Analyzing the Performance of Large Language Models on Code Summarization

Rajarshi Haldar, Julia Hockenmaier
Department of Computer Science, University of Illinois Urbana-Champaign
{rhaldar2, juliahmr}@illinois.edu

Abstract

Large language models (LLMs) such as Llama 2 perform very well on tasks that involve both natural language and source code, particularly code summarization and code generation. We show that for the task of code summarization, the performance of these models on individual examples often depends on the amount of (subword) token overlap between the code and the corresponding reference natural language descriptions in the dataset. This token overlap arises because the reference descriptions in standard datasets (corresponding to docstrings in large code bases) are often highly similar to the names of the functions they describe. We also show that this token overlap occurs largely in the function names of the code and compare the relative performance of these models after removing function names versus removing code structure. We also show that using multiple evaluation metrics like BLEU and BERTScore gives us very little additional insight since these metrics are highly correlated with each other.

Keywords: Natural Language Generation, Neural Language Representation Models, Summarisation

1. Introduction

There is a growing interest in applying NLP techniques to tasks related to automated program understanding, generation, and retrieval, all of which promise to improve access to code. Popular tasks include Code Summarization (translating code into natural language, e.g. Miceli Barone and Sennrich, 2017), Code Generation, Code Completion, Code Translation and Natural Language Code Search (retrieving a code snippet given a natural language query, e.g. Gu et al., 2018a). There is a practical need for such systems since the ability to automatically generate code snippets or doc strings or search large code bases can significantly increase the productivity of software developers. However, there is also a growing interest in developing models and datasets for these tasks within the NLP community, often driven by an assumption that code can be seen as a semantic interpretation of its natural language description.

But while current models show impressive performance, it is still important to analyze how much understanding these models have of the structure or semantics of the code. To make their code more human-readable, software developers often employ English words in the names of functions, variables, or data structures. And, although they did not evaluate the most recent web-scale large language models, the authors of MCoNaLa (Wang et al., 2023) showed that the performance of code generation models drops significantly compared to English if the input is in Spanish, Japanese, or Russian.

We, therefore, ask to what extent the large language models (LLMs) that are used for tasks like code generation or summarization actually understand the semantic relation between natural language and code, and to what extent they simply rely on this superficial token similarity. In this paper, we attempt to address this question by analyzing the performance of large language models (LLMs) on code summarization. Our analysis aims to shed light on the following, more specific questions: (1) to what extent do the summaries generated by these models simply consist of tokens that are directly copied from the code? (2) How much does model performance depend on the presence of function names that give away the semantics of the code? (3) To what extent do these models rely on the syntactic structure and underlying logic of the code?

To answer the first research question, we split the examples from the dataset into different buckets depending on the token overlap between the code and the description and see how much the performance varies across those buckets. For the second and third research questions, we make several transformations to the code before feeding it to an LLM. This includes changing or obfuscating certain function names, and removing the control structures in the body of the code. We then examine their impact on code summarization performance. We observe the effect of the code in standard datasets like CodeXGLUE having informative function and identifier names with a high token overlap with their descriptions, and analyze how this affects model behavior. This token overlap between the function names and the target summary makes the task easier. Our experiments[1]

[1] The code for this paper will be released at https://github.com/rajarshihaldar/analyze-llm-code-summarization

LREC-COLING 2024, pages 995–1008
20-25 May, 2024.

show that the performance of several state-of-the-art LLMs is often due to the high string similarity of the natural language descriptions to the code they are paired with.

2. Background

2.1. Large Language Models For Code

Applying natural language processing techniques to source code has produced outstanding results. Code2vec (Alon et al., 2019) showed that techniques that are used to induce semantic embeddings or vectors for natural language input also work well to represent input code snippets and predict their semantic properties. DeepCS (Gu et al., 2018b) showed that by mapping both code and natural language prompts to embeddings you could perform retrieval on code. Another breakthrough was the introduction of sequence-to-sequence (seq2seq) models for generating comments for a given input code (Hu et al., 2018a). These initial models treat code as a sequence of tokens and were quickly followed by models that account for the structure of the code through Abstract Syntax Trees (Zhang et al., 2019a; Wan et al., 2019; Haldar et al., 2020), Graph Neural Networks (Sieper et al., 2020; Ling et al., 2021; Liu et al., 2021), and Graph Attention Neural Networks (Wang et al., 2022a).

Inspired by the success of transformer-based (Vaswani et al., 2017), pre-trained large language models (LLMs) like BERT (Devlin et al., 2019), XLNet (Yang et al., 2019), GPT (Brown et al., 2020), RoBERTa (Liu et al., 2019), SynCoBERT (Wang et al., 2021a), and T5 (Raffel et al., 2020) on core NLP tasks, and facilitated by the availability of large datasets that pair natural language with code, e.g. CodeSearchNet (Husain et al., 2019), LLMs are now commonly used for tasks that involve source code and natural language. CodeBERT (Feng et al., 2020), and its successor GraphCodeBERT (Guo et al., 2020), were two of the initial wave of LLMs trained on paired natural (NL) and programming language (PL) sequences. Soon other LLMs like PLBART (Ahmad et al., 2021) and CodeT5 (Wang et al., 2021b) made significant gains in code summarization. At the same time, models like CodeGen (Nijkamp et al., 2023) and Codex (Chen et al., 2021) made gains in code generation.

The current state of the art includes even larger instruction-tuned models that can perform a wide variety of tasks including ones related to code-NL, e.g. GPT-4 (OpenAI, 2023), Llama 2 (Touvron et al., 2023) and PaLM 2 (Anil et al., 2023). Instead of being fine-tuned for a specific task, these models are trained to respond to a prompt describing a task followed by an input. This leads to improved performance on a variety of code-NL tasks. While GPT-4 has the best performance[2], some researchers have found ways to improve its output further. For example, leveraging verbal reinforcement on an LLM improves its code generation capabilities (Shinn et al., 2023).

