Towards Summarizing Code Snippets Using Pre-Trained Transformers

Antonio Mastropaolo
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

Matteo Ciniselli
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

Luca Pascarella
Center for Project-Based Learning,
ETH Zurich
Zurich, Switzerland, CH

Rosalia Tufano
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

Emad Aghajani
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

Gabriele Bavota
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

ABSTRACT

When comprehending code, a helping hand may come from the natural language comments documenting it that, unfortunately, are not always there. To support developers in such a scenario, several techniques have been presented to automatically generate natural language summaries for a given code. Most recent approaches exploit deep learning (DL) to automatically document classes or functions, while little effort has been devoted to more fine-grained documentation (*e.g.,* documenting code snippets or even a single statement). Such a design choice is dictated by the availability of training data: For example, in the case of Java, it is easy to create datasets composed of pairs <*method*, *javadoc*> that can be fed to DL models to teach them how to summarize a method. Such a comment-to-code linking is instead non-trivial when it comes to inner comments documenting a few statements. In this work, we take all the steps needed to train a DL model to automatically document code snippets. First, we manually built a dataset featuring 6.6k comments that have been (i) classified based on their type (*e.g.,* code summary, TODO), and (ii) linked to the code statements they document. Second, we used such a dataset to train a multi-task DL model taking as input a comment and being able to (i) classify whether it represents a "code summary" or not, and (ii) link it to the code statements it documents. Our model identifies code summaries with 84% accuracy and is able to link them to the documented lines of code with recall and precision higher than 80%. Third, we run this model on 10k projects, identifying and linking code summaries to the documented code. This unlocked the possibility of building a large-scale dataset of documented code snippets that have then been used to train a new DL model able to automatically document code snippets. A comparison with state-of-the-art baselines shows the superiority of the proposed approach, which however, is still far from representing an accurate solution for snippet summarization.

ICPC '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0586-1/24/04.
https://doi.org/10.1145/3643916.3644400

CCS CONCEPTS

• **Software and its engineering** → **Extra-functional properties**.

KEYWORDS

Software Documentation, Pre-trained Transformer Models

ACM Reference Format:
Antonio Mastropaolo, Matteo Ciniselli, Luca Pascarella, Rosalia Tufano, Emad Aghajani, and Gabriele Bavota. 2024. Towards Summarizing Code Snippets Using Pre-Trained Transformers. In *32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3643916.3644400

1 INTRODUCTION

Empirical studies showed that code comprehension can take up to 70% of developers' time [42, 68]. While code comments can support developers in such a process [12], their availability [52] and consistency with the documented code [13, 14, 35] cannot be taken for granted. A helping hand may come from tools proposed in the literature to automatically document code [3, 5, 16, 19, 22, 26, 30, 39, 43, 48, 50, 53, 53, 66, 67, 70]. The most recent techniques (*e.g.,* [19, 22, 30]) train deep learning (DL) models with the aim of learning how to summarize a given piece of code in natural language. This requires the building of a large-scale dataset composed by pairs <*code*, *description*> that can be used to feed the model with *code* instances asking it to generate their *description*. These approaches are usually trained to work at function-level granularity. This means that, in the case of Java, methods are mined from open source projects and linked to the first sentence of their Javadoc which is assumed to represent a plausible code summary.

Having such a granularity could be, however, suboptimal to support comprehension activities. Indeed, while the overall goal of a method might be clear to a developer, they may not understand a specific set of statements in it. Also, looking at the datasets used in the literature to train these models, we found that the methods' descriptions extracted from the Javadoc are usually very short. For example, the seminal dataset by LeClair and McMillan [32], features an average of 7.6 words (median=8.0) to summarize each Java method. While such short descriptions could provide a grasp about the overall goal of the method, it is unlikely that they can actually support a developer struggling to understand it.

For this reason, a few attempts have been made to automatically summarize code snippets rather than entire functions [3, 24, 48, 53, 60, 66, 67]. Most of them are based on information retrieval [3, 48, 66, 67] meaning that, given a code snippet *CS* to document, the most similar snippet to it is identified in a previously built dataset and its comments are reused to summarize *CS*. These approaches, while valuable, rely on manually crafted heuristics to automatically identify the "scope of an inner comment", *i.e.,* the statements that a given comment documents. For example, one may assume that an `//inline comment` in Java documents all following statements until a blank line is found [8]. As we will show, such a heuristic fails in several cases. Other techniques [53, 60] exploit pre-defined templates to document code snippets that, however, cannot generalize to all combinations of code statements one could find.

Given the limitations of previous work, Huang *et al.* [24] proposed an approach exploiting reinforcement learning to document code snippets. The first challenge they faced was the creation of a training dataset. Indeed, while it is relatively easy to collect pairs of <*code*, *description*> when working at function-level granularity, this is not the case for code snippets. For this reason, Huang *et al.* exploited an approach proposed by Chen *et al.* [8] to automatically detect the scope of code comments. The approach exploits a combination of heuristics and learning-based techniques to automatically identify, given a comment, the set of statements documented by it. Using this approach, Huang *et al.* [24] built a dataset of ~124k <*snippet*, *description*> pairs which has been used to train *RL-BlockCom*, a DL model combining reinforcement learning with a classic encoder-decoder model. *RL-BlockCom* is able, given a code snippet as input, to automatically document it reaching a BLEU-4 [44] of 24.28. While being the first DL-based approach to support code snippets' summarization, *RL-BlockCom* suffers of some major limitations mostly related to the way in which its training/test sets have been built exploiting the approach in [8]:

1. Simplified/unrealistic linking of code comment to the documented snippet [8]. This is due to some of the design choices made in the scope detection approach [8]. For example, the authors "*regard the first out-of-scope statement as the demarcation point of the scope of the comment*". This means that, accordingly to their approach, it is not possible for a code comment to document non-contiguous statements. As we will show, our manual validation of 6,645 instances reveals 1598 (~27%) cases of code comments that document non-contiguous statements. These are all cases which cannot be successfully supported by the scope detection approach and, as a consequence, by *RL-BlockCom*.

2. Lack of filters to identify code summaries [8]. Chen *et al.* correctly observed that not all comments "describe" code statements. Thus, they use heuristics to remove commented out code, TODO comments, IDE-generated comments, and non-text comments containing dates or links. Despite these filters, using such an approach to create a training dataset for a snippet summarization approach such as *RL-BlockCom* means feeding it with comments which may not be an actual code summary of the documented snippet. For example, when manually looking at the previously mentioned 6,645 instances, we found 33% of them to just act as a logical split of source code (*i.e.,* a "formatting" comment [46]) without providing additional information on the documented code (*e.g.,* a comment `//get messages` put on top of a method call `getMessages()`). These comments are useless to train a code summarizer, but are not excluded from the *RL-BlockCom* training dataset.

3. The training dataset used in RL-BlockCom includes code summaries as short as two words [24]. These are unlikely to be code summaries useful to support program comprehension.

To address these limitations, in this work we take all steps needed to foster the research on snippets summarization, as depicted in Fig. 1. First (step 1 in Fig. 1), we manually built a dataset of 6,645 <*snippet*, *description*> pairs, in which we classified the code comment (*description*) as being or not a code summary and linked it to the documented Java statements. Such a dataset has been built by ensuring two evaluators for each analyzed comment, with a third one solving conflicts when needed. The overall effort spent by the six involved authors accounts for 815 man-hours.

