selected summary from the training set (different from the original summary). The result is a dataset of 164,923 ⟨*method*, *positive*, *negavite*⟩ triplets featuring, for each method, a positive and a negative summary. The approach used to create the negative samples, while simple, may associate unrelated summaries to methods, simplifying the learning of the model, *i.e.,* it becomes rather easy to discriminate between positive and negative samples at training time. However, when assessing the quality of an automatically generated summary (our final goal for SIDE), it is unlikely that the latter is completely unrelated to the input method, even when it is of low quality. Thus, we must train SIDE so that it is able to identify as "unsuitable" for a given method even summaries which are plausible yet still suboptimal. These are known in the literature as hard-negative samples [54, 70].

To automatically generate these summaries, we conjecture that inner comments only documenting a subset of the method's statements are unsuitable as "method's summary". Nevertheless, they are still likely to be related to the method, certainly more than randomly selected summaries. Thus, we parsed the methods in the training set to extract all their inner comments and associated each inner comment to the set of statements it documents. For such association, we use the heuristics previously proposed in the literature, linking each inner comment to all following statements until an empty line or a closing curly bracket is reached [11, 34]. Comments reporting self-admitted technical debt [61] have been identified using keywords matching and excluded (we removed all

comments including one of the following words: *to-do, fix-me, todo, fixme, xxx, hackme, hack-me*).

Indeed, these comments do not describe the code. We then computed the percentage of statements in the method that each comment documents, only considering actual code statements (*i.e.,* excluding comments and blank lines). We consider a comment as a good hard-negative sample (*i.e.,* a plausible method description being, however, unsuitable as a method's summary) if it documents less than 25% of the method's statements. The choice of the 25% threshold is motivated by the following assumption: if a comment describes at most one fourth of the statements in a method, it is unlikely that it can represent a comprehensive summary of it while still possibly documenting some of the responsibilities implemented in the method. We managed to create a hard-negative for ~15% (24,951) of the instances in our training dataset, since the others did not have any inner comment matching our selection procedure.

Our training dataset features 164,923 triplets generated using random negatives, and 61,001 triplets featuring instead hard-negatives, since each of the 24,951 methods for which we managed to create a hard-negative can contribute with more than a single instance.

3.4 Training and Model Evaluation

MPNet has been trained for 10 epochs (which accounted for 141,205 training steps) using a batch size of 16 and a maximum sequence length of 512 tokens. The sequence length acts on the input of the model (in our case, the concatenation of a method and its summary), cutting out longer sequences. This impacted only 4.27% of the instances in our fine-tuning dataset. The learning rate has been warmed-up by taking into account the overall size of the training dataset, batch size and number of epochs. This strategy, which increases the learning rate from 0 to 2e-5 (default values when using the AdamW optimizer [49]) has been previously adopted when training BERT-based models using Sentence Transformers [66] The best-performing checkpoint which we found to be the last one saved (*i.e.,* after 141,205 training steps) has then been selected as the one maximizing the following score (also used in previous work using contrastive learning [69, 96]):

To reduce the chance of overfitting, we save checkpoints every 5k training steps while using a patience of 5. Specifically, we evaluate the model on the validation set (Section 3.3), which includes 5,183 positive samples (the original descriptions associated with each method) and the same number of negative samples randomly generated as explained in Section 3.3. The best-performing checkpoint has then been selected as the one maximizing the following score (also used in previous work using contrastive learning [69, 96]):

$$\frac{\sum_{i=1}^{N} \text{CSpositive}_i - \text{CSnegative}_i}{N}$$

where N represents the number of positive and negative samples (which in the evaluation set is guaranteed by construction to be the same), CSpositive_i represents the cosine similarity returned by the model between the method m_i and its "positive" summary, while CSnegative_i represents the cosine similarity returned between method m_i and its "negative" summary. A perfect model would return 1.0 for such a metric, reporting 1.0 as the similarity of all positive summaries and -1.0 as the similarity of all negative summaries. We acknowledge that our evaluation set features randomly

generated negative instances that, as such, are simple to identify by the model.

However, such a procedure is just used to select the best-performing checkpoint for SIDE, and not for its empirical evaluation (described in Section 4).

4 STUDY DESIGN

The *goal* of this study is to evaluate metrics used in the literature to assess the quality of code summarization approaches, including our newly proposed metric SIDE using contrastive learning. The *quality focus* is the complementarity of the considered metrics with respect to others, and the extent to which the considered metrics would explain summary quality assessments performed by developers. The *perspective* is of researchers wanting to define a framework for the assessment of code summary quality, that can be used, for example, to evaluate code (re)documentation approaches.

The *context* consists of (i) the dataset by Roy *et al.* [68] featuring more than 5k developers' quality assessments of automatically generated summaries; and (ii) 40 summary evaluation metrics, including SIDE. The study addresses the following research questions:

RQ$_1$: *To what extent different metrics to evaluate the quality of source code summarization correlate with each other?* Before analyzing how metrics contribute to explaining the quality of generated summaries, we analyze the extent to which they are related with each other, or, instead, capture different dimensions of the dataset variability.

RQ$_2$: *To what extent different metrics to evaluate the quality of source code summarization contribute to explain user-based evaluations?* After having identified a subset of unrelated metrics capturing the dataset variability, we analyze the extent to which these metrics contribute to explaining different summarization quality indicators. This would aim at providing a quantitative indication not only of the complementarity of the different metrics but also of the importance of each metric to describe an extrinsic quality indicator.

4.1 Evaluation Dataset

We use the dataset by Roy *et al.* [68] described in Section 2. As previously explained, this dataset features humans' evaluations of both automatically generated summaries for Java methods as well as of the original summaries written by the methods' developers. The latter are not suitable for our study. Indeed, to compute some of the evaluation metrics (*e.g.,* BLEU score), we need an automatically-generated summary to contrast against a reference summary. Thus, we remove from the set of 6,253 evaluations the 1,052 evaluations referring to the (manually written) original summaries, leaving us with the dataset of 5,201 evaluations.

4.2 Variable Selection

The 5,201 evaluations performed by developers [68] concern three quality aspects of summaries, all rated on an ordinal scale from 0 to 5 [55] (the higher the better):

- **Conciseness:** Assesses the degree to which the summary contains unnecessary information.

- **Fluency:** Evaluates the continuity or smoothness rate in the generated summary.
- **Content Adequacy:** Assesses the extent to which the summary lacks information needed to understand the code.