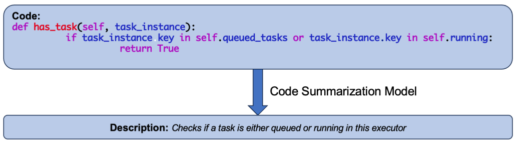
Although current LLMs differ in important details (size, amount of training data, pre-training regime, fine-tuning, etc.), they use a form of **subword tokenization** (Kudo and Richardson, 2018; Sennrich et al., 2016), in which words are split into shorter strings (e.g. extracts → ext ract s) and underscores are treated as separators (e.g. from_url → from _ url), such that the model's vocabulary consists of subwords rather than whitespace-delineated tokens. Figure 2 shows how even just a function definition reveals valuable information about the tokens that are likely to occur in the reference description when it is tokenized by a subword tokenizer (here, Llama 2).

2.2. Code Summarization

Code Summarization systems translate code snippets into automatically generated English summaries that describe what the code does. Given an input code snippet C, the system has to return a description D that accurately describes what that code does. Figure 1 shows an example of code summarization being performed.

When the task was first introduced, template-based approaches (Sridhara et al., 2010) requiring expert domain knowledge were used, followed by information retrieval-based approaches (Eddy et al., 2013; Rodeghero et al., 2014). In IR-based approaches, the model would extract the most relevant tokens from the code to generate a term-based summary. This did not require any domain knowledge but the models used did not show any deep understanding of the code structure and instead relied on informative function names and comments to generate summaries. This was followed by early neural approaches (Iyer et al., 2016), Encoder-decoder frameworks like (Hu et al., 2018a) and transformer-based approaches, like PLBART (Ahmad et al., 2021), Structure-induced Transformers (Wu et al., 2021), CoTexT (Phan et al., 2021) CodeGPT (Lu

[2] Although it is unclear what specific data GPT-4 has been trained on, their training data is likely to include the code snippets contained in the datasets used in this paper and data contamination is known to be a significant factor in the performance of these models on coding tasks (Narayanan and Kapoor, 2023). Moreover, GPT-4 is frequently updated behind the scenes based on the data customers provide. This makes any analyses on this data non-reproducible. Due to these concerns, we have not included GPT-4 in our studies.



Code:

```
def has_task(self, task_instance):
        if task_instance.key in self.queued_tasks or task_instance.key in self.running:
                return True
```

Description: *Checks if a task is either queued or running in this executor*

Figure 1: In code summarization, a code snippet (here, from CodeXGLUE) is given as input to a model that returns an English description.

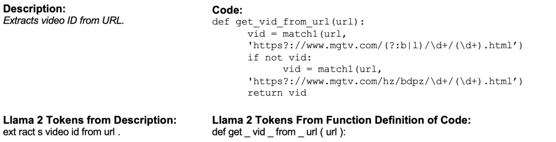
et al., 2021), Codex (Chen et al., 2021), and CodeT5 (Wang et al., 2021b). Recently, these models have been surpassed by large language models using decoder-only architectures like PaLM 2 (Anil et al., 2023), GPT-4 (OpenAI, 2023), and Llama 2 (Touvron et al., 2023).

Benchmark datasets for this task include TL-CodeSum (Hu et al., 2018b), Funcom (LeClair et al., 2019), CodeSearchNet (Husain et al., 2019) and more recently, CodeXGIUE (Lu et al., 2021).

Dataset: **CodeXGLUE** (Lu et al., 2021) is a standard benchmark for several code-NL tasks including code summarization. It is a filtered subset of **CodeSearchNet** (Husain et al., 2019), consisting of code snippets paired with English descriptions that are scraped from public open-source GitHub repositories for Go, Java, JavaScript, PHP, Python, and Ruby. The description paired with each code snippet is the first paragraph of the repository. Examples were filtered out from this dataset if their code could not be parsed into an abstract syntax tree, or if their descriptions were not in English, contained special strings like "http://", were empty, or not between 3 and 256 tokens in length. Figure 1 shows an example from the dataset being used for code summarization. In our experiments, we use the Python examples from CodeXGLUE, which contain 251,820 training, 13,914 dev, and 14,918 testing data points.

Metric: BLEU (Papineni et al., 2002) is computed by matching *n*-grams between generated summaries and human written summaries. A perfect match between the two summaries would give a score of 100%, and a perfect mismatch would give a score of zero. BLEU-4 is the standard metric for evaluating code summarization (Shi et al., 2022). In this case, *n*-grams are matched up to $n = 4$. This matching is done cumulatively: the BLEU scores for *n*-grams for values of *n* = 1 through 4 are computed, and then the mean of these four scores is computed to get the final score.

It was found that different implementations of BLEU-4 lead to different results (Shi et al., 2022). Out of these variants, the sentence BLEU metric based on NLTK (Bird et al., 2009) smoothing method 4 had the highest correlation with human evaluation. The scores we report are computed by the implementation of this method in NLTK 3.8.1.



Description:
Extracts video ID from URL.

Code:

```
def get_vid_from_url(url):
      vid = match1(url,
      'https?://www.mgtv.com/(?:b|l)/\d+/(\d+).html')
      if not vid:
            vid = match1(url,
      'https?://www.mgtv.com/hz/bdpz/\d+/(\d+).html')
      return vid
```

Llama 2 Tokens from Description:
ext ract s video id from url .

Llama 2 Tokens From Function Definition of Code:
def get _ vid _ from _ url (url):

Figure 2: Subword tokenization (here performed by the Llama 2 tokenizer on the first line of the code and the description) exposes valuable information about which tokens are present in the description.