We use this dataset to fine-tune SALOON (step 2 in Fig. 1), a multi-task pre-trained Text-to-Text-Transfer-Transformer (T5) [47] model able to take as input an inner comment in a method and (i) classify whether it represents a valid code summary with a 83% accuracy; and (ii) link it to the relevant code snippets it documents with a recall/precision higher than 80%. We show that the performance of SALOON are significantly better than the comment-to-code linking approach by Chen *et al.* [8].

Finally (step 3 in Fig. 1), we run SALOON on 10k GitHub Java projects to automatically build a large-scale dataset of ~554k <*snippet*, *description*> pairs. The latter has been used to train and test STUNT, a DL-based approach taking as input a code snippet and automatically generating its code summary. We show that STUNT performs better than IR-based and RL-based baselines *RL-BlockCom*.

Despite this finding, our results also show that STUNT is not yet ready to be deployed to developers and point to more research being needed on the task of snippet summarization.

In summary, our contributions are: (i) the largest manually built dataset in the literature featuring classified and linked code comments; (ii) SALOON, a multi-task DL model able to achieve state-of-the-art performance in the tasks of comment classification and linking; and (iii) STUNT, a code snippet summarization model trained on a large-scale and more realistic dataset as compared to the one used in the literature [24]. The dataset and all code used to train and test the models in this paper are available in our replication package [7].

2 BUILDING A DATASET OF DOCUMENTED CODE SNIPPETS

We detail the process used to build a manually validated dataset featuring triplets <*D*, {*CC*}, *DC*> where *D* represents a natural language comment documenting the code snippet *DC* (*Documented Code*) and {*CC*} represents the *Comment Category* (*e.g.,* code summary, TODO comment), with more than one category possibly being assigned to the comment. We later use such a dataset to train and evaluate the model described in Section 3, taking as input a comment *D* and automatically (i) classifying it, thus being able to check whether *D* is a code summary (*i.e.,* an actual description of the documented code) or another type of comment (*e.g.,* TODOs), and (ii) linking *D* to the corresponding documented code *DC*.

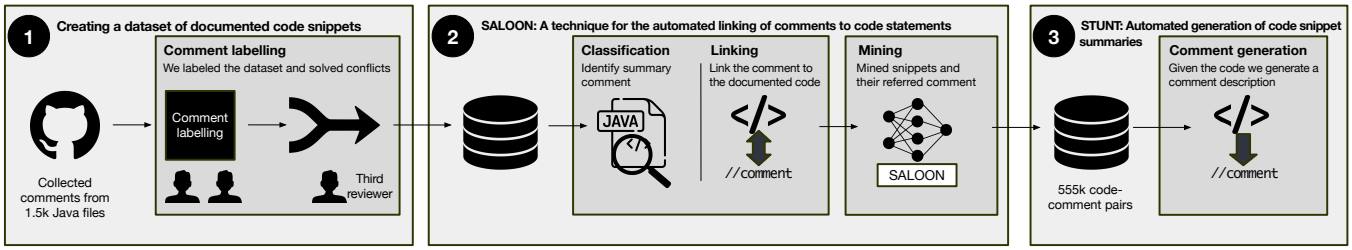

Figure 1: Approach Overview

2.1 Study Design

As a first step to build our dataset we needed to collect the set of code comments D_1, D_2, ..., D_n to manually analyze. To collect these comments, we used the web application by Dabic *et al.* [11] to query GitHub for all Java projects having at least 500 commits, 25 contributors, 10 stars, and not being forks. These filters aim at discarding personal/toy projects and reducing the chance of mining duplicated code. The focus on Java was dictated by the will of accommodating the expertise of the manual validators (*i.e.,* the authors) all having extensive knowledge of the Java programming language. Despite the focus on Java, our methodology to build the dataset as well as to train the models described in the subsequent sections is general and can be reproduced for different languages.

We randomly cloned 100 of the 1,681 projects resulting from our search on GitHub, for a total of ~768k Java files.

We parsed their code to identify comments within each method to manually analyze. We ignored Javadoc comments since they document entire methods rather than code snippets: We only considered single-line (starting with "`//`") and multi-line (starting with "`/*`") comments as subject of our manual analysis. Also, we did not extract comments from test methods (*i.e.,* methods annotated with `@Test`) to increase the cohesiveness of our dataset and only focus on documentation related to production code. The manual analysis has been performed by the six authors (from now on, evaluators) through a web app we developed to support the process.

We targeted the labeling of valid comments (*i.e.,* excluding those removed by the above-described procedure) within 1,500 Java files, with the idea of creating a dataset of ~10k triplets (<D, {CC}, DC>). The web app assigned each Java file to two evaluators who independently labeled the comments in it. If the number of comments in a file was higher than 10, the web app randomly selected a number of comments to label going from 10 to m, where m was the actual number of valid comments in the file. Otherwise all comments in the file were labeled. We opted for this process to avoid an evaluator being stuck too much time on a single file. Also, we did not consider comments belonging to methods longer than 1,024 tokens and made sure no duplicated methods were present in the final dataset (*i.e.,* the same method might be present across different files/projects). The filter on the method length was driven by the final usage we envision for our dataset, namely training DL-models which usually works on inputs of limited size (≤512 tokens, or even less, see *e.g.,* [19, 36, 37, 54–56]). Thus, labeling instances longer than 1,024 tokens would have been a waste of resources.

The goal of the labeling was to firstly assign the comment D to one or more categories CCs. The starting set of categories to use was taken from the work by Pascarella *et al.* [46] and included: *summary*, *rationale*, *deprecation*, *usage*, *exception*, *TODO*, *incomplete*, *commented code*, *formatter*, and *pointer*. We do not describe these categories due to the lack of space, pointing the reader to [46] for a complete description. However, as concrete examples, *summary* represents the classic code description explaining what the code is about, *formatter* is a comment used by developers to better organize the code into logical sections, while *pointer* refers to comments linking external resources. We excluded from the original list by Pascarella *et al.* [46] the following categories (i) *directive* and *autogenerated* since, as described by the authors, they both concern comments automatically generated by the IDE; and (ii) *license* and *ownership*, since this information is usually featured in Javadoc comments.

Finally, we merged the *expand* category into *summary*, since the former is defined by the authors as a code description providing more information than a usual summary. Such a distinction is irrelevant for our work. Besides the set of predefined categories, we also gave the possibility to evaluators to define new categories. If an evaluator defined a new category, it was immediately visible to all other evaluators. The following additional categories have been defined by us: *orphan*, indicating a code comment not linked to any line of code, and *code example*, indicating a comment describing *e.g.,* how to invoke a specific method.

Once the category for a given comment under analysis was defined, the next step was the linking of the comment to the documented code DC. The linking has been performed at line-level granularity. This means, for example, that for a comment D the evaluator could indicate lines 11, 12, and 17 as documented. Note that gaps are possible in DC (*i.e.,* the documented code could be composed by non-contiguous lines). Our replication package [7] shows concrete examples of this scenario, that we omit here due to space limitations. Then, we started resolving conflicts arisen from the manual analysis. Two types of conflicts are possible for each manually defined triplet <D, {CC}, DC>: The two evaluators could have (i) selected a different set {CC} when classifying the comment; and (ii) identified different sets of lines (DC) documented by the comment. Out of the 6,645 manually labeled comments, 1,395 (21%) resulted in a conflict: 1,144 were due to different comment categories selected by the evaluators; 47 to differences in the selected DC; 204 concerned both the categories and the DC. Conflicts were solved by a third evaluator not involved in the labeling of the conflicting instance. Overall, we spent 815 man-hours on the labeling and conflict resolution, manually annotating 6,645 comments (with two evaluators for each of them) coming from 1,508 Java files and

85 software projects. We labeled a bit more than the target 1,500 since multiple evaluators were working in parallel without noticing that we hit our target. The obtained dataset, publicly available in our replication package [7], is briefly described in the following.