In addition Roy *et al.* [68] also collected a Direct Assessment (DA) score [22], expressed on a scale from 0 to 100, and providing an overall quality assessment of the summary. The three quality aspects (conciseness, fluency, and content adequacy), and the DA score represent the *dependent variables* of our study.

Regarding the *independent variables—i.e.,* automated evaluation metrics for code summaries—we considered, besides SIDE, (i) all metrics in the work by Roy *et al.* [68], (ii) additional metrics in the work by Haque *et al.* [28] (see Section 2), (iii) the c_coeff metric proposed by Steidl *et al.* [76], and (iv) a baseline we define based on CodeT5+ [84] (a Transformer pre-trained on code and English text) to compute the textual similarity between the generated summary and the code. The latter has the purpose to let us check the actual benefits (if any) brought by the contrastive learning we adopt in SIDE as compared to other DL-based approaches. The considered metrics, besides being a comprehensive set of those used in the literature to assess code summarization techniques, approach the quality assessment of code summaries in different ways.

In particular, the ones inherited from the works by Roy *et al.* [68] and Haque *et al.* [28] are conventional metrics, in the sense that they look at the similarity between the generated summary and the reference summary. These metrics range from very simple (*e.g.,* word overlapping between the two summaries), to more complex ones exploiting DL-based embeddings. Differently, c_coeff [76], CodeT5+, and SIDE look for the similarity between the generated summary and the documented code. This is the first time such a dimension is considered in the assessment of code summarization techniques. While several approaches can be used to measure such a similarity, we opted for three metrics including a "trivial" solution (*i.e.,* the c_coeff metric exploiting word overlap), a state-of-the-art Transformer pre-trained model (CodeT5+), and a contrastive learning-based solution being SIDE. For what concerns SIDE, we consider two of its variants: One exploiting hard-negatives in the training set (as extracted using the procedure described in Section Section 3.3) and one only including random negatives. This is basically an ablation study investigating the role played by the hard-negatives on the ability of SIDE to assess the suitability of a summary for a given code.

4.2.1 Words/characters-overlap based Metrics.

BLEU (BilinguaL Evaluation Understudy) [58] measures the similarity between the candidate (predicted) and reference (oracle) summaries. Such a similarity assesses the overlap in terms between the two summaries and it is defined on a scale between 0 (completely different summaries) to 1 (identical summaries). We compute the BLEU score at the sentence level for various values of n, including $n=\{1, 2, 3, 4\}$. We also compute the BLEU-A being the geometric mean of the four BLEU variants we consider.

METEOR (Metric for Evaluation of Translation with Explicit ORdering) [8] is computed as the harmonic mean of unigram precision and recall, with recall being assigned a higher weight than precision. In contrast to BLEU, METEOR incorporates stemming and synonyms matching to align more closely with human perception of similarity between sentences. The METEOR score ranges from 0 to 1, with a score of 1 indicating two identical sentences.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [45] is a set of metrics for evaluating both automatic summarization of texts and machine translation techniques. ROUGE metrics compare an automatically generated summary or translation with a set of reference summaries (typically, human-produced). Similarly to Roy *et al.* [68], we compute ROUGE-N(1-4), ROUGE-L, and ROUGE-W. ROUGE-N measures the number of matching *n*-grams between the generated summary and the reference summary with results reported in terms of recall, precision and f1-score.

Jaccard [27] measures the degree of overlap between two sets of tokens (summaries in our case). It is calculated by dividing the size of their intersection by the size of their union, thus obtaining a value between 0 and 1 (the higher the Jaccard, the more similar the two summaries).

chrF (character *n*-gram F-score) [60] measures the similarity between the generated and the reference summaries at the character level (rather than at the word-level as done by the above metrics), reporting the computed value using the F-score.

c_coeff [76], while still based on word overlap, focuses on the similarity between a summary and its associated code: It computes the percentage of words in a summary that is similar to words in the code, where two words are considered similar if they have a Levenshtein distance lower than two.

4.2.2 Embedding-based Metrics.

TF-IDF (Term frequency-inverse document frequency) [65] is a widely used term weighting schema assessing the importance of a word within a document collection. In our context, TF-IDF is used to compute the cosine similarity (TF-IDF_CS) and the Euclidean distance (TF-IDF_ED) between the terms' vectors representing the generated and the reference summary.

BERTScore [95] computes sentence similarity using the embedding of the BERT model [17], which has been trained on English textual data. We report all the BERTScores which include, precision (BERTScore-P), recall (BERTScore-R), and F1-score (BERTScore-F1).

SentenceBERT [66] employs a siamese network architecture to generate fixed-length representations of sentences using BERT [17] as a backbone to produce the encoding. The representations of the generated and the reference summary are compared via cosine similarity (SentenceBERT_CS) and the Euclidean distance (SentenceBERT_ED).

InferSent [13] relies on GloVe vectors [59] as pre-trained word embeddings for the sentence pair. The embeddings are then passed through RNN encoder layers to obtain fixed-length vector representations for each sentence. Also in this case both the cosine similarity (InferSent_CS) and the Euclidean distance (InferSent_ED) are considered to contrast the generated and the reference summary.

Universal Sentence Encoder (USE) [10] employs transformer encoders to generate context-aware representations of words within a sentence by levering the self-attention mechanism. Both USE_CS (cosine similarity-based) and USE_ED (Euclidean distance-based) are considered.

CodeT5+_CS [84] exploits CodeT5+, a model pre-trained on code and natural language. We use the CodeT5+$_{base}$ variant ($\sim$220M

trainable parameters) to compute the cosine similarity between the embeddings of the generated summary and the code to document. CodeT5+_CS acts as a further baseline for SIDE which has been fully trained via contrastive learning.

4.3 Analysis Methodology

In the following, we describe the study analysis methodology. The whole analysis has been performed using the *R* [62] statistical environment. For statistical tests, we assume a significance level $\alpha = 0.05$.

To address RQ_1, we first analyze the correlation between different summary evaluation metrics. To avoid being constrained with linear relationships only, we leverage the non-parametric, Spearman's rank correlation [14]. To show the correlation, we create a visual overview of correlation among metrics using the *varclus* function part of the *R Hmisc* package [30]. The output of this procedure is a hierarchical clustering of variables (*i.e.,* our metrics), producing a dendogram (*i.e.,* the clustering tree) visualizing correlated metrics. Note that we do not use Spearman's correlation to select uncorrelated variables but, as explained below, the *redun* procedure which allow accounting for multicollinearity other than collinearity.

