Code Summarization: Do Transformers Really Understand Code?

Abstract

Recent approaches for automatic code summarization rely on fine-tuned transformer based language Models often injected with program analysis information. We perform empirical studies to analyze the extent to which these models understand the code they attempt to summarize. We observe that these models rely heavily on the textual cues present in comments/function names/variable names and that masking this information negatively impacts the generated summaries. Further, subtle code transformations which drastically alter program logic have no corresponding impact on the generated summaries. Overall, the quality of the generated summaries even from State-Of-The-Art models is quite poor, raising questions about the utility of current approaches and datasets.

1 Introduction

Code summaries play an important role in program understanding, maintenance and debugging. Recent work towards automated code summarization adopts two primary approaches: (i) Fine tuned Language Models (LM) or (ii) Deep models that inject Program Analysis Information (PAI) to facilitate better understanding of program semantics. The resulting models yield BLEU scores (Papineni et al., 2002) ranging from 7 to 45 on publicly available datasets. In this paper, we perform an empirical analysis to evaluate the code understanding capabilities of these models for summary generation. We apply code transformations that change the underlying logic of the input code and observe the resulting change in the summaries and associated BLEU scores. Conversely, we observe the change in generated summaries when we subject the code to semantic preserving transformations such as replacing variable names. We also observe the effect on the model performance after removing the data leakage problems specifically in TL-CodeSum (Hu et al., 2018b) and Python (Wan et al., 2018) datasets (refer Section A of the Appendix). We make the following observations, which may prove useful for the code summarization research community:

1. The BLEU scores of existing code summarization models on reported datasets are very low (in the range of 5 to 8), especially for out-of-domain data where train and test codes belong to distinct projects (Liu et al., 2020). This calls into question the utility of these models for real-life applications.

2. Testing the models on codes with semantic-preserving transformations negatively impacts the BLEU score (average drop of 7). This is not only true for the LM based models but also for the models which claim to understand the program structure by injecting PAI. This likely points to prevailing models performing ‘short-cut’ learning by relying on the inductive biases from meaningful function and variable names.

3. Training with codes after semantic-preserving transformations leads to no improvements in BLEU over the original biased models. This indicates that the models are extremely reliant on textual cues and are unable to learn code semantics when these are removed. This highlights the need for designing better training strategies to facilitate code understanding, such as self-supervision with semantic-preserving and disrupting transformations.

4. Transformations which change the semantics of the code have very minimal impact on the BLEU scores (average drop of 0.13), demonstrating that the models are not paying much attention to code semantics while generating the summaries.

5. Getting rid of the leakages in the datasets leads to a large drop in the BLEU scores (average 11), highlighting the need for carefully designing datasets where there is no code overlap across the splits, not only in terms of codes having the same surface forms (syntax), but also in terms of codes with the same semantics. Such datasets would better evaluate the generalization capabilities of dif-
Table 1: Example of transformed code from Python dataset (Wan et al., 2018) and TL-CodeSum (Hu et al., 2018b). Summaries generated by SIT (Wu et al., 2021) and PLBART (Ahmad et al., 2021b) with the transformations and experiments (Sections 4). GT: Ground-Truth summary, EXP: Experiment, Te: Test set, Tr: Training set, SPT: Semantic Preserving Transformations, SDT: Semantic Disrupting Transformations, FN: Function Name, VN: Variable Names, 1. SPT-FN (Green), 2. SPT-VN (Blue), 3. SDT (Red).

Different summarization approaches. Datasets should also facilitate learning of code semantics and prevent over reliance on textual correlations.