2.2 Dataset

Table 1: Dataset output of manual labeling

Category	#Instances	Documented Statements		
		mean	median	sd
Summary	3,841	3.40	3.0	2.70
Formatting	2,209	2.32	2.0	2.65
Rationale	983	3.04	2.0	2.74
TODO	258	0.46	0.0	1.16
Commented Code	184	0.00	0.0	0.00
Pointer	33	2.66	2.0	5.27
Orphan	29	0.00	0.0	0.00
Code Example	9	1.77	2.5	1.48
Deprecation	7	3.14	3.0	1.34
Incomplete	2	1.5	1.5	0.70
Overall	**6,645**	**1.83**	**1.60**	**1.80**

Table 1 summarizes the dataset obtained as output of our analysis. We excluded from the table the categories for which we did not find any instance (*e.g., exception* [46], likely to be more prevalent in Javadoc comments). Since a single comment can be associated to multiple categories (*e.g., summary* and *rationale*), the sum of the "#Instances" column does not add up to the total number of comments we manually classified (*i.e.,* 6,645).

Besides reporting the categories to which the comments in our dataset belong, Table 1 also shows descriptive statistics related to the number of statements documented by comments belonging to different categories. As expected, *orphan* and *commented code* comments are not linked to any code statement. More than 80% of *TODO* comments are also not linked to any statement, since in many cases todos are related to *e.g.,* feature that must be implemented. Similarly, the only two *incomplete* comments we found both of them not linked to any code: These are partially written comments needing rework.

The most frequent category is, as expected, the *summary* one (3,841 instances) grouping comments summarizing one or more code statements (on average, 3.40 statements). Another popular category is "*formatting*", with 2,209 instances.

While one could expect no code linked to formatting comments, this is actually not the case since we used such a category also for comments not adding new information to the documented code but just acting as a logical split of the code (*e.g.,* a comment `//get messages` put on top of a method call `getMessages()`).

Finally, comments explaining the *rationale* for implementation choices account for 983 instances. While we focus on the generation of code summaries, these instances often contains interesting information that are hard to automatically synthesize and could represent a seed for future research.

Interestingly, 1,598 of the comments in our dataset (~27%) include "gaps" in the linked code. This means, for example, that a comment documents lines 11, 12, and 17 (but not lines 13-16) — see [7] for concrete examples. This means that approaches to automatically link comment and code must take such a scenario into account. Motivated by these insights, we fill this gap by creating a novel method for classifying and linking code comments, as elucidated in Section 3.

3 AUTOMATIC CLASSIFICATION OF CODE COMMENTS AND LINKAGE TO DOCUMENTED CODE

We start by presenting SALOON (claSsification And Linking Of cOmmeNts), the approach we devised for the classification of code comments and their linking to the documented code (Section 3.1). Then, we discuss the design of the study we run to assess its accuracy (Section 3.2) and the achieved results (Section 3.3).

Once trained, SALOON can be run on hundreds of projects to build a large-scale dataset featuring classified and linked code comments. While we could just refer to SALOON as a "T5 model trained for comment classification and linking", we preferred to name it to simplify the reading when we introduce the other T5 model we train for the task of code summarization (Section 4).

3.1 Approach Description

SALOON is built on top of T5, a DL transformer-based model [47]. T5 has been presented by Raffel *et al.* [47] as a model that can be trained to support any Natural Language Processing (NLP) task that can be represented in a text-to-text format, meaning that both the input and the output of the model are text strings. Such a representation is well-suited for code-related tasks, as demonstrated by the recent literature (see *e.g.,* [38, 56, 62]).

Raffel *et al.* [47] reported state-of-the-art results for several NLP benchmarks, especially when leveraging the "pretrain-then-finetune" paradigm: The model is first pre-trained on a large dataset with the goal of learning patterns about the underlying language of interest (*e.g.,* Java). Then, it is fine-tuned to learn a specific task of interest (*e.g.,* code summarization). The pre-training is performed using self-supervised pre-training objectives such as the *masked language model*.

The idea is to provide the model with input sentences (*e.g.,* Java methods) in which a percentage of randomly selected tokens has been masked, with the model in charge of guessing them. This prepares the model's weights for the fine-tuning in which tailored datasets are used to teach the model the specific task to support (*e.g.,* pairs of code and comments). The pre-training phase is particularly important when the dataset used for the fine-tuning is expensive to build (*i.e.,* it requires manual validation) and, as a consequence, is limited in size. This is the case for our work, since our fine-tuning is performed on the dataset described in Section 2, in which comments have been categorized and linked to the relevant statements.

In SALOON, we exploit the $T5_{small}$ architecture described by Raffel *et al.* [47]. Due to space constraints, we point the reader to the original paper for all architectural details. We describe how we built the pre-training and fine-tuning datasets for the tasks of comment classification and linking.

3.1.1 Pre-training Dataset. We start from the Java CodeSearchNet dataset [25], which features ~1.6M Java methods, ~499k of which including a Javadoc. Given the tasks we aim at supporting (*i.e.,* automatic classification of code comments and linking to the code they document), there are two "target languages" we aim to expose to T5 during pre-training: Java code and technical natural language in the form of code comments. CodeSearchNet features both of them. We preprocess the dataset by discarding all instances having #tokens > 1,024. During pre-training we treat Java methods and Javadoc comments as separated instances (*i.e.,* we ignore their association), thus removing Java methods and Javadoc comments being longer than 1,024 tokens. Such a filter removed ~32k instances (*i.e.,* 31,702 methods and 178 Javadoc comments). Then, we excluded instances containing non-ASCII characters as well as Javadoc comments composed by less than 5 tokens (words), since unlikely to represent meaningful code descriptions (~57k instances removed). After removing duplicates, we end up with 1,870,888 pre-training instances (1,501,013 Java methods and 369,875 Javadoc).

3.1.2 Fine-tuning Dataset. Two fine-tuning datasets are needed to support the tasks we target (*i.e.,* comment classification and linking). For comment classification, we built a dataset composed by pairs $\langle M_{j,D_i}, C_c \rangle$, in which a specific inner comment D_i within a method M_j is linked to a category C_c classifying it (*e.g.,* code summary). For comment-to-code linking, we built a dataset featuring pairs $\langle M_{j,D_i}, DC \rangle$, in which DC reports the M_j's statements documented by D_i. Both datasets have been extracted from the manually built dataset of 6,645 classified and linked comments (Section 2).

Comment classification. Given the goal of our work (*i.e.,* summarizing code snippets), we are interested in automatically identifying comments we classified as *code summary* while excluding all the others. Starting from the dataset in Table 1, we extracted 3,841 $\langle M_{j,D_i}, C_c \rangle$ having C_c = *code summary* and 2,921 having having C_c = *other*. Basically, we target the training of a binary classifier taking as input a code comment (D_i) in the context of the method it belongs to (M_j) and guessing whether it is a code summary or not.

The specific input we provide to T5 is M_j's code with special tokens `<comment></comment>` surrounding the comment of interest (this is the representation of M_{j,D_i}), and expect as output either "*code summary*" or "*other*" (*i.e.,* C_c).

Differently from the pre-training dataset, we did not need to remove sequences longer than 1,024 tokens, since this has already been done in the first place during the building of the dataset described in Section 2. We randomly split the dataset into 80% training, 10% evaluation, and 10% test. The first row in Table 2 shows the number of instances in these three sets.