3. How well do LLMs perform on Code Summarization?

We now show how some recent LLMs perform on the standard code summarization dataset, CodeXGLUE. In this section, we study the extent to which the generated English summaries contain tokens that are copied from the subword-tokenized code. Following this, we examine how prevalent this token overlap is across the entire dataset. We further investigate whether the models also follow the trend of generating summaries with high token overlap with the input code and whether the BLEU scores of the generations are impacted by this overlap. Finally, we inspect which types of tokens are more likely to be involved in this overlap, since the tokens present in function names give away more information about the function than the tokens in the body of the function.

After determining the prevalence of high token overlap in the dataset and its impact on generation performance, we examine how much LLMs prioritize function names over the code structure and its control flow. We also evaluate how they perform when given code that has all the control structure removed, and in an extreme case if we only give the function definition as input. This will also help us answer our first two research questions about how much models leverage function names over the body of the code and how much models care about the code syntax and structure. Additionally, studying the relative performance of multiple LLMs of different parameter counts on these different types of input will give us a deeper insight into what they understand about code. While there is prior work showing how transformations like masking function names make code summarization harder (Sontakke et al., 2022), our work goes further and performs a deeper qualitative analysis to see what causes the model to perform well in the first place.

Dataset	CodeT5	PaLM 2	Llama 2 (7b)	Llama 2 (70b)
CodeXGLUE	17.66	19.23	22.36	22.41

Table 1: BLEU-4 across all models on CodeXGLUE

3.1. Experimental Setup

To examine a wide range of model sizes, we analyze CodeT5 (220M parameters), PaLM 2 (340B parameters), and Llama 2 (with 7B and 70B parameters):

CodeT5: CodeT5 (Wang et al., 2021b) is a pre-trained encoder-decoder model that uses objectives during pre-training like Identifier Tagging (the model has to predict whether each token in the input is an identifier or not) which make it more suitable for understanding code. It is pre-trained on CodeSearchNet. It is adapted to multiple downstream tasks including code summarization and code generation through multi-task learning.

PaLM 2: PaLM 2 (Anil et al., 2023) is an LLM made by Google AI for reasoning tasks, question answering, classification, translation, and code summarization. It was pre-trained on a large corpus of parallel multilingual text. As it is proficient in sequence-to-sequence tasks, we can make it perform code summarization with the right prompt.

Llama 2: Llama 2 (Touvron et al., 2023) is an LLM designed by Meta AI that can also be used for sequence-to-sequence tasks, including an instruct model that can be asked to perform any task with a prompt. It comes in three sizes - 7B, 13B and 70B.

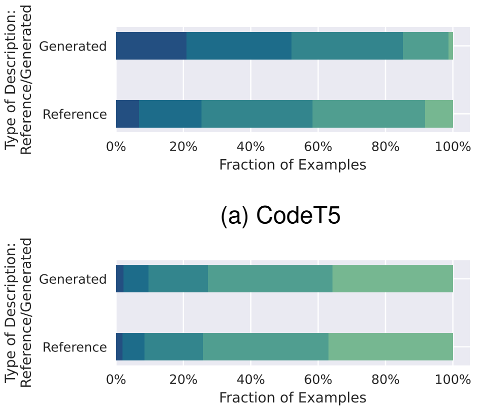
While CodeT5 is fine-tuned on this dataset, we use the other models PaLM 2 and Llama 2 in inference-only mode with few-shot prompting. We use the following prompt before each input code snippet - *"Pretend that you are a programmer writing Python functions. For a given Python function you have to generate a short documentation describing what the function does."*, along with ten examples from the dataset to show what kind of description we are looking for.

3.2. Overall Performance

To analyze these models, we first evaluate them on code summarization on the standard CodeXGLUE dataset. Table 1 shows that the BLEU-4 scores of the models discussed above range from 17.66 for the 220M-parameter model CodeT5 to 22.41 for the 70B-parameter Llama 2. The much smaller 7b parameter version of Llama 2 performs similarly at 22.36, while PalM 2 (340B parameters) achieves a score of 19.23, between CodeT5 and Llama 2.

3.3. Exploring if LLMs are copying tokens from code to description

We now analyze if these models are copying tokens from the code when generating their summaries. While copying in itself would not be a design flaw, if there is an instance of widespread copying, this would indicate that it is possible to perform well in this task without showing much understanding of the semantics of the code. We define our own metric called p_{copy}, which is the percentage of tokens (as generated by the corresponding model's tokenizer) in the description that was also present in the code.



(a) CodeT5

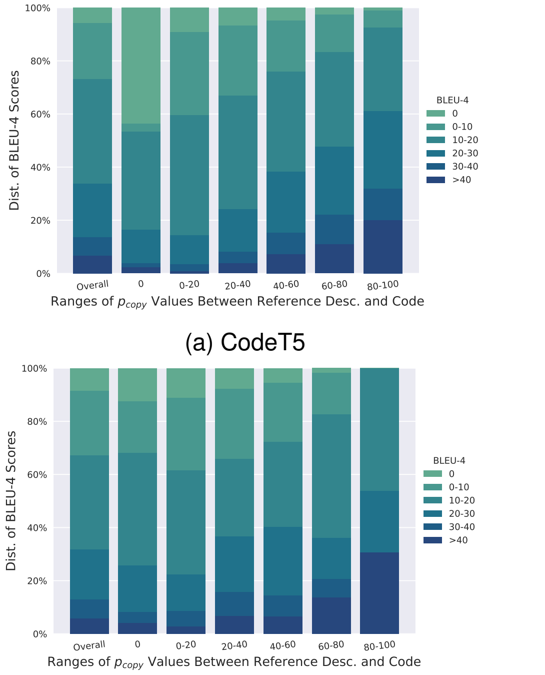
(b) Llama 2 (70b)

Figure 3: Distribution of p_{copy} in the dataset. Generated descriptions have much higher p_{copy} than Reference descriptions, and Llama 2's has fewer issues with copying tokens than CodeT5.