2 Related Work

Code Summarization Datasets Publicly available datasets such as TL-CodeSum (Hu et al., 2018b), Python (Wan et al., 2018), Funcom (LeClair et al., 2019), CCSD (Liu et al., 2020), CodeSearchNet (Husain et al., 2019) and CodeXGLUE (Lu et al., 2021) have function-summary pairs collected from open source GitHub repositories. Current datasets have some serious limitations such as: (i) having code comments as a part of the source code (CodeSearchNet), (ii) data leakage i.e. having common code-summary pairs in train and test set (TLCodeSum, Python), (iii) meaningful function and variable names having textual correlations with the words in the summary (iv) Highly abstract summaries that are divorced from the code logic (v) domain specific summaries that are not obtainable from code and require external knowledge out side the code logic for summary generation (CodeNet (Puri et al., 2021)) (vii) no datasets for legacy programming languages like COBOL. The details of limitations with examples are provided in Appendix Section A. We perform our analysis on CodeSearchNet, TL-CodeSum and Python datasets for Python and Java programming languages. The data statistics are provided in Appendix Section A.

Code Summarization Approaches Neural code summarization approaches utilize one of the following: (i) Language Models (LM) pre-trained with monolingual programming data and further fine-tuned with code summary pairs or (ii) Deep models (Transformers, LSTMs, Graph Neural Networks) exploiting program analysis information in terms of Abstract Syntax Trees (ASTs), data dependencies

1https://github.com/
and/or control flows to incorporate code semantics. Details are provided in Appendix Section B. For our analysis, we include one model from each of the above categories, namely PLBART (Ahmad et al., 2021b) and Structure Induced Transformers (SIT) (Wu et al., 2021).

3 Transformations

We perform causal analysis by tweaking the code using the following transformations to preserve or change code semantics and then observe the effect on the resulting summary and BLEU scores. Table 1 demonstrates the transformations.

SPT are the set of Semantic Preserving Transformations, which include (i) \( CC \) removing the Code Comments from 17% of the codes in CodeSearchNet (ii) \( FN \) replacing meaningful user-defined Function Names with more generic (but unique) function names, and (iii) \( VN \) replacing meaningful user-defined local Variable Names with more generic variable names, unique per existing variable name, such that data-dependencies are preserved. Generic names carry no semantics and are selected from the existing model vocabulary. \( FN \) and \( VN \) are applicable to all codes in all the datasets.

SDTs are the set of Semantic Disrupting Transformations, which include (i) replacing an arithmetic and relational operator with its inverse (For example, replacing + with − or equality == with inequality !=, etc) and (ii) replacing a logical operator with its complement (For example, replacing \( AND \) with \( OR \)) such that the code execution is not hampered but the semantics of the code is disrupted. \(~78\%, 68\%, 40\% \) and \(43\%\) of codes in CodeSearchNet-Java and Python, TlCodeSum and Python datasets are modified with SDT. The intent is to observe the change in BLEU, by comparing the summaries generated by the models with the transformed and original codes, against the original ground truth summaries, which are retained for both the transformations.

4 Experimental Setup

We perform the following experiments:

**EXP-Te-DL** We address the Data Leakage (DL) in the datasets by removing 38.49% Java and 21.66% Python code snippets from the Test Set of TL-CodeSum and Python datasets, that syntactically match with the code snippets in the train set resulting in inflated BLEU scores. We expect a drop in average BLEU scores after filtering these samples from the test set. We use this filtered test set for the following experiments.

**EXP-Te-SPT** Models trained on the original train data are tested on the SPT transformed Test Set. For an ideal model BLEU scores should not change from unmodified trainset-testset scores as SPTs are semantic preserving.

**EXP-Tr-SPT** Models trained with the SPT transformed Train Set are tested on the original test data. Since the model can no longer exploit function and variable names to generate summaries, this experiment should test whether the model is capable of understanding the program logic and if so, improve the BLEU scores.

**EXP-TrTe-SPT** Models trained with the SPT transformed Train Set are tested on the SPT transformed Test Set. Along similar lines of EXP-TrSPT, improvements in the BLEU scores over unmodified trainset-testset results would indicate that the model better understands code.

**EXP-Te-SDT** Models trained on the original train data are tested on the SDT transformed Test Set. As the \( SDT \) changes the semantics of the programs, if the model understands the code semantics, the resulting summaries generated by the model should be different from the original ground truth summaries, leading to a drop in the BLEU scores.