Code Linking. Concerning the task of liking comments to code snippets, our training instances are only those comments that we manually labelled as *code summary*. Indeed, we are interested in linking this specific type of comments to their code. Thus, we start from the 3,841 *code summary* instances to build the needed $\langle M_{j,D_i}, DC \rangle$ pairs. Concerning the representation of M_{j,D_i}, it is similar to the previously discussed for the comment classification dataset (*i.e.,* the method M_j with special tags surrounding the inner comment of interest D_i) with the only difference being a special tag `<N>` preceding each statement and reporting its line number in an incremental fashion.

As for the expected output DC (*i.e.,* documented code), it is represented as a stream of "`<N>`" tags representing the line numbers (*i.e.,* statements) within M_j linked to D_i (*e.g.,* `<1><2><4>`). Such a representation allows marking non-contiguous statements documented by D_i. The code linking fine-tuning dataset is composed by 3,841 instances split into 80% training, 10% evaluation, and 10% test as shown in the second row of Table 2. Note that to ensure a fair evaluation of the proposed approach, we split the dataset by taking into consideration the Java class from which these methods were originally extracted.

Table 2: Fine-tuning datasets

Task	Train	Eval	Test
Comment Classification	4,833	726	1,203
Code Linking	2,805	403	633

3.1.3 Training Procedure and Hyperparameters Tuning. We evaluated the performance of eight T5 models (four pre-trained and four non pre-trained) on the evaluation set of each task in terms of correct predictions, namely cases in which the generated output (*i.e.,* the comment category or the documented statements) was identical to the expected output.

We pre-train the T5 model from scratch (*i.e.,* starting from random weights) rather than starting from already pre-trained models for code such as CodeT5 [63], which is based on the same architecture proposed by Raffel *et al.* [47] we exploit in our investigation. Our decision is primarily motivated by the desire to have a model pre-trained on a single programming language (Java) as opposed to a multi-language model (as CodeT5).

We pre-train T5 for 300k steps using a 2x2 TPU topology (8 cores) from Google Colab with a batch size of 16. During pre-training, we randomly mask 15% of tokens in an instance (*i.e.,* Java method or Javadoc comment), asking the model to guess the masked tokens. To avoid over-fitting, we monitored the loss function every 10k steps and stopped the training if such value did not improve after 12 consecutive evaluations (*i.e.,* after 120k steps, one epoch on our pre-training dataset). We use the canonical $T5_{small}$ configuration [47] during pre-training. We also used the pre-training dataset to train a SentencePiece model (*i.e.,* a tokenizer for neural text processing) with vocabulary size set to 32k word pieces.

We fine-tuned a pre-trained and a non pre-trained model experimenting with four different learning rate schedulers (thus leading to eight overall trained models).

Constant Learning Rate (C-LR) fixes the learning rate during the whole training; Inverse Square Root Learning Rate (ISR-LR), in which the learning rate decays as the inverse square root of the training step; Slanted Triangular Learning Rate (ST-LR), in which the learning rate first linearly increases and then linearly decays to the starting value; and Polynomial Decay Learning Rate (PD-LR), having the learning rate decaying polynomially from an initial value to an ending value in the given decay steps. The parameters used for the learning rates are available in [7].

We fine-tuned each of the eight models for a total of 75k steps on the fine-tuning training set of each task. We include in our replication package [7] a table showing the percentage of correct predictions (for the *comment classification* task), precision and recall (for the *code linking* task) achieved by each of the pre-trained and non pre-trained models on the evaluation sets.

Overall, the pre-trained models work substantially better, especially when it comes to the *code linking* task. In particular, in their respective best configuration, pre-trained models achieve (i) a 75% classification accuracy in the *comment classification* task as compared to the 58% of the non pre-trained models; and (ii) 85% precision and 89% recall in the *code linking* task, as compared to the 53% precision and 67% recall of the non pre-trained models. Such a result is expected considering that the fine-tuning training datasets are quite small due to the substantial manual effort required to build them (~6.7k instances for *comment classification* and ~3.8k for *code linking*). Having small fine-tuning datasets is the scenario in which pre-training is known to bring major benefits [49]. As for the learning rate, the best results are achieved with ISR-LR when pre-training and with PD-LR when not pre-training.

To obtain the final model to use in SALOON, we fine-tuned the best performing model (*i.e.,* pre-trained with ISR-LR) using an early-stopping strategy in which we evaluated the model on the evaluation sets every 5k steps, stopping when no improvements were observed for 5 consecutive evaluations. We discuss the results achieved by SALOON as compared to other baselines in Section 3.3.

3.2 Study Design

The goal of the study is to assess the accuracy of SALOON in the two tasks it has been trained for: *comment classification* and *code linking*. The context is represented by the test sets reported in Table 2, featuring 1,203 instances for the task of comment classification and 633 for the task of code linking.

Concerning the comment classification task, we do not compare SALOON against any baseline, since our goal (*i.e.,* identifying only code summaries) is quite specific of our work. Instead, we compare the performance of SALOON against the three following baselines for the task of code linking (the implementation of all baselines is publicly available [7]).

Heuristic-1: blank line [8]. The first baseline is a straightforward heuristic assuming that a given `//inline comment` documents all following statements until a blank line is reached.

Heuristic-2: token-based string similarity [13]. The basic idea of this heuristic is that statements sharing terms with a code comment are more likely to be documented by it. We use the token-based string similarity by Fluri *et al.* [13] to compute the textual similarity between each comment in the test set and all statements in the method it belongs to. A statement is linked to the comment if its similarity with it is higher or equal than a threshold λ. The similarity is computed as the percentage of overlapping terms between the two strings (*i.e.,* comment and statement), with the terms being extracted through space splitting. We experiment with different values for λ, going from 0.1 (*i.e.,* 10% of terms are shared between the two strings) to 0.9 at steps of 0.1.

ML-based solution [8]. The approach by Chen *et al.* [8] relies on the random forest machine learning algorithm to classify statements in a method as linked or not to a given comment. Unfortunately, the source code of such approach is not available and, thus, we had to reimplement it following the description in the corresponding article. In a nutshell, the approach works as follows. The random forest uses three families of features to characterize a given statement and classify it as linked or not to a given comment. The first family comprises eight "code features", capturing characteristics of the statement, such as the statement type (*e.g.,* `if`, `for`) and whether the statement shares method calls with the statements preceding and following it. The second family includes four "comment features", focusing on characteristics of the comment of interest, such as its length and the number of verbs/nouns it contains. Finally, the third family groups four "relationship features", representing the relationship between the comment and the statement (*e.g.,* textual similarity). For a fair comparison, we train the random forest on the same training set used for SALOON.

3.2.1 Data Collection And Analysis. Concerning the *comment classification* task, we run SALOON on the test set and report the accuracy of the model in classifying comments representing "code summaries". As for the *code linking*, we start computing the percentage of **correct predictions**, namely cases in which all statements linked to a comment in the test set match the ones in the oracle. This means that a comment instance correctly linked to two out of the three statements it documents is considered wrong. We also compute the **recall** and **precision** of the techniques at statement-level. The recall is computed as TP/(TP+FN), where TP represents the set of code-to-comment links correctly identified by a technique (*i.e.,* a statement correctly linked to a comment) and FN are the set of correct code-to-comment links in the oracle missed by the approach. The precision is instead computed as TP/(TP+FP), with FP representing the code-to-comment links wrongly reported by the approach (*i.e.,* statements wrongly identified as linked to the comment). We also statistically compare the techniques assuming a significance level of 95%. We compare precision and recall using the Wilcoxon signed-rank test [65]. To control for multiple pairwise comparisons (*e.g.,* SALOON's precision compared with that of the three baselines), we adjust p-values with Holm's correction [20].