Figure 3 shows the distribution of p_{copy} in the reference descriptions written by human developers (bottom) and in the summaries produced by our two models (top), CodeT5 and Llama 2. Although copying is present in the human descriptions in both models (allowing the model to pick up on it during training), the models rely on it to a greater extent, since p_{copy} skews higher in the generated summaries. CodeT5 generates descriptions with much higher copying than Llama 2, but we see that even in the reference descriptions CodeT5 tokenizer gives much higher overlap than Llama 2. Therefore, it is not just the model but also the tokenizer that greatly influences the extent to which this copying strategy is employed.

3.4. Is copying tokens a viable strategy for summarization?

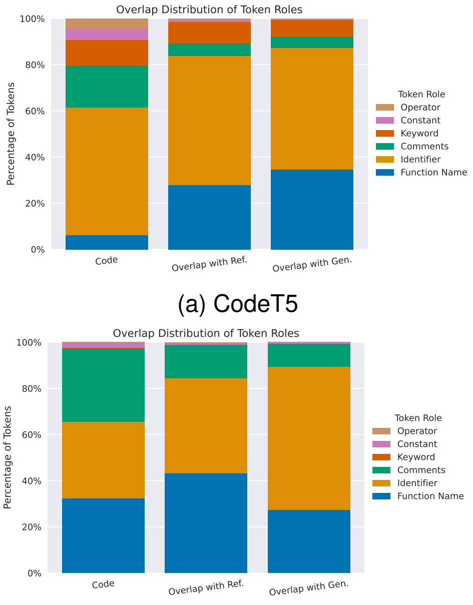
We see that the models learn to generate summaries by copying tokens from the code, but how effective is that strategy? For examples with high



(a) CodeT5

(b) Llama 2 (70b)

Figure 4: Distributions of BLEU-4 on all test examples and in different p_{copy} buckets. In general, higher p_{copy} leads to a higher BLEU-4 score.



(a) CodeT5

(b) Llama 2 (70b)

Figure 5: Distributions of token types in the overall CodeXGLUE code, and among the tokens that are copied to the reference and generated descriptions, according to the tokenizer and model of Code T5 and Llama 2 (70b)

p_{copy} in the reference description, a copying strategy should yield higher BLEU scores than for examples with lower p_{copy}. To see if this is the case, we split the test set into buckets based on the reference descriptions' p_{copy}, and plot the distribution of the model's BLEU-4 scores in each bucket (Figure 4). Figure 4a shows the results on the smallest model we have tested, CodeT5, and Figure 4b shows the results on Llama 2 (70b).

The first bar of the two plots in Figure 4 shows the overall distribution of BLEU-4 scores. The subsequent bars show the distribution of scores for a certain p_{copy} range. For example, the second bar shows the scores for examples where 0 to 10% of the tokens in the reference description are present in the code. We see that in general as p_{copy} increases, the BLEU-4 scores also increase. However, this correlation is less pronounced for Llama 2. This shows that both models tend to fall back on copying tokens from code to the description as a viable strategy, with the most difficult examples in the dataset being the ones where p_{copy} is zero, as in this bucket, half have a zero BLEU score.

3.5. Which tokens are copied?

We see that copying tokens from the code when generating descriptions can be a viable strategy. However, we need to explore which types of tokens are most likely to be copied. If it is mostly the function names that are being copied, that suggests the model relies most on the function definition and largely ignores the semantics of the code. We also want to see what kind of tokens get copied from the code in the reference descriptions in the dataset.

In Figure 5 we see the distribution of different token types like function names, identifiers, and comments for both CodeT5 and Llama 2 (70b). One main reason for the differences in their distributions, even in the reference descriptions, is the fact that they use different tokenizers. For example, the function names form a small part of the code snippets after tokenization by CodeT5 as seen in the first column in Figure 5a. On the other hand, the Llama 2 tokenizer (Figure 5b) allows for function names to be a much bigger part of the code. This pattern also holds true for the reference description in the dataset, with function names being more represented by the Llama 2 tokenizer. However, when we look at the generated descriptions, we see that CodeT5-generated summaries have a much higher presence of function names than Llama 2. This is probably due to fine-tuning, and CodeT5 learning that the optimal strategy for generating good summaries is to not only copy tokens from the code but to copy tokens from the function definition of the code. On the other hand, Llama 2 seems to prefer to rely more on identifiers, which can also contain useful information about the code semantics. Llama 2 also seems to better understand that the keywords in the code, while important to understand the semantics, are not supposed to be in the generated description.

4. How do Code Transformations Affect Performance?

We saw that the performance of LLMs on code summarization benefits from the high token overlap between the code and the reference description, and this overlap occurs primarily due to informative function names. To see how well they work when this information is unavailable, and also how strongly they rely on the internal structure of the code instead, we make several transformations to the code in the dataset. We then evaluate the performance of multiple LLMs under these scenarios.

In each transformation, we modify one aspect of the dataset used for training and evaluation of the classifier to understand what effect it has on performance. For example, the change in performance when we remove function names will show us how important function names are to the model's understanding of the code.

4.1. Code Transformations

Each of our four variants applies one transformation to the code to remove some information. The corresponding drop in performance indicates the importance of that information for the task. Figure 6 shows how an original snippet (6a), is affected by each transformation. We remove comments from the 32.5% of snippets that have them so that the models are forced to only look at the code.

Original Function Names: This is the unmodified code from the dataset (Figure 6a).

Obfuscated Function names: Function names often have a higher token overlap with the query than the rest of the code. We obfuscate them by replacing each character with the next character in the alphabet ('a' by 'b', 'b' by 'c' etc.). This forces the model to focus on other cues, like comments, variable names, or the actual structure of the code like for-loops and if-statements (Figure 6b).

Adversarial Function Names: We replace the original function name with the name of another function. Unlike obfuscation, this may mislead the model. Performance on this variant will tell us how well the model works when the function name is at odds with the actual operations performed in the body of the code (Figure 6c).