To programatically transform the codes, we use javalang\(^2\) and ast\(^3\) packages. We detect and replace the function and variable names by constructing an AST for the functions. We detect the logical and arithmetic operators by using regex\(^4\). SIT\(^5\) is originally trained on TL-CodeSum and Python dataset and PLBART on CodeSearchNet. For having comparisons across the models, we fine-tune pre-trained PLBART\(^6\) with TL-CodeSum and Python, where the codes are tokenized using the Tree-sitter tokenizer \(^7\). For fair comparison, we use the same set-of hyper-parameters described in the original papers (Ahmad et al., 2021a; Wu et al., 2021) and run the experiments on one Nvidia Tesla V100 32 GB GPU. SIT and PLBART take \(~34 and 8\) hours to train. Experiments on CodeSearchNet are performed with only PLBART as the program

\(^2\)https://github.com/c2nes/javalang
\(^3\)https://docs.python.org/3/library/ast.html#
\(^4\)https://github.com/python/cpython/blob/3.10/Lib/re.py
\(^5\)https://github.com/gingasan/sit3
\(^6\)https://github.com/wasiahmad/PLBART
\(^7\)https://github.com/tree-sitter/tree-sitter
Table 2: Results on Python (Wan et al., 2018), TL-CodeSum (Hu et al., 2018b) and CSN: CodeSearchNet (Husain et al., 2019). PL: Programming Languages, EXP: Experiment, Te: Test set, Tr: Training set, SPT: Semantic Preserving Trans, SDT: Semantic Disrupting Trans, DL: Data Leakage, FN: Function Name, VN: Variable Names, CC: Code Comments. *Results from (Wu et al., 2021), #Results from (Ahmad et al., 2021a).

<table>
<thead>
<tr>
<th>Model Method</th>
<th>PL &amp; Dataset</th>
<th>Python</th>
<th>Java TL-CodeSum</th>
<th>Python CSN</th>
<th>Java CSN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SIT</td>
<td>PLBART</td>
<td>SIT</td>
<td>PLBART</td>
<td>PLBART</td>
</tr>
<tr>
<td>Original</td>
<td>BLEU Drop</td>
<td>BLEU</td>
<td>BLEU Drop</td>
<td>BLEU</td>
<td>BLEU</td>
</tr>
<tr>
<td>EXP-Te-DL</td>
<td>34.11</td>
<td>-</td>
<td>25.53</td>
<td>-</td>
<td>20.61</td>
</tr>
<tr>
<td>EXP-Te-SPT</td>
<td>15.35</td>
<td>8.26</td>
<td>16.10</td>
<td>6.89</td>
<td>11.46</td>
</tr>
<tr>
<td>SPT − FN</td>
<td>18.26</td>
<td>5.35</td>
<td>16.61</td>
<td>6.38</td>
<td>12.81</td>
</tr>
<tr>
<td>SPT − VN</td>
<td>18.39</td>
<td>5.22</td>
<td>21.20</td>
<td>1.79</td>
<td>14.08</td>
</tr>
<tr>
<td>SPT − CC</td>
<td>23.61</td>
<td>0.00</td>
<td>22.80</td>
<td>0.19</td>
<td>19.34</td>
</tr>
<tr>
<td>EXP-Te-DL</td>
<td>23.61</td>
<td>10.5</td>
<td>22.99</td>
<td>2.54</td>
<td>19.34</td>
</tr>
<tr>
<td>EXP-Te-SPT</td>
<td>15.35</td>
<td>8.26</td>
<td>16.10</td>
<td>6.89</td>
<td>11.46</td>
</tr>
<tr>
<td>SPT − FN</td>
<td>18.26</td>
<td>5.35</td>
<td>16.61</td>
<td>6.38</td>
<td>12.81</td>
</tr>
<tr>
<td>SPT − VN</td>
<td>18.39</td>
<td>5.22</td>
<td>21.20</td>
<td>1.79</td>
<td>14.08</td>
</tr>
<tr>
<td>SPT − CC</td>
<td>23.61</td>
<td>0.00</td>
<td>22.80</td>
<td>0.19</td>
<td>19.34</td>
</tr>
</tbody>
</table>