We estimate the magnitude of the differences using the Cliff's Delta (d), a non-parametric effect size measure [15]. We follow well-established guidelines to interpret the effect size: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [15]. As for the percentage of correct predictions, we pairwise compare them among the experimented techniques, using the McNemar's test [41], which is a proportion test suitable to pairwise compare dichotomous results of two different treatments. We complement the McNemar's test with the Odds Ratio (OR) effect size. Also in this case we use the Holm's correction procedure [20] to account for multiple comparisons.

3.3 Results Discussion

As for the *comment classification* task, SALOON correctly classifies 78.05% (939/1,203) of instances. Out of the 633 *code summary* comments present in the test set, 536 (84%) have been correctly classified, while 97 have been mistakenly reported as *other*.

Concerning the 570 "*other*" comments, SALOON correctly predicted 403 (70%) of them, wrongly reporting 167 instances as *code summary*. This results in a recall=0.85 and precision=0.76 when identifying a comment as a *code summary*. This means that by running our approach on the comments of a previously unseen software system, we can expect to identify 85% of code summaries present in it accompanied, however, by 25% of false positives (*i.e.,* non *code summary* comments).

Table 3: T5 *vs* baselines on the code linking task

Technique	Correct Predictions	Recall	Precision
Blank line [8]	0.20	0.87	0.57
Token-based similarity [13]			
λ=0.1	0.03	0.62	0.33
λ=0.2	0.05	0.38	0.34
λ=0.3	0.05	0.23	0.26
ML-based [8]	0.23	0.49	0.58
SALOON	**0.58**	**0.89**	**0.86**

Concerning the *code linking* task, Table 3 reports the correct predictions (*i.e.,* for a given comment in our test set **all** linked statements have been correctly identified), recall, and precision achieved by SALOON and the three baselines. Table 4 reports the results of the statistical tests. For the Cliff's Delta *d* we use N, S, M, and L to indicate its magnitude from Negligible to Large.

Note that for the *token-based string similarity* baseline we report the results achieved with different values of λ (*i.e.,* minimum similarity threshold to link a code statement to a comment).

While we also experimented with values going up to 0.9 [7], the recall values were too close to 0 to consider these variants as reasonable baselines.

Table 4: Code linking task: SALOON *vs* baselines

Comparison	Metric	*p*-value	d	OR
Blank line [8] *vs* SALOON	Correct Predictions	<0.05	-	19.28
	Recall	<0.05	-0.04 (N)	-
	Precision	<0.05	-0.48 (L)	-
Token sim.(0.1) [13] *vs* SALOON	Correct Predictions	<0.05	-	70.80
	Recall	<0.05	-0.45 (M)	-
	Precision	<0.05	-0.75 (L)	-
Token sim.(0.2) [13] *vs* SALOON	Correct Predictions	<0.05	-	37.77
	Recall	<0.05	-0.66 (L)	-
	Precision	<0.05	-0.68 (L)	-
Token sim.(0.3) [13] *vs* SALOON	Correct Predictions	<0.05	-	38.00
	Recall	<0.05	-0.80(L)	-
	Precision	<0.05	-0.73 (L)	-
ML-Based [8] *vs* SALOON	Correct Predictions	<0.05	-	15.80
	Recall	<0.05	-0.49 (L)	-
	Precision	<0.05	-0.33 (M)	-

SALOON predicts all statements linked to a given comment in 58% of cases, against the 23% achieved by the best-performing baseline (*ML-based*). The *blank-line technique* achieves 20% of correct predictions.

The results of the statistical tests confirm the better performance ensured by SALOON in terms of correct predictions: McNemar's test always indicates significant differences in terms of correct predictions accompanied by ORs indicating that SALOON has between 15.80 to 70.80 higher odds of providing a correct prediction against the baselines.

Recall and precision values confirm the superiority of SALOON for the *code linking* task. In terms of recall, SALOON is able to correctly link 89% of statements in our dataset, achieving the best performance among all the experimented techniques. While the *blank-line* approach achieves a similar recall (87%) it pays a much higher price in terms of precision, with a 43% false positive rates as compared to the 14% of SALOON. Note that a high recall for this heuristic is expected, considering that it links all statements following a comment until a blank line is found. The *ML-Based* technique can only predict half of the correct links (0.49) while achieving a precision score of 0.58. Accordingly to our results, the *token-based similarity* heuristic does not represent a viable solution for the *code linking* task: The best results are achieved when considering (λ=0.1) as a threshold, for which the technique can ensure a recall of 0.62 and a precision of 0.33. Differences in terms of recall and precision are always statistical significant (see Table 4). The effect size is in most of cases medium or large, with the only exception of the recall test comparing T5 with the *blank-line* baseline, for which a negligible effect size is reported.

To summarize, SALOON is able to identify comments representing code summaries with a recall of 0.85 and a precision of 0.76. Also, it achieves state-of-the-art results in linking comments to the documented code, with a recall of 0.89 and a precision of 0.86. In Section 4 we explain how we exploit this model to build a large-scale dataset aimed at training a T5 fine-tuned for the task of code snippet summarization.

4 SNIPPETS SUMMARIZATION USING T5

We discuss how we trained a T5 model for the task of code snippet summarization (Section 4.1), the study we run to evaluate it (Section 4.2) and the achieved results (Section 4.3). We refer to the snippet summarization approach as "STUNT" (SnippeT sUmmarizatioN using T5).

4.1 Approach Description

We rely on the same T5 architecture described in Section 3.1 and we reuse the same pre-trained model we built for the *comment classification* and *code linking* tasks. Indeed, as explained in Section 3.1.1, we pre-trained the model on a dataset composed by ~1.5M Java methods and their inner comments and ~370k Javadoc comments. Thus, T5 has been pre-trained to acquire knowledge about the two "target languages" relevant for the summarization task as well (*i.e.,* Java code and technical language used to summarize it). We detail the fine-tuning dataset and the training procedure.

4.1.1 Fine-tuning Dataset. We used the GHS tool by Dabic *et al.* [11] to query GitHub for all public non-forked Java projects with minimum 50 commits, 5 contributors, and 10 stars. The idea of these filters was to remove toy/personal projects while still obtaining a large set of projects to provide as input to SALOON with the goal of identifying comments representing summaries and linking them to the relevant code. We cloned 10k of the 18.7k projects returned by our query and extracted their methods using srcML [10].

We excluded all methods longer than 512 tokens and removed all duplicates, obtaining a set of methods *S*. We also removed duplicates between our pre-training dataset and *S* and between our manually labeled dataset (Section 2.2) and *S*.

Concerning the removal of duplicates between the pre-training dataset and S, this was needed since S is our starting point to build the fine-tuning dataset for the snippet summarization task from which we will also extract the test set on which STUNT will be evaluated. Thus, we ensure that STUNT is not evaluated on already seen instances. As for the removal of duplicates between the manually labeled dataset and S, this is due to the fact that SALOON (*i.e.,* our approach for comment classification and linking) has been trained on those instances and we will run it on S to build the fine-tuning dataset for STUNT (*i.e.,* for code summarization). Running SALOON on already seen instances would inflate its performance, and not provide a realistic picture of what can be achieved by training STUNT on a dataset automatically built using SALOON.

From the remaining methods, we extracted all inner comments, filtering out those shorter than 5 words (unlikely to represent a meaningful code summary). As done in previous code summarization works [30], we lowercased and stemmed the comments (using the spaCy NLP library [2]). Then, for each comment D_i extracted from a method M_j we created an instance M_{j,D_i} in which M_j's code features special tokens `<comment></comment>` to surround the comment of interest (D_i). This means that if M_j features three inner comments, three M_{j,D_i} instances will be created, each having a different comment (D_i) "tagged". This format is the one expected by SALOON to automatically (i) classify D_i as *code summary* or *other*, and (ii) link D_i to the relevant code statements.