No Code Structure: We remove keywords (`if`, `return` etc.), operators (`not`, `>`, `+`), and delimiters (`,`,`,`, `etc.`), removing any information about the underlying logic of the program while keeping the rest of the code intact (Figure 6d).

No Function Body: We remove the entire body of the code and leave only the function definition. Here, we will observe how well the model performs when it only has the function name and its arguments available (Figure 6e).

```python
def get_vid_from_url(url):
    return match1(url, r'youtu\\.be/([^?/]+)') or
        match1(url, r'youtube\\.com/embed/([^/?]+)') or
        match1(url, r'youtube\\.com/v/([^/?]+)') or
        match1(url, r'youtube\\.com/watch/([^/?]+)') or
        parse_query_param(url, 'v') or
        parse_query_param(parse_query_param(url, 'u'), 'v')
```

(a) Original Function Names

```python
def hfu_wje_gspn_vsm(url):
    return match1(url, r'youtu\\.be/([^?/]+)') or
        match1(url, r'youtube\\.com/embed/([^/?]+)') or
        match1(url, r'youtube\\.com/v/([^/?]+)') or
        match1(url, r'youtube\\.com/watch/([^/?]+)') or
        parse_query_param(url, 'v') or
        parse_query_param(parse_query_param(url, 'u'), 'v')
```

(b) Obfuscated Function Names

```python
def train(url):
    return match1(url, r'youtu\\.be/([^?/]+)') or
        match1(url, r'youtube\\.com/embed/([^/?]+)') or
        match1(url, r'youtube\\.com/v/([^/?]+)') or
        match1(url, r'youtube\\.com/watch/([^/?]+)') or
        parse_query_param(url, 'v') or
        parse_query_param(parse_query_param(url, 'u'), 'v')
```

(c) Adversarial Function Names

```
get_vid_from_url url
match1 url r'youtu\\.be/([^?/]+)'
match1 url r'youtube\\.com/embed/([^/?]+)'
match1 url r'youtube\\.com/v/([^/?]+)'
match1 url r'youtube\\.com/watch/([^/?]+)'
parse_query_param url 'v'
parse_query_param parse_query_param url 'u' 'v'
```

(d) No Code Structure

```python
def get_vid_from_url(url)
```

(e) No Function Body

Figure 6: Different transformations of a Python function in CodeXGLUE

Examples in these variants will show us what part of the code LLMs tend to prioritize when given an input code snippet to analyze.

After computing the performance across multiple models and datasets, we also investigate how the models' performance is affected by the high token overlap between the code and the reference descriptions. Since this overlap can be exploited by copying tokens from the code to generate descriptions that look correct even without understanding the underlying semantics of the code, we explore how often this occurs. We also look at which parts of the code the models tend to copy tokens from most often i.e. function names versus variable names. Finally, we also looked at how our perception of the performance of these models is affected by our choice of similarity-based metrics which compare the descriptions to a human-generated docstring, since similarity alone may not be a perfect judge of whether a generated description is good or not.

4.2. Performance on Transformed Code

Table 2 shows the BLEU scores of CodeT5, PaLM 2, and Llama 2 (with 7b and 70b parameters) on

Variant	CodeT5	CodeT5 (FT)	PaLM 2	Llama 2 (7b)	Llama 2 (70b)
Original Function Names	**17.66**	**17.66**	**19.23**	**22.36**	**22.41**
Obfuscated Function Names	11.59	14.80	18.72	20.09	21.25
Adversarial Function Names	11.34	13.12	15.69	19.53	21.23
No Code Structure	13.96	16.57	11.92	15.11	18.42
No Function Body	13.94	15.27	11.90	14.93	18.16

Table 2: BLEU-4 across all models and variants

Variant	CodeT5	CodeT5 (FT)	PaLM 2	Llama 2 (7b)	Llama 2 (70b)
Original Function Names	**83.95**	**83.95**	**84.26**	**84.34**	**86.95**
Obfuscated Function Names	78.57	81.25	82.28	82.52	84.41
Adversarial Function Names	78.53	80.01	80.81	80.84	84.30
No Code Structure	81.32	82.86	79.36	79.66	83.25
No Function Body	81.14	82.40	79.14	79.72	83.14

Table 3: BERTScores across all models and variants

the different variants,

The first CodeT5 column shows the scores when the model is only fine-tuned on examples from the original dataset, whereas the second column (FT) shows the scores when the model is fine-tuned on the transformed examples so it knows what kind of code to expect as input.

Fine-tuning on transformed examples helps CodeT5. On the transformed variants, the performance of CodeT5 improves significantly after fine-tuning on examples from those variants, especially in Obfuscated Function Names. Not all variants have the same improvement, though. In Adversarial Function Names, the improvement is much smaller, suggesting that the presence of incorrect function names still misleads the model.

Scores drop for the transformed variants. We see that there is a drop in Obfuscated Function Names and an even bigger drop in Adversarial Function Names, suggesting that models struggle when the function name is hidden and are misled when given an incorrect function name, showing how important function names are to the models' understanding of the code semantics. However, the drop from Obfuscated to Adversarial is much higher for CodeT5 and PaLM 2 than Llama 2, showing that these models are more dependent on the function names. Section 3.5 will show that this may be explained by the fact that CodeT5 is much more dependent on the function names than Llama 2.