We complement the correlation analysis with a Principal Component Analysis (PCA) [38]. PCA leverages Singular Value Decomposition (SVD) to describe the underlying data variance and covariance expressed as linear combinations of the considered variables.

To avoid having the results of the PCA being affected by collinearity, we run a variable selection procedure using the *redun* function from the *Hmisc* package [30]. This function performs a stepwise removal of the independent variables determining how well each variable can be predicted by the remaining ones. The process starts by first removing the most predictable variable and continuing subsequently until no variable among the predictors can be predicted with a given (adjusted) R^2, which we set equal to 0.8. Then, before applying PCA, we re-scale the variables in the range [0,1] using a min-max re-scaling.

The PCA on the selected variables returns the eigenvectors related to each independent component, where the (absolute) values in correspondence of each metric indicate the importance of the metric for that component. Moreover, it returns the standard deviation captured by each principal component, which indicates the importance of the component itself. The PCA has been performed using the *prcomp* function of the *R* (default) *stats* package.

To address RQ_2, we determine the extent to which different summary quality metrics can complement each other to explain extrinsic quality indicators provided by users. Given the (ordinal) scale of the dependent variables, and given that we cannot assume a linear relation between independent and dependent variables, this analysis has been performed by employing a multivariate logistic regression. A conventional logistic regression model is not suitable for our analysis, because the dependent variables are not dichotomous but, rather, expressed on a 0-5 ordinal scale [55], or on a 1-100 scale (in the case of DA score). Therefore, we use an ordered logistic regression. Given the *l* levels of an ordinal variable, the ordinal logistic regression models ℓ different logits as:

$$ ln\left(\frac{P(Y \leq \ell)}{P(Y > \ell)}\right) = \xi_\ell - \eta_1 X_1 - \cdots - \eta_n X_n \tag{1} $$

where Y is the model's dependent variable, X_i the independent variables, η_i their coefficients (estimates), and ξ_ℓ the intercept.

We use the *polr* function from the *MASS* package [82]. The interpretation of the model estimates is similar to the ones of logistic regression, with the difference that, given an estimate η_i, the OR=e^{η_i} indicates what are the increased odds of a unitary value increment for the dependent variable given a unitary increase of an independent variable. The application of the ordered logistic regression follows three subsequent steps, *i.e.,* (i) variable selection, (ii) variable re-scaling, and (iii) model building.

For the variable selection, we use the output of the *redun* procedure used in RQ_1. Then, given the selected variables, we re-scale them in the same range of the dependent variable being modeled. This makes the interpretation and comparison of the model's ORs easier. As for the dependent variables, we leave them as they are (*i.e.,* on a scale from 0 to 5 for *conciseness*, *fluency*, and *content adequacy*, and on a scale from 1-100 for *DA score*).

Finally, we apply the *polr* procedure, relating each dependent variable to all independent variables that were left after the *redun* feature selection. We report (i) fitting diagnostics (in particular the Akaike Information Criterion - AIC value), (ii) independent variables estimates, OR, and significance *p*-value adjusted, due to multiple factors, with the Benjamini-Hochberg correction [9].

5 RESULTS DISCUSSION

RQ_1: Correlation of different metrics to evaluate the quality of source code summarization. Fig. 1 depicts the hierarchical clustering of the independent variables output of the *varclus* procedure. Note that the values on the *x*-axis show the square of the Spearman's correlation (*e.g.,* a value of 0.64 indicates a correlation of $0.8^2 = 0.64$).

The top-level fork of the tree splits the metrics into two different families. The first groups all conventional metrics capturing the similarity between a generated and a reference summary, while the second includes the three metrics focusing on the relevance of the generated comment for the documented code (*i.e.,* CodeT5-plus_CS, c_coeff, and SIDE). This is a first indication of the fact that the two families of metrics focus on different aspects of summary quality.

Looking inside the sub-tree composed of conventional metrics, BLEU-1 is not correlated with all others, not even with the other BLEU variants. While this may look counter-intuitive, BLEU-1 focuses on single-word overlap, while its other variants look at *n*-grams overlap. This usually results in much higher values for the BLEU-1 as compared to BLEU-2, BLEU-3, and BLEU-4.

The deeper we go into the tree, the more cohesive the clusters of metrics that can be observed. The three metrics looking at the documented code (*i.e.,* CodeT5-plus_CS, c_coeff, and SIDE) are very high in the dendogram indicating that, while they stay apart from the other conventional metrics, the correlation among them is small, likely due to the different underlying solutions used to assess the relevance of a generated summary for a given code.
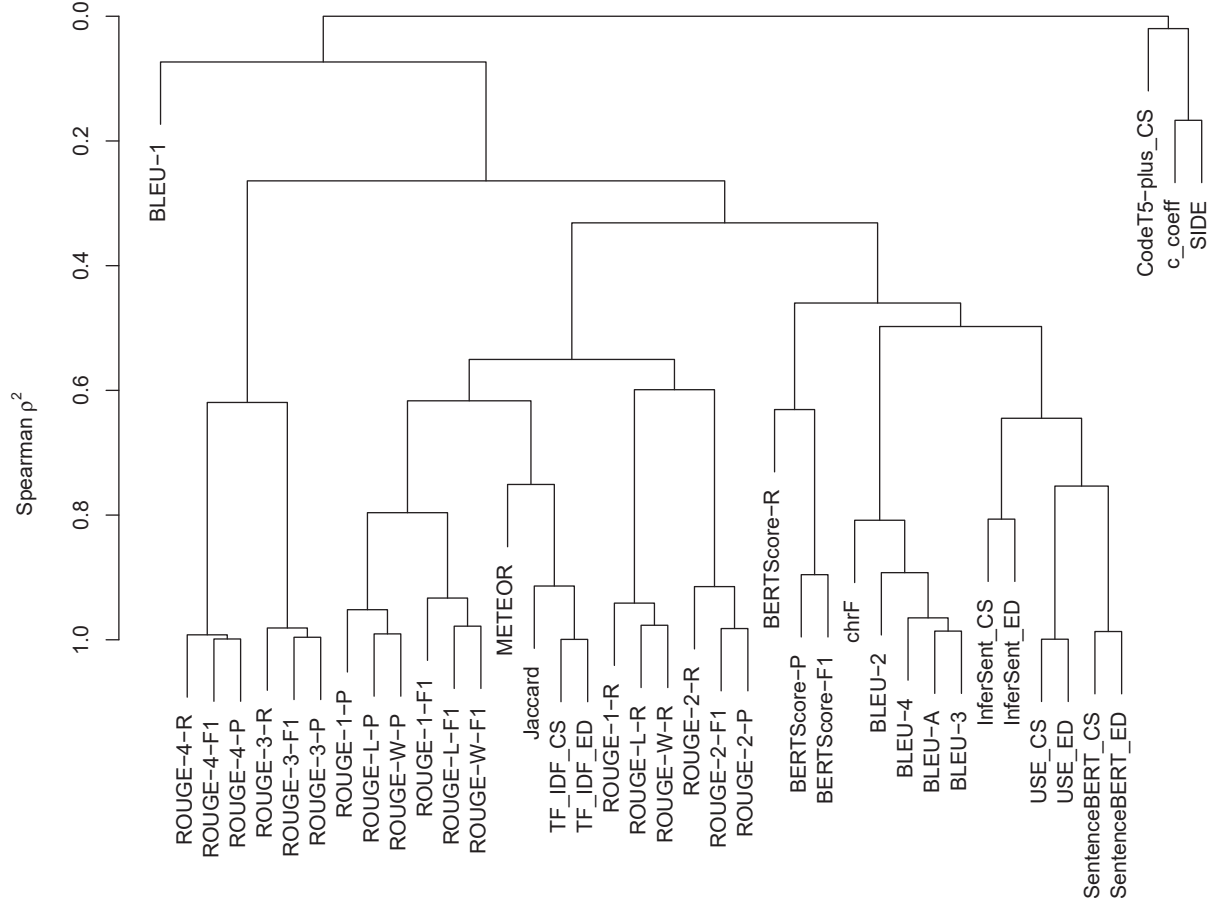
Figure 1: RQ$_1$ - Spearman's correlation clustering on the independent variables with varclus