The above-described process resulted in 2,210,602 M_{j,D_i} instances that we provided as input to SALOON, which classified 907,660 of them as *code summary*. Among these, SALOON automatically linked code statements to the *code summaries* in ~85% of cases (776,531). These instances are $\langle M_{j,DC}, D_i \rangle$ pairs, where $M_{j,DC}$ represents the method M_j with special tokens `<start><end>` surrounding the statements (DC) documented by D_i.

If more non-contiguous statements are documented, multiple `<start><end>` pairs are injected in M_j. These pairs are those needed to fine-tune STUNT for the task of snippet summarization: the input provided to the model is $M_{j,DC}$ (*i.e.,* a snippet to document) and the expected output is the documentation D_i. To avoid favoring the model during testing, we also removed all duplicates at snippet-level granularity. This means that if we have in our dataset two different methods containing the same DC (*i.e.,* the same code snippet to document), we only keep one of them. Also, being SALOON an automated approach, it is expected to produce wrong instances (*e.g.,* comments linked to wrong statements) which, in turn, will penalize the performance of STUNT. By manually inspecting a sample of the pairs in our dataset, we noticed that one clear case of wrong instances are those in which the model had very low confidence in identifying the documented statements thus producing random symbols rather than the expected documented line numbers. We automatically remove those instances, obtaining a set of 554,748 pairs, split into 80% training (443,798), 10% evaluation (55,475), and 10% testing (55,475).

4.1.2 Training Procedure and Hyperparameters Tuning. As explained, we started from the already pre-trained T5 model. We then followed the same hyperparameters tuning discussed in Section 3.1.3, assessing the performance of four different learning scheduler on the evaluation set using the BLEU-4 score [44] as performance metric. The BLEU-4 variant computes the BLEU score by considering the overlap of 4-grams between the generated text (*i.e.,* the synthesized snippet summary) and the target text (*i.e.,* the summary written by the original developers). This metric has been used by most of the previous work on code summarization (see *e.g.,* [4, 19, 21–24, 27–29, 31, 57, 59, 61, 64, 69, 71]). Each of the four models has been trained for 100k steps before its evaluation. C-LR (*i.e.,* constant learning rate) provided the best performance. Data about this evaluation are available in our replication package [7].

Once identified the best T5 variant, we fine-tuned it for up to 500k steps, using an early-stopping strategy to tame over-fitting. To this aim, we monitored the BLEU-4 score achieved on the evaluation set every 5k steps, stopping the training when no improvements were observed after 5 consecutive evaluations.

4.2 Study Design

The goal is to assess the accuracy of STUNT for snippet summarization. The context is represented by (i) 55,475 $\langle M_{j,DC}, D_i \rangle$ pairs identified by SALOON as described in Section 4.1.1 and belonging to the test set, and (ii) the test set made publicly available by Huang *et al.* [24] when presenting *RL-BlockCom*, the state-of-the-art snippet summarization approach discussed in Section 1.

We assess the performance of STUNT against an information retrieval (IR)-based technique (*i.e.,* IR-Jaccard) and *RL-BlockCom*. To explain the basic idea behind the IR-based baseline let us remind that both our training and test set are composed by $\langle M_{j,DC}, D_i \rangle$ pairs. Given a pair in the test set, the baseline retrieves in the training set the pair having the DC snippet being the most similar to the one in the test set pair. This means that this pair contains a documented snippet that is very similar to the one in the test set for which we have to generate a code summary. Once identified the most similar snippet in the training set, the IR-based technique reuses its description to document the instance in the test set. This baseline serves as a representative of works using IR to retrieve similar comments from a given dataset, including *e.g.,* [67].

IR: Jaccard index [17]. IR-Jaccard identifies the most similar snippet using the Jaccard similarity index. The latter considers the overlapping between two sets of unique elements, representing in our case the tokens composing the documented code (DC) in the test instance and in each of the training instances. Indeed, we need to compare each instance in the test set to all those in the training set to find the most similar one. The similarity is computed as the percentage of overlapping tokens between the two sets.

An additional baseline for STUNT is *RL-BlockCom* by Huang *et al.* [24]. Despite the code being available, we did not manage to re-train their approach on our dataset. We contacted the authors asking for help without, however, receiving answer. Thus, as an alternative form of comparison, we thought about training and testing STUNT on their dataset, which is publicly available, and then comparing the summaries generated by STUNT with those generated by *RL-BlockCom*. Unfortunately, the authors did not make the summaries generated by their approach publicly available. The only viable form of comparison we found was to (i) re-train STUNT on the training dataset made available by Huang *et al.* [24] and used to train *RL-BlockCom*; (ii) use this trained version of STUNT to generate predictions on the same test set on which *RL-BlockCom*

has been evaluated; (iii) use the evaluation scripts made available by Huang *et al.* for the computation of the sentence-level BLEU score; and (iv) compare the achieved results with those reported in their paper. Indeed, not having access to the summaries generated by *RL-BlockCom* does not allow us to double-check the data reported in the original paper nor to compute additional metrics besides those used by the authors (BLEU). Note also that the training/test datasets shared by Huang *et al.* feature pairs $\langle DC, D_i \rangle$ as compared to our $\langle M_{j,DC}, D_i \rangle$ pairs. This means that STUNT cannot exploit the contextual information of the method M_j when generating the predictions on their dataset.

4.2.1 Data Collection And Analysis. To compare the performance of our model against the two IR-based baselines, we exploit three metrics explained in the following.

Out of those, only BLEU has been used in the comparison with *RL-BlockCom* for the reasons previously explained.

BLEU [44] assesses the quality of the automatically generated summaries by assigning a score between 0 and 1. In our case, 1 indicates that the natural language summary automatically generated is identical to the one originally written by the developer. Since in the test set we built there are no summaries shorter than 4 words, we use the BLEU-4 variant in the comparison with the IR-based baselines. When comparing with *RL-BlockCom* on their test set, we also compute BLEU-1, BLEU-2 and BLEU-3 as done by Huang *et al.* [24].

METEOR [6] is a metric based on the harmonic mean of unigram precision and recall (the recall is weighted higher than the precision). Compared to BLEU, METEOR uses stemming and synonyms matching to better match the human perception of sentences with similar meanings. Values range from 0 to 1, with 1 being a perfect match.

ROUGE [34] is a set of metrics focusing on automatic summarization tasks. We use the ROUGE-LCS (Longest Common Subsequence) variant, which identifies longest co-occurring in sequence n-grams. ROUGE-LCS returns three values, the recall computed as *LCS(X,Y)/length(X)*, the precision computed as *LCS(X,Y)/length(Y)*, and the F-measure computed as the harmonic mean of recall and precision where X and Y represent two sequences of tokens.

We also statistically compare the different approaches assuming a significance level of 95%. Also in this case we use the Wilcoxon signed-rank test [65], adjusting p-values to account for multiple comparisons (Holm's correction procedure [20]) and the Cliff's Delta (d) as effect size measure [15]. The statistical comparison was not possible with *RL-BlockCom* since we only had access to the overall BLEU scores reported in the paper (*i.e.,* the BLEU scores for each generated summary were not available).

4.3 Results

Table 5 compares STUNT and *RL-BlockCom*, using the values reported in the paper by Huang *et al.* [24] as BLEU scores for *RL-BlockCom*. STUNT achieves better performance for all BLEU scores, outperforming the state-of-the-art approach by a large margin (*e.g.,* +7 points of BLEU-4). A deeper comparison of the two techniques is not possible since the summaries generated by *RL-BlockCom* are not available.