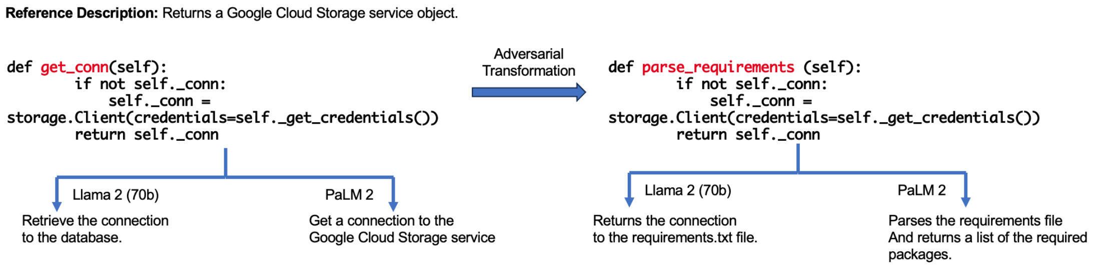
PaLM 2 and Llama 2 perform worse in No Function Body and No Code Structure compared to the other variants, whereas CodeT5 performs better. This shows that CodeT5 does not mind when given code that is incomplete or syntactically incorrect, unlike the larger models. However, despite performing well on these particular variants, CodeT5 is worse overall than Llama 2, showing that fine-tuning for higher performance on one narrow metric may not always give the best result.

CodeT5 is better at variants that have invalid code. No Function Body and No Code Structure have incomplete and syntactically incorrect code respectively. CodeT5 performs better at these variants than the ones where we just modify the function names. However, the other models perform much worse here. This shows that the larger variants care more about code syntax whereas a smaller model like CodeT5 looks mostly at function names. In fact, the largest model in our experiments, PaLM 2, performs the worst at these variants.

All models get misled by Adversarial Function Names. While the bigger models like Llama 2 and PaLM 2 perform better on this variant, they are still prone to be misled. We have shown some examples where it is obvious that the function name is wrong but the models still prioritize that over the semantics in the code. Figure 7 shows an example where both PaLM 2 and Llama 2 (70b) falter when the function name is changed.

5. Investigating An Alternative Metric

Although BLEU is a commonly used and well-established metric, it is very strict because it captures only exact matches between n-grams. To address this, a number of metrics have been recently developed that instead compute the similarity of token embeddings returned by a neural model. We investigate whether using such a metric changes the main findings from our experiments. We consider BERTScore, which uses BERT (Devlin et al., 2019) as the neural model.



Reference Description: Returns a Google Cloud Storage service object.

```
def get_conn(self):
        if not self._conn:
            self._conn =
storage.Client(credentials=self._get_credentials())
        return self._conn
```

```
def parse_requirements (self):
        if not self._conn:
            self._conn =
storage.Client(credentials=self._get_credentials())
        return self._conn
```

Retrieve the connection to the database.

Get a connection to the Google Cloud Storage service

Returns the connection to the requirements.txt file.

Parses the requirements file And returns a list of the required packages.

Figure 7: Examples of Llama 2 (70b) and PaLM 2 generations when given code from Adversarial Function Names. We see that even these larger models occasionally falter when given misleading function names. These are in cases where they gave acceptable responses when we made no perturbations in the code

BERTScore (Zhang et al., 2019b) is a precision/recall-inspired metric that uses a BERT model to compute token similarities between two summaries. To compute BERTScore, both summaries are given as input to the BERT model (we use the base uncased model of DistilBERT (Sanh et al., 2020)) to get their token embeddings (say x for reference and $\hat{x}$ for generated). Then recall (R) and precision (P) over these embedding sequences are computed by considering the maximal similarity of each reference (candidate) token to any candidate (reference) token:

$$R = \frac{1}{|x|} \sum_{x_i \in x} \max_{\hat{x_j} \in \hat{x}} x_i^T \hat{x_j} \quad P = \frac{1}{|x|} \sum_{\hat{x_j} \in \hat{x}} \max_{x_i \in x} x_i^T \hat{x_j}$$

The final BERTScore is the *F1* score (harmonic mean) of P and R:

$$\text{BERTScore} = \frac{2 \cdot P \cdot R}{P + R}$$

5.1. Overall Performance (BERTScore)

Table 3 shows the BERTScores for each model on the original test set. This correlates very strongly with the results in Table 2, with a Pearson Correlation of 0.87 and a Spearman's Rank Correlation of 0.86 between the 25 pairs of scores.

There is also a high correlation of BLEU and BERTScores between every pair of reference and generated descriptions we have in the original dataset across all models. We get a Pearson Correlation of 0.78 and a Spearman's Rank Correlation of 0.73. The heatmap in Figure 8 shows this positive correlation.

However, we observe that BERTScores are overall much higher and closer to each other here than was the case for BLEU scores. This is possibly due to BERTScore being a more forgiving metric, and most generations are assigned a very high BERTScore despite their quality. For example, the minimum BERTScore assigned to an example is 60, whereas there are several cases that have a BLEU score of zero.

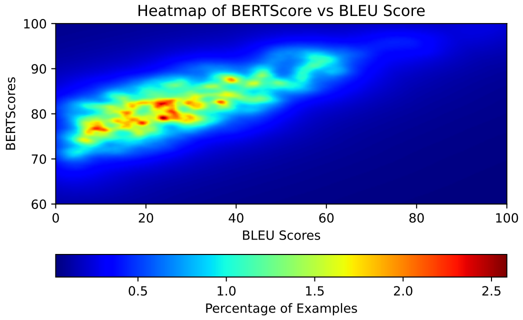


Figure 8: Heatmap of the BERTScore vs the BLEU-4 score for all examples in the test set containing the untransformed code.

Regardless of the metric used, performance generally increases with increasing the number of parameters, except Llama 2 outperforming PaLM 2. However, this increase in performance is not linear, with Llama 2(7b) being quite close to Llama 2(70b). As for variants, all models see a drop in performance in Obfuscated Function Names and an even bigger drop in Adversarial, though the second drop is less in Llama 2. And while Llama 2 and PaLM 2 justifiably underperform when given incomplete or incorrect code in No Function Body and No Code Structure, CodeT5 performance takes a much smaller hit, especially after fine-tuning.

5.2. Distribution of BERTScores

In Figure 9, we plot the distribution of BERTScores of all the model outputs on the original test data. The orange curve is the distribution of BERTScores between random pairs of reference descriptions and generated descriptions, and the blue curve shows the distribution of BERTScores between the reference descriptions and their corresponding descriptions.