Table 1: RQ$_1$ - PCA for the evaluated metrics

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10
Prop. of Variance	0.55	0.16	0.08	0.07	0.04	0.03	0.02	0.02	0.01	0.01
Cumulative Prop.	0.55	0.71	0.80	0.87	0.91	0.94	0.96	0.98	0.99	1.00
BLEU-1	0.26	0.24	-0.44	0.11	-0.50	0.13	-0.44	0.45	-0.09	-0.09
BERTScore-R	0.47	0.15	-0.24	0.25	-0.18	-0.18	0.74	-0.07	0.10	-0.09
SentenceBERT_CS	0.38	-0.10	0.04	0.17	0.22	-0.68	-0.37	-0.18	-0.32	-0.18
InferSent_CS	0.23	-0.01	-0.01	0.06	0.09	-0.21	-0.17	0.10	0.54	0.75
ROUGE-1-P	0.44	0.07	-0.10	0.21	0.33	0.63	-0.17	-0.43	-0.14	0.11
ROUGE-4-R	0.41	0.26	0.38	-0.73	-0.25	0.01	-0.02	-0.14	-0.05	0.02
ROUGE-W-R	0.30	-0.04	0.36	0.06	0.44	0.15	0.05	0.64	0.19	-0.32
c_coeff	0.13	-0.47	0.53	0.39	-0.54	0.13	-0.07	-0.10	0.06	-0.01
CodeT5-plus_CS	-0.00	-0.02	-0.17	-0.08	-0.03	-0.02	-0.23	-0.34	0.72	-0.53
SIDE	0.20	-0.79	-0.39	-0.39	0.03	0.07	0.08	0.10	-0.07	0.02

The next step in our data analysis is to use the PCA to identify the orthogonal components in the performed measurements. To run it, we first selected the metrics to consider using the *redun* function as explained in Section 4.3.

This process resulted only in the 10 metrics listed in Table 1, already providing a first indication of the fact that the other metrics considered in our study are redundant and highly correlated with at least one of the 10 selected. The latter include (i) four

words/characters-overlap based metrics capturing the similarity between the generated and the reference summary (*i.e.,* BLEU-1, ROUGE-1-P, ROUGE-4-R, and ROUGE-W-R); (ii) three embedding-based metrics also focusing on the similarity between the generated and the reference summary (*i.e.,* BERTScore-R, SentenceBERT_CS, and InferSent_CS); and (iii) all three metrics focusing on the similarity/relevance of the generated summary to the documented code (*i.e.,* c_coeff, CodeT5-plus_CS, and SIDE).

Table 1 reports the results of the PCA, with 10 principal components (PCs) identified. The first row indicates the proportion of variance captured by each PC. The higher such a value, the higher the variance in the dataset described by the PC. The second row reports the cumulative proportion of variance when considering the first *n* PCs. For example, by just using five PCs, it is possible to capture 91% of the variance in the data, *i.e.,* 55% (PC1) + 16% (PC2) + 8% (PC3) + 7% (PC4) + 4% (PC5). The remaining 10 rows in Table 1 show the importance of each metric for each PC: The higher the absolute value, the higher the metric's contribution to that PC. For each PC, we highlight with a dark background the metric(s) contributing the most to it. In particular, we highlight the metric with the highest absolute value in the eigenvector and all

those close to it (at most -5% within the absolute value). The "5%" choice is arbitrary and just meant to simplify the results' visualization and discussion highlighting all metrics similarly contributing to a PC. PC1 is captured by conventional metrics looking for textual similarity between the generated and the reference summary (mostly BERTScore and ROUGE). PC2 is instead exclusively captured by SIDE, suggesting the importance of including metrics considering the code to document when measuring summary quality.

The second-highest value for PC2 is obtained by using c_coeff, but it is substantially smaller than the one associated with SIDE (0.47 *vs* 0.79). For PC3, c_coeff exhibits instead the highest eigenvector.

The analysis conducted so far indicates that by just considering three metrics (*e.g.,* BERTScore-R, SIDE, and c_coeff) it is possible to capture 80% of the variance in the data.

Conventional metrics are also associated with PC4 (ROUGE-4-R), with c_coeff being instead the one capturing PC5. Basically, among the top-5 PCs, 3 are mostly captured by metrics considering the code to document in the equation.

In summary, RQ$_1$ findings show that conventional metrics looking at the similarity between the generated and the reference summary do not correlate with those looking at the relevance/similarity of the generated summary for/with the documented code. Also, they only capture part of the variance in the data.

This only indicates that the metrics considering the documented code (*e.g.,* SIDE, or c_coeff) capture orthogonal information as compared to the conventional ones, not whether or not they better capture code summary quality as perceived by developers.