Table 5: BLEU scores: STUNT *vs* RL-BlockCom [24]

	RL-Com	STUNT
BLEU-1	32.18	**34.17**
BLEU-2	25.98	**31.09**
BLEU-3	24.36	**30.63**
BLEU-4	24.28	**31.22**

Table 6 compares STUNT against IR-Jaccard on the large-scale dataset we built. Accordingly to all metrics used in our evaluation, the gap in performance between STUNT and the baseline (*i.e.,* IR-Jaccard) is substantial, with at least a +11 in terms of BLEU-4, a +12 in terms of ROUGE-LCS f-measure, and a +16 in terms of METEOR score. As observed by Roy *et al.* [51], METEOR is "*extremely reliable for differences greater than 2 points*" in assessing code summarization quality as perceived by humans (*i.e.,* also humans are likely to prefer STUNT's summaries over those generated by the baselines).

Table 6: Evaluation Metrics: STUNT *vs* IR-Jaccard

	IR-Jaccard	STUNT
BLEU-4 [44]	27.43	**38.42**
ROUGE-LCS [34]		
$precision$	23.00	**34.21**
$recall$	23.04	**37.39**
$fmeasure$	22.33	**34.57**
METEOR [6]	25.04	**41.75**

The statistical analyses presented in Table 7 validate STUNT's superior performance compared to IR-Jaccard. Notably, we observe significant p-values and medium effect sizes for BLEU-4 and ROUGE-LCS (f-measure), while METEOR demonstrates a large effect size.

Table 7: Statistical Tests: STUNT *vs* IR-Jaccard

Comparison	Metric	p-value	d
IR (Jaccard) *vs* STUNT	BLEU-4	<0.001	-0.451 (M)
	ROUGE-LCS (f-measure)	<0.001	-0.471 (M)
	METEOR	<0.001	-0.474 (L)

While the metrics we computed provide a fair comparison among the experimented techniques, they do not give a clear idea of the quality of the summaries generated by STUNT. To this aim two of the authors manually inspected 384 randomly selected summaries generated by STUNT for which the generated text was different from the target summary (*i.e.,* the one written by developers). These are cases that in a "binary quantitative evaluation" would be classified as wrong predictions. The authors independently classified each summary as *meaningful* or *not meaningful*, based on the ability of the summary to properly describe the documented snippet. In the labeling, the two involved authors achieved a Cohen's kappa [9] of 0.61, indicating a substantial agreement when measuring inter-rater reliability for categorical items.

Conflicts, arisen in 71 cases and have been solved through open discussion among the authors. We classified 224 summaries as meaningful, with some of them representing even a better summary than the one manually written by the original developers. For example, we found the comment `if we have a frontend then we need to get the action list` to be more meaningful and detailed than the `exit if we do not have a frontend` written the developer. However, we also want to highlight the ~41% (160) of automatically generated summaries which were not meaningful and that stress how far we still are from obtaining a code summarizer being accurate enough to be deployed to developers (*i.e.,* generating correct summaries in most of cases).

5 THREATS TO VALIDITY

We discuss the threats that could affect the validity of our findings.

Internal Validity. Building our dataset of classified and linked code comments (Section 2) involved a certain degree of subjectivity. To partially address this threat, two evaluators independently assessed each instance and a third one solved conflicts when needed. Still, imprecisions are possible.

We performed a limited hyperparameters tuning of the T5 models, only experimenting with different learning rates. For example, we did not change the number of layers, but relied on the default $T5_{small}$ architecture by Raffel *et al.* [47]. Better results could be achieved with additional tuning. Also, relying on pre-trained code models like CodeT5 [63], might produce better results.

Construct Validity. When experimenting with SALOON, we compared its performance with the technique by Chen *et al.* [8]. However, since their approach is not publicly available, we had to reimplement it following the paper's description.

We release our implementation [7]. Still related to the used baselines, as explained in Section 4.2 we did not manage to compare STUNT (our approach for snippet summarization) with RL-BlockCom [24] on our dataset. At least, we presented a comparison performed on the dataset released by the authors.

External Validity. The manually built dataset represents the obvious bottleneck in terms of generalizability, since it is based on the analysis of "only" 1,508 Java files and also capped our training/evaluation of SALOON. Still, building such a dataset costed over 815 man-hours. Also, we did not compare our technique against general purpose large language models such as ChatGPT [1], since designing a fair evaluation is challenging due to the unknown training set behind these LLMs. For example, we could have tested the ability of ChatGPT to summarize specific snippets which, however, were part of its training set together with their related comment.

6 RELATED WORK

We discuss techniques for (i) the automated linking of code to comments, and (ii) code summarization.

6.1 Linking Documentation to Code

Some works, while studying code comments from different perspectives, came up with possible heuristics to identify the scope of code comments. Haouari *et al.* [18] showed that code comments frequently document the following code.

While this is confirmed in our dataset, we also observed that it is far from trivial to assess the exact set of (following) lines actually documented by the comment due to the lack of a clear separator isolating the documented from the undocumented code.

Fluri *et al.* [13], while studying the co-evolution of code and comments, suggested that token-based similarity between the code and the comment can be used to identify documented statements. Such an intuition has also been echoed by McBurney and McMillan [40]. As shown in our study, our DL-based approach substantially outperforms similarity-based heuristics.

Finally, Chen *et al.* [8] recently proposed a machine-learning based method for the automatic identification of code comments scope. Such an approach has been extensively described in Section 3.2 as one of the baselines we compared with. Our approach outperforms this approach as well.

6.2 Code Summarization

Several techniques have been proposed to automatically summarize source code [73]. We focus our discussion on (i) techniques aimed at documenting code snippets (regardless of the underlying techniques used) and (ii) DL-based approaches (regardless of the target code granularity).

Most of the techniques targeting the documentation of code snippets are based on IR. Representative works in this area are *CodeInsight* [48], *ColCom* [66, 67], and ADANA [3]. The IR baselines we exploit is Section 4.2 are representative of these works.

Another family of techniques related to code snippets documentation relies on manually defined templates to describe high-level actions performed within functions. Seminal work in this area are from Sridhara *et al.* [53] and Wang *et al.* [60]. These approaches, while valuable, cannot generalize to all combinations of code statements one could expect to find since they are based on predefined templates. For this reason, data-driven techniques exploiting DL have been proposed [24, 72]. When it comes to snippet-level granularity, *RL-BlockCom* [24] represents the state-of-the-art. As shown, our approach performs substantially better than *RL-BlockCom*.

Most of the other DL-based techniques proposed in the literature focused on documenting entire functions. Liang *et al.* [33] presented Code-RNN, a Recursive Neural Network exploiting a GRU cell (Code-GRU) specifically designed for code comments generation. The authors show that their approach can achieve higher ROUGE score [34] as compared to vanilla DL models not tailored for source code. Hu *et al.* [21] built a dataset of ⟨method, javadoc⟩ pairs from ~9k Java projects to train a Deep Neural Network (DNN) aimed at documenting Java methods. The authors used the BLEU-4 score [45] to compare the summaries generated by their approach to those of the neural attention model by Iyer *et al.* [26], showing the superiority of the proposed technique.

While previous works represented code as a stream of tokens, other authors combined such a representation with one capturing AST information [30, 58]. For example, LeClair *et al.* [30] showed how exploiting AST-based information allows to improve the performance achieved by both Hu *et al.* [21] and Iyer *et al.* [26]. The work by LeClair *et al.* has been later on extended and improved by Haque *et al.* [19], which provide as input to the model additional

information related to the "file context" of the method to summarize. They show that such a contextual information helps to further boost performance.