We see that the median BERTScore even be-

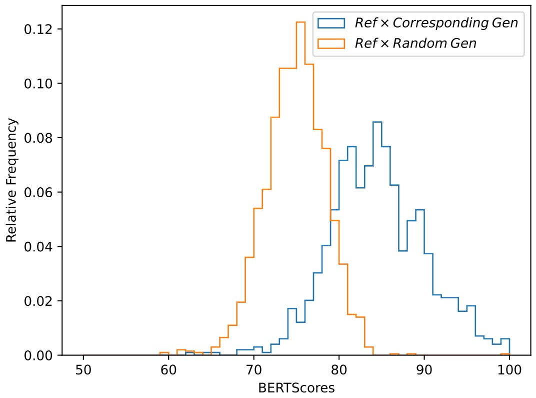


Figure 9: Distribution of BERTscores between reference descriptions and their corresponding generated descriptions, and between reference descriptions and a generated description from another random example across all models

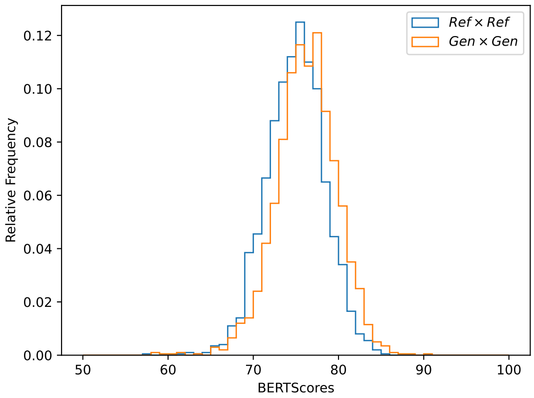


Figure 10: Distribution of BERTScores between two random reference descriptions and two random generated descriptions across all models

tween two unrelated descriptions is high, around 75.03. Therefore, even incorrect summaries are assigned very high BERTScores, and the threshold for good summaries is much higher than that. The median BERTScore between the reference descriptions and their corresponding generation as returned by all models is 84.33, which is more than 9 points higher, showing that this metric still shows discernment between related and unrelated descriptions. In Appendix A.1, we see that BLEU scores can discern between correct and incorrect descriptions as well as assign low scores (mostly zero) to randomly sampled generated descriptions.

Another interesting observation about BERTScores is that the average BERTScore is higher between two random generated descriptions than two random reference descriptions in the dataset, as seen in Figure 10. This tells us that the model generates descriptions with less diversity than that present in the dataset. In Appendix A.1, we see this is also true for BLEU scores.

6. Conclusion

This paper presented a series of experiments to gain a deeper insight into what makes current LLMs effective at code summarization. Section 4 suggests that LLMs often rely on function names and on shared tokens between the code and the description compared to the code structure to perform well. Relying on token overlap seems to work well because, at least in the standard datasets for these tasks, code and the corresponding descriptions often have high token overlap. Our experiments establish a clear trend between the token overlap of the code and descriptions and summarization performance.

We expect other LLM-based models, like PLBART (Ahmad et al., 2021), and CoText (Phan et al., 2021) to show similar behavior since we believe our results point to a feature of this entire class of models.

The current state of the art also falls short when the description is in a language other than English. Wang et al. (2022b) released a benchmark MCoNaLa, which contains a parallel corpus of code paired with multiple languages, and showed that current code generation models perform poorly for languages like Spanish, Japanese, and Russian. This may be because there are fewer tokens in the description that overlap with the code, so a model cannot learn to take advantage of informative function names and identifier names.

Finally, we also believe that human evaluation should be used in conjunction with building more comprehensive evaluation methodologies for code summarization, in order to measure the accuracy and the usefulness of a generated description, something that similarity-based metrics like BLEU and BERTScore cannot capture. While there has been work in that direction (Shi et al., 2022), the current practice still relies on comparing generated descriptions with a reference instead of measuring their actual usefulness to the user.

Acknowledgements

This work is supported by Agriculture and Food Research Initiative (AFRI) grant no. 2020-67021-32799/project accession no.1024178 from the USDA National Institute of Food and Agriculture.

Bibliographical References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL).

Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernandez Abrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vlad Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, Guy Gur-Ari, Steven Hand, Hadi Hashemi, Le Hou, Joshua Howland, Andrea Hu, Jeffrey Hui, Jeremy Hurwitz, Michael Isard, Abe Ittycheriah, Matthew Jagielski, Wenhao Jia, Kathleen Kenealy, Maxim Krikun, Sneha Kudugunta, Chang Lan, Katherine Lee, Benjamin Lee, Eric Li, Music Li, Wei Li, YaGuang Li, Jian Li, Hyeontaek Lim, Hanzhao Lin, Zhongtao Liu, Frederick Liu, Marcello Maggioni, Aroma Mahendru, Joshua Maynez, Vedant Misra, Maysam Moussalem, Zachary Nado, John Nham, Eric Ni, Andrew Nystrom, Alicia Parrish, Marie Pellat, Martin Polacek, Alex Polozov, Reiner Pope, Siyuan Qiao, Emily Reif, Bryan Richter, Parker Riley, Alex Castro Ros, Aurko Roy, Brennan Saeta, Rajkumar Samuel, Renee Shelby, Ambrose Slone, Daniel Smilkov, David R. So, Daniel Sohn, Simon Tokumine, Dasha Valter, Vijay Vasudevan, Kiran Vodrahalli, Xuezhi Wang, Pidong Wang, Zirui Wang, Tao Wang, John Wieting, Yuhuai Wu, Kelvin Xu, Yunhan Xu, Linting Xue, Pengcheng Yin, Jiahui Yu, Qiao Zhang, Steven Zheng, Ce Zheng, Weikang Zhou, Denny Zhou, Slav Petrov, and Yonghui Wu. 2023. Palm 2 technical report.