RQ$_2$: Contribution of different code summarization metrics in explaining user-based evaluations. Before discussing how various metrics to assess source code summarization impact user-based evaluations, we must anticipate that, in this section will focus exclusively on the outcomes achieved when training SIDE using hard-negatives. Additional information about the conducted ablation study is provided in Section 5.2.

Table 2 reports the results of the ordered logistic regression for each code summary quality attribute (dependent variables) manually evaluated by developers (*i.e., DA score, content adequacy, conciseness,* and *fluency*). We focus our discussion on the Odds Ratios (ORs) of the statistically significant *p*-values (<0.05), which are the ones highlighted with a black background. The interpretation of the ORs is as follows: they indicate the odds of a unitary increment in the dependent variable given a unitary increment in the independent variable. For example, if we look at the OR of SIDE when considering the *content adequacy* as a dependent variable (1.6265), it indicates ~62% higher odds of observing a unitary increment in the *content adequacy* as perceived by developers for each unitary increment of the SIDE value. Remember that both independent and dependent variables have been normalized on the same scale: In the case of *DA score*, all variables are in the 1-100 range, while the three other variables are in the range 0-5, since we followed the scales used in the developers' evaluation. This also explains the lower ORs in the *DA score* table. Indeed, a unitary increase on a 1-100 range has a different magnitude as compared to a unitary increase on a 0-5 scale.

Table 2: RQ$_2$ - Ordered logistic regression model

Overall DA Score [68]					
Metric	OR	Value	Std. Error	*t*-value	*p*-value
BLEU-1	0.9990	-0.0010	0.0017	-0.5822	0.6222
BERTScore-R	1.0029	0.0029	0.0023	1.2634	0.2943
SentenceBERT_CS	1.0057	0.0057	0.0020	2.8882	0.0100
InferNet_CS	0.9980	-0.0020	0.0025	-0.8064	0.5250
ROUGE-1-P	1.0058	0.0058	0.0018	3.2552	0.0033
ROUGE-4-R	1.0024	0.0024	0.0011	2.1987	0.04667
ROUGE-W-R	0.9996	-0.0004	0.0017	-0.2485	0.8040
c_coeff	1.0143	0.0142	0.0012	11.4549	<0.0001
CodeT5-plus_CS	1.0044	0.0044	0.0017	2.5383	0.0220
SIDE	1.0205	0.0203	0.0017	11.7029	<0.0001

Content Adequacy [68]					
Metric	OR	Value	Std. Error	*t*-value	*p*-value
BLEU-1	0.9672	-0.0333	0.0347	-0.9615	0.4200
BERTScore-R	1.0525	0.0512	0.0468	1.0927	0.3929
SentenceBERT_CS	1.1121	0.1063	0.0406	2.6172	0.0225
InferNet_CS	1.0000	0.0000	0.0517	-0.0004	1.0000
ROUGE-1-P	1.0943	0.0901	0.0364	2.4733	0.0260
ROUGE-4-R	0.9914	-0.0086	0.0226	-0.3814	0.7811
ROUGE-W-R	1.0437	0.0428	0.0351	1.2202	0.3700
c_coeff	1.3729	0.3169	0.0255	12.4350	<0.0001
CodeT5-plus_CS	1.1413	0.1322	0.0358	3.6877	<0.0001
SIDE	1.6265	0.4864	0.0366	13.2942	<0.0001

Conciseness [68]					
Metric	OR	Value	Std. Error	*t*-value	*p*-value
BLEU-1	1.0656	0.0636	0.0342	1.8576	0.1575
BERTScore-R	1.0160	0.0159	0.0470	0.3375	0.8770
SentenceBERT_CS	1.0587	0.0571	0.0403	1.4163	0.2617
InferNet_CS	1.0079	0.0079	0.0511	0.1547	0.8770
ROUGE-1-P	1.0075	0.0075	0.0359	0.2091	0.8770
ROUGE-4-R	0.9937	-0.0063	0.0228	-0.2766	0.8770
ROUGE-W-R	1.0954	0.0911	0.0346	2.6352	0.0267
c_coeff	1.2433	0.2178	0.0254	8.5798	<0.0001
CodeT5-plus_CS	1.0630	0.0611	0.0355	1.7212	0.1700
SIDE	1.3844	0.3253	0.0354	9.1835	<0.0001

Fluency [68]					
Metric	OR	Value	Std. Error	*t*-value	*p*-value
BLEU-1	1.0650	0.0629	0.0345	1.8243	0.2267
BERTScore-R	1.0559	0.0544	0.0467	1.1638	0.4600
SentenceBERT_CS	0.9958	-0.0042	0.0404	-0.1043	0.9170
InferNet_CS	1.0543	0.0529	0.0515	1.0272	0.4600
ROUGE-1-P	1.0334	0.0329	0.0365	0.9004	0.4600
ROUGE-4-R	1.0304	0.0299	0.0229	1.3061	0.4600
ROUGE-W-R	0.9672	-0.0334	0.0347	-0.9603	0.4600
c_coeff	1.1680	0.1553	0.0253	6.1437	<0.0001
CodeT5-plus_CS	1.0249	0.0246	0.0357	0.6902	0.5444
SIDE	1.2826	0.2489	0.0359	6.9359	<0.0001

A first observation that can be made is that SIDE is the metric having the highest OR independently from the considered dependent variable. Also, only five metrics obtained a significant *p*-value for at least one dependent variable (SIDE is always among them).

The "overall" quality of the summary (*i.e., DA score*) and its *content adequacy* as perceived by developers is captured by five metrics: SIDE, c_coeff, CodeT5-plus_CS, SentenceBERT_CS, and ROUGE-1-P, with SIDE and c_coeff having the highest and second-highest OR, respectively, for both dependent variables. Still, conventional metrics looking at the similarity between a generated and

a reference summary also play a role in capturing code summary quality as assessed through *DA score* and *content adequacy*.

When moving to the summary *conciseness* and *fluency*, SIDE and c_coeff confirm their strong relationship with the human assessment (with SIDE being the best in the class), while the conventional metrics struggle in capturing these quality aspects, with the only exception being the ROUGE-W-R when looking at the *conciseness*.