Zhang [70] showed that combining IR and DL techniques it is possible to boost the performance of function-level code summarization. Our work focuses on the related but different problem of snippet summarization that, as explained, poses different challenges especially in the building of the training data.

7 CONCLUSIONS

We targeted the problem of code snippet summarization, presenting (i) a manually labeled dataset of ~6.6k code comments classified in terms of information they provide (*e.g.*, code summary) and linked to the code statements they document; (ii) SALOON, a T5 model trained on our manually built dataset to automatically classify and link inner comments in Java code; and (iii) STUNT, a T5 model trained on a large-scale dataset of documented code snippets automatically created by running SALOON on 10k Java projects.

We achieved promising results for both code linking and snippet summarization, pointing however to the need for research in this field. Our dataset and our models, publicly released [7], represent a step in that direction.

ACKNOWLEDGMENT

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851720).

REFERENCES

[1] [n.d.]. ChatGPT https://openai.com/blog/chatgpt.
[2] [n.d.]. Spacy. https://spacy.io.
[3] E. Aghajani, G. Bavota, M. Linares-Vásquez, and M. Lanza. 2021. Automated Documentation of Android Apps. *IEEE Transactions on Software Engineering* 47, 1 (2021), 204–220.
[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).
[5] 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)*.
[6] 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
[7] Double Blind. [n.d.]. https://snippets-summarization.github.io.
[8] 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* 153 (2019), 45–63.
[9] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.
[10] Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 516–519.
[11] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 560–564.
[12] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. 2005. A Study of the Documentation Essential to Software Maintenance. In *International Conference on Design of Communication*. 68–75.
[13] 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 2007)*. 70–79.
[14] Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C. Gall. 2009. Analyzing the Co-evolution of Comments and Source Code. *Software Quality Journal* 17, 4 (2009), 367–394.
[15] Robert J Grissom and John J Kim. 2005. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers.
[16] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *2010 17th Working Conference on Reverse Engineering*. 35–44.
[17] John M Hancock. 2004. Jaccard distance (Jaccard index, Jaccard similarity coefficient). *Dictionary of Bioinformatics and Computational Biology* (2004).
[18] Dorsaf Haouari, Houari Sahraoui, and Philippe Langlais. 2011. How good is your comment? a study of comments in java programs. In *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 137–146.
[19] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved Automatic Summarization of Subroutines via Attention to File Context *(MSR '20)*. 300–310.
[20] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.
[21] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation *(ICPC '18)*. Association for Computing Machinery, 200–210.
[22] 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* 25 (2020), 2179–2217.
[23] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. (2018).
[24] 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. *Information and Software Technology* 127 (2020), 106373.
[25] 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).
[26] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.
[27] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.
[28] Alexander LeClair, Aakash Bansal, and Collin McMillan. 2021. Ensemble Models for Neural Source Code Summarization of Subroutines. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 286–297. https://doi.org/10.1109/ICSME52107.2021.00032
[29] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*. 184–195.
[30] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines *(ICSE '19)*. 795–806.
[31] 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)*. 795–806. https://doi.org/10.1109/ICSE.2019.00087
[32] Alexander LeClair and Collin McMillan. 2019. Recommendations for datasets for source code summarization. *arXiv preprint arXiv:1904.02660* (2019).
[33] Yuding Liang and Kenny Q. Zhu. 2018. Automatic Generation of Text Descriptive Comments for Code Blocks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence (AAAI'18/IAAI'18/EAAI'18)*. AAAI Press, Article 641, 8 pages.
[34] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.
[35] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk. 2015. How do Developers Document Database Usages in Source Code?. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 36–41. https://doi.org/10.1109/ASE.2015.67
[36] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. *Neural-Machine-Translation-Based Commit Message Generation: How Far Are We?* Association for Computing Machinery, New York, NY, USA, 373–384. https://doi.org/10.1145/3238147.3238190
[37] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. 2022. Using Deep Learning to Generate Complete Log Statements. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2279–2290. https://doi.org/10.1145/3510003.3511561
[38] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE,

336–347.
[39] P. W. McBurney and C. McMillan. 2016. Automatic Source Code Summarization of Context for Java Methods. *IEEE Transactions on Software Engineering* 42, 2 (2016), 103–119.
[40] Paul W McBurney and Collin McMillan. 2016. An empirical study of the textual similarity between source code and source code summaries. *Empirical Software Engineering* 21, 1 (2016), 17–42.
[41] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.
[42] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I know what you did last summer: an investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, Andrea De Lucia, Christian Bird, and Rocco Oliveto (Eds.). IEEE Computer Society, 25–35.
[43] 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 *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.
[44] 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.
[45] 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 on Association for Computational Linguistics (ACL '02)*. 311–318.
[46] Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in Java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 227–237.
[47] 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. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html
[48] M. M. Rahman, C. K. Roy, and I. Keivanloo. 2015. Recommending insightful comments for source code using crowdsourced knowledge. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 81–90.
[49] Romain Robbes and Andrea Janes. 2019. Leveraging Small Software Engineering Data Sets with Pre-Trained Neural Networks. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 29–32.
[50] Paige Rodeghero, Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Detecting User Story Information in Developer-Client Conversations to Generate Extractive Summaries *(ICSE 2017)*. 49–59.
[51] 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1105–1116. https://doi.org/10.1145/3468264.3468588
[52] D. Spinellis. 2010. Code Documentation. *IEEE Software* 27, 4 (July 2010), 18–19. https://doi.org/10.1109/MS.2010.95
[53] 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.
[54] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 301–312.
[55] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4 (2019), 19:1–19:29.
[56] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using Pre-Trained Models to Boost Code Review Automation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2291–2302. https://doi.org/10.1145/3510003.3510621
[57] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 397–407.
[58] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. *Improving Automatic Source Code Summarization via Deep Reinforcement Learning*. 397?407.
[59] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip S. Yu, and Guandong Xu. 2022. Reinforcement-Learning-Guided Source Code Summarization Using Hierarchical Attention. *IEEE Transactions on Software Engineering* 48, 1 (2022), 102–119. https://doi.org/10.1109/TSE.2020.2979701
[60] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. 2017. Automatically generating natural language descriptions for object-related statement sequences. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 205–216.
[61] Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Shi Han, Hongyu Zhang, and Dongmei Zhang. 2021. CoCoSum: Contextual Code Summarization with Multi-Relational Graph Neural Network. *arXiv preprint arXiv:2107.01933* (2021).
[62] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685
[63] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
[64] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 349–360.
[65] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.
[66] Edmund Wong, Taiyue Liu, and Lin Tan. [n.d.]. CloCom: Mining existing source code for automatic comment generation. In *Software Analysis, Evolution and Reengineering (SANER), 2015*. 380–389.
[67] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 562–567.
[68] 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* (2018), 951–976.
[69] Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. 2020. Leveraging code generation to improve code retrieval and summarization via dual learning. In *Proceedings of The Web Conference 2020*. 2309–2319.
[70] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1385–1397.
[71] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1385–1397.
[72] Wenhao Zheng, Hong-Yu Zhou, Ming Li, and Jianxin Wu. 2019. CodeAttention: translating source code to comments by exploiting the code constructs. *Frontiers Comput. Sci.* 13, 3 (2019), 565–578.
[73] Yuxiang Zhu and Minxue Pan. 2019. Automatic code summarization: A systematic literature review. *arXiv preprint arXiv:1909.04352* (2019).