Steven Bird, Edward Loper, and Ewan Klein. 2009. *Natural Language Processing with Python*. O'Reilly Media Inc.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Brian P. Eddy, Jeffrey A. Robinson, Nicholas A. Kraft, and Jeffrey C. Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 13–22.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018a. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018b. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944.

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. 2020. Graphcodebert: Pre-training code representations with data flow. *CoRR*, abs/2009.08366.

Rajarshi Haldar, Lingfei Wu, JinJun Xiong, and Julia Hockenmaier. 2020. A multi-perspective architecture for semantic code search. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8563–8568, Online. Association for Computational Linguistics.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 200–210, New York, NY, USA. Association for Computing Machinery.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred api knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, page 2269–2275. AAAI Press.

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*.

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)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 795–806. IEEE Press.

Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. 2021. Deep graph matching and searching for semantic code retrieval. *ACM Trans. Knowl. Discov. Data*, 15(5).

Shangqing Liu, Xiaofei Xie, Lei Ma, Jingkai Siow, and Yang Liu. 2021. Graphsearchnet: Enhancing gnns via capturing global dependency for semantic code search. *arXiv preprint arXiv:2111.02671*.

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.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 314–319, Taipei, Taiwan. Asian Federation of Natural Language Processing.

Arvind Narayanan and Sayash Kapoor. 2023. Gpt-4 and professional benchmarks: The wrong answer to the wrong question.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*.

OpenAI. 2023. Gpt-4 technical report.

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*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task learning with code-text transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 40–47, Online. Association for Computational Linguistics.

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(1).

Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 390–401, New York, NY, USA. Association for Computing Machinery.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2020. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1597–1608, New York, NY, USA. Association for Computing Machinery.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning.

Anna Abad Sieper, Omar Amarkhel, Savina Diez, and Dominic Petrak. 2020. Semantic code search with neural bag-of-words and graph convolutional networks. In *SKILL 2020 - Studierendenkonferenz Informatik*, pages 103–115, Bonn. Gesellschaft für Informatik e.V.

Ankita Nandkishor Sontakke, Manasi Patwardhan, Lovekesh Vig, Raveendra Kumar Medicherla, Ravindra Naik, and Gautam Shroff. 2022. Code summarization: Do transformers really understand code? In *Deep Learning for Code Workshop*.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 43–52, New York, NY, USA. Association for Computing Machinery.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 13–25. IEEE Press.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021a. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*.

Yu Wang, Yu Dong, Xuesong Lu, and Aoying Zhou. 2022a. Gypsum: learning hybrid representations for code summarization. In *Proceedings of the 30th IEEE/ACM International Conference* *on Program Comprehension*, ICPC '22, page 12–23, New York, NY, USA. Association for Computing Machinery.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021b. 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*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F. Xu, and Graham Neubig. 2022b. Mconala: A benchmark for code generation from multiple natural languages.

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F. Xu, and Graham Neubig. 2023. MCoNaLa: A benchmark for code generation from multiple natural languages. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 265–273, Dubrovnik, Croatia. Association for Computational Linguistics.

Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code summarization with structure-induced transformer. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 1078–1090, Online. Association for Computational Linguistics.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019a. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2019b. Bertscore: Evaluating text generation with BERT. *CoRR*, abs/1904.09675.

A. Appendix

A.1. Distribution of BLEU Scores

Similar to BERTScores in Section 5.2, we plot the distribution of BLEU scores between random pairs of reference descriptions and generated descriptions, as well as between the reference descriptions and their corresponding descriptions in Figure 11. We see that unlike BERTscores, generated summaries that are from a different example are assigned very low, mostly zero scores. This shows that BLEU can not only discern between a correct and incorrect description better than BERTScore but can also identify an incorrect generation by assigning it a zero score.

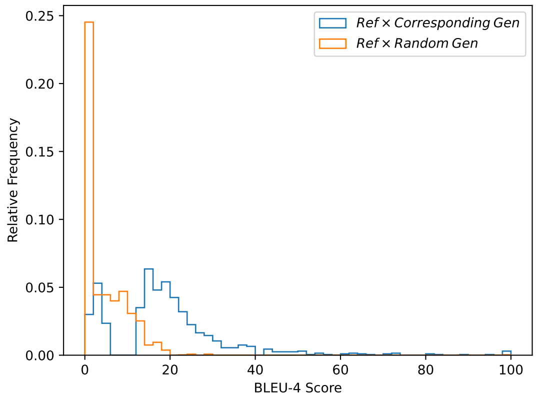


Figure 11: Distribution of BLEU scores between reference descriptions and their corresponding generated descriptions, and between reference descriptions and a generated description from another random example across all models and variants

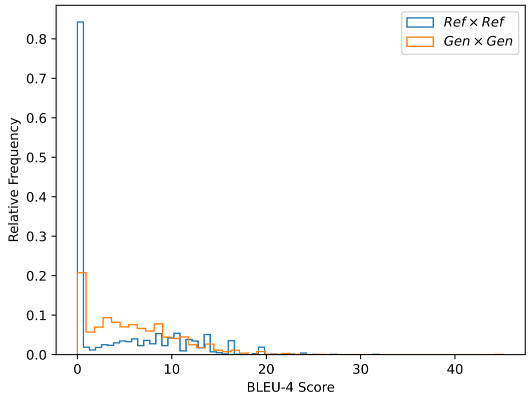


Figure 12: Distribution of BLEU Scores between two random reference descriptions and two random generated descriptions across all models and variants

In Figure 12, we see that the distribution of BLEU scores between two randomly sampled generated descriptions is higher than those between two randomly sampled reference descriptions, showing that the models have a problem of producing generations that are less diverse than the data they were trained on. We made a similar observation in Section 5.2 which shows that the problem persists independent of the metric used.