Still, the OR for SIDE is 1.3844 (*i.e.,* 38% higher odds of a unitary increment in the human assessment of summary conciseness for a one unit increase of SIDE), while for ROUGE-W-R is substantially lower (1.0954). The reason may lie behind the different assumptions made by the two families of metrics. To explain this point, let us take the example of the *conciseness* quality attribute. Metrics such as SIDE might learn during training that longer methods may need longer summaries and, thus, that very long summaries for short methods may not be appropriate. This is likely aligned with what a developer would think when judging the *conciseness* of a summary. Similarly, c_coeff is likely to produce low values when a very long summary is associated with a short method, since the denominator of the used formula (*i.e.,* the number of terms in the summary) increases. Differently, conventional metrics would judge a generated summary of high quality if it is similar to the reference one. Thus, if the reference is not concise and the generated summary is similar to the reference (thus, not concise as well), the assessed quality will be high and not aligned with the developers' perception.

5.1 Qualitative Analysis

To provide further insights into the complementarity of the two families of metrics, we selected four qualitative examples to discuss (Fig. 2). Each example features (i) the Java method for which a summary was automatically generated; (ii) the target summary (*i.e.,* the one written by developers); (iii) the generated summary; and (iv) the quality scores assigned by the seven state-of-the-art (SOTA) metrics selected via the *redun* procedure (*i.e.,* BLEU-1, BERTScore-R, etc.), by our SIDE metric, and in the human assessment. We selected two examples in which SIDE provides an indication aligned with the developers' perception while the SOTA metrics fail, and two examples in which the opposite scenario occurs.

The first example ❶ is a typical scenario in which the reference summary is of low quality since it actually documents the fact that the method is "*not yet documented*".

The generated summary, instead, provides a good description of the method accordingly to the developers and, thus, is substantially different from the reference. This results in low values for the SOTA metrics, while SIDE agrees with the human assessment (*DA score* = 0.88), assigning the generated summary 0.91/1.00.

In the second ❷ and third ❸ examples, SIDE is instead in disagreement with developers. In ❷, it assigns a high score (0.87) to a correct but very generic summary, while the conventional metrics are able to assess the low quality of the summary thanks to the presence of additional material present in the reference but lacking in the generated summary (*i.e.,* the example).

In ❸, SIDE assigns a quite low score to the generated summary, despite it being well-judged by humans. SOTA metrics can instead exploit the word overlap between the generated and the reference summary, assessing the summary with higher scores.

Figure 2: Examples from Roy *et al.* [68] dataset

Finally, ❹ shows a low-quality generated summary (as assessed by developers) which is scored with the maximum scores by state-of-the-art metrics, since being identical to the reference one. SIDE is instead able to identify the summary as low quality (0.15) by contrasting it with the documented code.

In summary, our analysis suggests that a more comprehensive evaluation framework must be considered when assessing the quality of automatically generated summaries: Conventional metrics looking at the similarity between the generated and the reference

summary are not enough and should be augmented by metrics judging the suitability of a generated summary for the code it documents, similarly to what SIDE does. The latter is the metric more related to the human judgment of code summary quality.

5.2 Ablation Study - Impact of Hard-negatives

Due to the lack of space, we present in our replication package [2] a table reporting the results of the ordered logistic regression model for each dependent variable including two variants of SIDE: One trained using hard-negatives (*i.e.,* the one related to the results discussed as of now) and one trained without hard-negatives. In short, the impact of including or not hard-negatives in the training set is negligible, and the all findings of our study are not changed by such a design choice (*e.g.,* no impact on which ORs are statistically significant nor on their magnitude). This result may be due to several reasons. First, due to our definition of hard-negatives (*i.e.,* comments documenting at most 25% of the method's statements), we managed to create hard-negatives for only ~15% of the methods in our dataset. Thus, it is possible that a higher coverage is needed to observe any influence on the achieved results. Second, our definition of hard-negatives may be suboptimal, opening to better approaches aimed at creating hard-negative samples for contrastive learning applied to software engineering tasks.

6 THREATS TO VALIDITY

Construct validity threats. As also detailed in the work of Roy *et al.* [68], the considered dependent variables reflect the user's assessment of a summary from different perspectives (*conciseness, fluency, content adequacy,* plus an *overall assessment*). We are aware that such an assessment could suffer from the assessor's error or subjectiveness. Also, we acknowledge that the fine-tuning dataset used to train SIDE may feature some low-quality summaries which may partially hinder its ability to assess whether a natural language text represents a suitable summary for a given code. However, our empirical evaluation showed that SIDE is still the metric better capturing human assessment of code summary quality, despite this potential source of noise in its training set.

Internal validity threats. These primarily pertain to the configurations applied during metric computation and evaluation, such as the decision to utilize cosine similarity. In addition, we recognize the potential value in exploring alternative hyperparameter settings (see Section 3.4) when developing SIDE.

Also, adopting CodeT5+ as the representative state-of-the-art pre-trained model for generating $\langle method, documentation \rangle$ embeddings, and subsequently using cosine similarity for comparison, might not reflect the best strategy for the discussed task.

Finally, the negligible impact of including hard-negative samples in the training set may be due to the design we employed to mine hard-negatives (*e.g.,* see the choice of the 25% coverage to identify hard-negative comments) and further studies are needed to investigate alternative solutions.

Conclusion validity threats. RQ_1 findings show both the metrics correlation, computed using a non-parametric procedure, and the PCA. RQ_2 is based on an ordered logistic regression, suitable for variables in ordinal scale. Note that statistically significant $p-$values are small enough to be unaffected by fishing and error rate.

External validity threats. The main limitation of our study is that it relies on human evaluations from a single dataset [68]. Nevertheless, such a dataset is large and features a total of 6,253 (5,201 used in our study) evaluations from 226 different participants.

Concerning the independent variables (evaluation metrics), our study attempted to consider several metrics having a different nature, some of which were used in previous work [28, 68, 76].

7 CONCLUSIONS

State-of-the-art metrics used to evaluate code summarization techniques only assess the similarity between the generated summary and the reference one written by developers. This implies that a generated summary, while suitable for the code to document, may be different from the reference one (*e.g.,* a low-quality one), and therefore scored low by these metrics. We argue that the code to document must also be considered in the equation when scoring automatically generated summaries, to assess whether they are suitable for such a code independently from their similarity with the reference summary. To capture this information, we presented SIDE, a metric exploiting contrastive learning to learn characteristics of suitable and unsuitable summaries for a given code.

We run a study involving 40 metrics (including SIDE) investigating their complementarity and the extent to which they correlate with humans' evaluation of summary quality. Our findings highlight the high complementarity between SIDE and the metrics focusing on the similarity between the generated and the reference summary. SIDE is also the metric that better describes humans' assessment of summary quality. Also, we show that the proposed contrastive learning-based metric captures the suitability of a summary for a given code better than simpler solutions. Further evaluations of code summarization methods should incorporate a broader assessment framework that includes both conventional metrics—*e.g.,* the similarity between generated and reference summaries—and SIDE-like metrics, evaluating the suitability of the generated summary for the code being documented. In addition, the findings of this study provide several opportunities for researchers, particularly in the area of program comprehension. To this end, a key application of SIDE would involve the identification of inconsistencies between code comments with respect to the documented code (*e.g.,* a method), as this could significantly improve the quality of software systems once the code and its documentation are realigned.

8 DATA AVAILABILITY

We provide [2]: (i) the datasets used to train and evaluate SIDE; (ii) the code for reproducing the experiments; (iii) the code for computing the *evaluation* metrics; (iv) R scripts for conducting the statistical analysis; (v) additional results not featured in the paper (*e.g.,* correlation matrices); and (vi) a script allowing other researchers to compute SIDE for pairs of $\langle method, summary \rangle$.

REFERENCES

[1] [n. d.]. CC-News Dataset. https://commoncrawl.org/2016/10/news-dataset-available/.

[2] [n. d.]. Replication Package https://github.com/antonio-mastropaolo/code-summarization-metric/tree/main.

[3] [n. d.]. Stories Dataset. https://github.com/tensorflow/models/tree/archive/research/lm_commonsense#1-download-data-files.

[4] Emad Aghajani, Gabriele Bavota, Mario Linares-Vásquez, and Michele Lanza. 2019. Automated Documentation of Android Apps. *IEEE Transactions on Software Engineering* 45, 1 (2019), 1–13.

[5] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT.* 2655–2668.

[6] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *International Conference on Machine Learning (ICML).*

[7] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[8] 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.* Association for Computational Linguistics, Ann Arbor, Michigan, 65–72. https://aclanthology.org/W05-0909

[9] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995).

[10] 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).

[11] Huanchao Chen, Yuan Huang, Zhiyong Liu, Xiangping Chen, Fan Zhou, and Xiaonan Luo. 2019. Automatically detecting the scopes of source code comments. *Journal of Systems and Software, JSS* 153 (2019), 45–63.

[12] Sumit Chopra, Raia Hadsell, and Yann LeCun. 2005. Learning a similarity metric discriminatively, with application to face verification. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, Vol. 1. IEEE, 539–546.

[13] 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).

[14] W. J. Conover. 1998. *Practical Nonparametric Statistics* (3rd edition ed.). Wiley.

[15] Ward Cunningham. 1992. The WyCash portfolio management system. *ACM Sigplan Oops Messenger* 4, 2 (1992), 29–30.

[16] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. 2005. A Study of the Documentation Essential to Software Maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting &Amp; Designing for Pervasive Information (SIGDOC '05).* 68–75.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT.* 4171–4186.

[18] Beat Fluri, Michael Wursch, and Harald C. Gall. 2007. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In *14th Working Conference on Reverse Engineering, WCRE.* 70–79.

[19] Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C. Gall. 2009. Analyzing the Co-evolution of Comments and Source Code. *Software Quality Journal, SQJ* 17, 4 (2009), 367–394.

[20] Haifeng Gao, Bin Dong, Yongwei Zhangand Tianxiong Xiao, Shanshan Jiang, and Yuan Dong. 2021. An Efficient Method of Supervised Contrastive Learning for Natural Language Understanding. In *7th International Conference on Computer and Communications (ICCC).* 1698–1704.

[21] Aaron Gokaslan and Vanya Cohen. 2019. OpenWebText Corpus. http://Skylion007.github.io/OpenWebTextCorpus.

[22] Yvette Graham, Timothy Baldwin, Alistair Moffat, and Justin Zobel. 2013. Continuous measurement scales in human evaluation of machine translation. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse.* 33–41.

[23] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2021. Code to Comment "Translation": Data, Metrics, Baselining & Evaluation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering.* 746–757.

[24] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021.*

[25] Raia Hadsell, Sumit Chopra, and Yann LeCun. 2006. Dimensionality Reduction by Learning an Invariant Mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR.* 1735–1742.

[26] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *17th Working Conference on Reverse Engineering, WCRE.* 35–44.

[27] John M Hancock. 2004. Jaccard distance (Jaccard index, Jaccard similarity coefficient). *Dictionary of Bioinformatics and Computational Biology* (2004).

[28] 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 (ICPC '22).* 36–47.

[29] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved Automatic Summarization of Subroutines via Attention to File Context. arXiv:2004.04881 [cs.SE]

[30] Frank E Harrell Jr, with contributions from Charles Dupont, and many others. 2020. *Hmisc: Harrell Miscellaneous.* https://CRAN.R-project.org/package=Hmisc R package version 4.3-1.

[31] Xing Hu, Qiuyuan Chen, Haoye Wang, Xin Xia, David Lo, and Thomas Zimmermann. 2022. Correlating Automated and Human Evaluation of Code Documentation Generation Quality. 31, 4 (2022).

[32] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Springer Empirical Software Engineering, EMSE* 25, 3 (2020), 2179–2217.

[33] Junjie Huang, Duyu Tang, Wanjun Zhong, Shuai Lu, Linjun Shou, Ming Gong, Daxin Jiang, and Nan Duan. 2021. WhiteningBERT: An Easy Unsupervised Sentence Embedding Approach. In *Findings of the Association for Computational Linguistics: EMNLP.* 238–244.

[34] Yuan Huang, Shaohao Huang, Huanchao Chen, Xiangping Chen, Zibin Zheng, Xiapu Luo, Nan Jia, Xinyu Hu, and Xiaocong Zhou. 2020. Towards automatically generating block comments for code snippets. *Inf. Softw. Technol.* 127 (2020).

[35] Peter J Huber. 1992. Robust estimation of a location parameter. In *Breakthroughs in statistics: Methodology and distribution.* 492–518.

[36] 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).

[37] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proc. of the 54th Annual Meeting of the Association for Computational Linguistics.* 2073–2083.

[38] Ian T. Jolliffe. 1986. *Principal Component Analysis.* Springer.

[39] Ninus Khamis, René Witte, and Juergen Rilling. 2010. Automatic Quality Assessment of Source Code Comments: The JavadocMiner. In *Natural Language Processing and Information Systems*, Christina J. Hopfe, Yacine Rezgui, Elisabeth Métais, Alun Preece, and Haijiang Li (Eds.). 68–79.

[40] Solomon Kullback. 1997. *Information theory and statistics.* Courier Corporation.

[41] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *41st IEEE/ACM International Conference on Software Engineering, ICSE.* 795–806.

[42] 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.

[43] Alexander LeClair and Collin McMillan. 2019. Recommendations for Datasets for Source Code Summarization. In *Proc. of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Vol. 1.* 3931–3937.

[44] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *58th Annual Meeting of the Association for Computational Linguistics, ACL.* 7871–7880.

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

[46] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk. 2015. How do Developers Document Database Usages in Source Code?. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE.* 36–41.

[47] 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. *CoRR* abs/1907.11692 (2019). https://doi.org/10.48550/arXiv.1907.11692

[48] Zhiyong Liu, Huanchao Chen, Xiangping Chen, Xiaonan Luo, and Fan Zhou. 2018. Automatic Detection of Outdated Comments During Code Changes. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 01. 154–163.

[49] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR.*

[50] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021.

Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[51] Alejandro Martín, Javier Huertas-Tato, Álvaro Huertas-García, Guillermo Villar-Rodríguez, and David Camacho. 2022. FacTeR-Check: Semi-automated fact-checking through semantic similarity and natural language inference. *Knowl. Based Syst.* 251 (2022), 109265.

[52] P. W. McBurney and C. McMillan. 2016. Automatic Source Code Summarization of Context for Java Methods. *IEEE Transactions on Software Engineering, TSE* 42, 2 (2016), 103–119.

[53] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 23–32.

[54] Hyun Oh Song, Yu Xiang, Stefanie Jegelka, and Silvio Savarese. 2016. Deep metric learning via lifted structured feature embedding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4004–4012.

[55] A. N. Oppenheim. 1992. *Questionnaire Design, Interviewing and Attitude Measurement.* Pinter Publishers.

[56] Lin Pan, Chung-Wei Hang, Avirup Sil, and Saloni Potdar. 2022. Improved Text Classification via Contrastive Adversarial Training. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022.* 11130–11138.

[57] Lin Pan, Chung-Wei Hang, Avirup Sil, and Saloni Potdar. 2022. Improved text classification via contrastive adversarial training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 11130–11138.

[58] 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.

[59] 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.

[60] Maja Popović. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the tenth workshop on statistical machine translation*. 392–395.

[61] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 91–100.

[62] R Core Team. 2020. *R: A Language and Environment for Statistical Computing.* https://www.R-project.org/

[63] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67.

[64] M. M. Rahman, C. K. Roy, and I. Keivanloo. 2015. Recommending insightful comments for source code using crowdsourced knowledge. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM*. 81–90.

[65] Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proc. of the 1st instructional conf. on machine learning*, Vol. 242. 29–48.

[66] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[67] Paige Rodeghero, Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Detecting user story information in developer-client conversations to generate extractive summaries. In *39th IEEE/ACM International Conference on Software Engineering, ICSE*. 49–59.

[68] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing Automatic Evaluation Metrics for Code Summarization Tasks. In *29th ACM Joint Meeting on European Software Engineering Conference and the ACM/SIGSOFT Symposium on the Foundations of Software Engineering, ESEC-FSE.* 1105–1116.

[69] Hadeel Saadany and Constantin Orasan. 2021. BLEU, METEOR, BERTScore: Evaluation of Metrics Performance in Assessing Critical Translation Errors in Sentiment-oriented Text. *CoRR* abs/2109.14250 (2021). https://doi.org/10.48550/arXiv.2109.14250

[70] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 815–823.

[71] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2020. MPNet: Masked and Permuted Pre-training for Language Understanding. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems*.

[72] D. Spinellis. 2010. Code Documentation. *IEEE Software* 27, 4 (2010), 18–19.

[73] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 101–110.

[74] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A Human Study of Comprehension and Code Summarization. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)*. 2–13.

[75] D. Steidl, B. Hummel, and E. Juergens. 2013. Quality analysis of source code comments. In *Proc. of the 21st IEEE International Conference on Program Comprehension (ICPC)*. 83–92.

[76] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*. 83–92.

[77] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*icomment: bugs or bad comments?*/. In *21st ACM Symposium on Operating Systems Principles, SOSP*. 145–158.

[78] Lin Tan, Ding Yuan, and Yuanyuan Zhou. 2007. HotComments: How to Make Program Comments More Useful?. In *Proceedings of HotOS'07: 11th Workshop on Hot Topics in Operating Systems*, Galen C. Hunt (Ed.). USENIX Association.

[79] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 260–269.

[80] T. Tenny. 1988. Program Readability: Procedures Versus Comments. *IEEE Trans. Softw. Eng.* 14, 9 (Sept. 1988), 1271–1279.

[81] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *30th Advances in Neural Information Processing Systems*. 5998–6008.

[82] W. N. Venables and B. D. Ripley. 2002. *Modern Applied Statistics with S* (fourth ed.). Springer, New York. http://www.stats.ox.ac.uk/pub/MASS4 ISBN 0-387-95457-0.

[83] Deze Wang, Yong Guo, Wei Dong, Zhiming Wang, Haoran Liu, and Shanshan Li. 2019. Deep Code-Comment Understanding and Assessment. *IEEE Access* 7 (2019), 174200–174209.

[84] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv:2305.07922* [cs.CL]

[85] Feng Wei, Zhenbo Chen, Zhenghong Hao, Fengxin Yang, Hua Wei, Bing Han, and Sheng Guo. 2022. Semi-supervised clustering with contrastive learning for discovering new intents. *arXiv preprint arXiv:2201.07604* (2022).

[86] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *27th International Conference on Program Comprehension, ICPC*. 53–64.

[87] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. CloCom: Mining existing source code for automatic comment generation. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. 380–389.

[88] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering*. 562–567.

[89] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. 1981. The Effect of Modularization and Comments on Program Comprehension. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. 215–223.

[90] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering, TSE* 44, 10 (2018), 951–976.

[91] Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, Michael Witbrock, and Vadim Sheinin. 2018. Graph2seq: Graph to sequence learning with attention-based neural networks. *arXiv preprint arXiv:1804.00823* (2018).

[92] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems*. 5754–5764.

[93] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *42nd IEEE/ACM International Conference on Software Engineering ICSE*. 1385–1397.

[94] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. 2020. PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization. In *37th International Conference on Machine Learning, ICML*. 11328–11339.

[95] 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).

[96] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. BERTScore: Evaluating Text Generation with BERT. In *8th International Conference on Learning Representations, ICLR*.

[97] Yukun Zhu, Ryan Kiros, Richard S. Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books. In *2015 IEEE International Conference on Computer Vision, ICCV*. 19–27.