**5 Result and Analysis**

Table 1 illustrates the examples of Java and Python codes from TL-CodeSum and Python datasets and the corresponding transformed code with SPT and SDT. However, it should be noted that, we never perform both transformations simultaneously. EXP-Te-SPT summaries do not match with the ground truth and are inferior to the original model summaries, showcasing the negative influence of SPT. EXP-Tr-SPT and EXP-TrTe-SPT summaries are closer to the ground truth as compared to EXP-Te-SPT demonstrating the positive effect of an SPT transformed train set. Summaries of EXP-Te-SDT remain unchanged, showcasing no influence of SDT.

Table 2 illustrates the smoothed BLEU-4 scores for all the experiments. As expected, EXP-Te-DL showcases substantial drop in BLEU (average 11) after removing data leakage. The BLEU scores for SIT and PLBART models are comparable. This questions the benefit of infusing program analysis information into the model as opposed to using a fine tuned LM. As CodeSearchNet has no data leakage, there are no drops in the BLEU with EXP-Te-DL. After EXP-Te-DL, the overall BLEU scores are in the range of 16-24, questioning their utility for real-life applications.8

There is a further drop in BLEU (7.09) with EXP-Te-SPT showcasing the role comments and meaningful function/variable names are playing in summary generation. The ablation experiments demonstrate that function names have the most impact on generation followed by variable names and comments leading to 4.99, 2.85 and 0.35 average drops in BLEU score. The drops in the BLEU scores with EXP-Tr-SPT (2.62) and EXP-TrTe-SPT (3.27) are less as compared to that of with the EXP-Te-SPT proving that training with more generic function and variable names is helping the model to better understand the semantics. However, no improvements in BLEU over EXP-Te-DL demonstrates the need for designing better preprocessing and training strategies for the task. With EXP-Te-SDT the drops in BLEU are very minor (0.13) showcasing that the transformations which change the semantics of the code (SDT) have no effect on the summaries and thus it is questionable if the models are paying any attention to the logic/semantics of the code.

**6 Conclusion**

Through empirical studies of SOTA code summarization models, we demonstrate the negative impact of semantics preserving code transformations on the generated summaries. Additionally, we demonstrate that semantic disrupting transformations leave the generated summaries largely unchanged. This questions the code understanding capabilities of these models and points to the need for better training strategies to facilitate code understanding and well-curated datasets. The SPT and SDT transformations devised here offer some ideas for potential self supervised strategies to better train these models. The current analysis is restricted to a subset of code-summary datasets, programming languages, neural models and the defined transformations. We are working on extending it to generalize our observations.

---

8https://cloud.google.com/translate/automl/docs/evaluate#bleu
References


Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. Editsum: A retrieve-and-edit framework for source code summarization.


**A Limitations of Code Summarization Datasets**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Language</th>
<th>Train</th>
<th>Valid</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>Python</td>
<td>57,203</td>
<td>19,067</td>
<td>19,066</td>
</tr>
<tr>
<td>TL-CodeSum</td>
<td>Java</td>
<td>69,708</td>
<td>8,714</td>
<td>8,714</td>
</tr>
<tr>
<td>CodeSearchNet</td>
<td>Python</td>
<td>251,820</td>
<td>13,914</td>
<td>10,955</td>
</tr>
<tr>
<td>CodeSearchNet</td>
<td>Java</td>
<td>164,923</td>
<td>5,183</td>
<td>14,014</td>
</tr>
</tbody>
</table>

Table 3: Dataset statistics for Python (Wan et al., 2018), TL-CodeSum (Hu et al., 2018b) and CodeSearchNet(CSN) (Husain et al., 2019)

Table 3 provides the statistics of the datasets, which we have used for our analysis. Publicly available code-summary datasets enlisted in section 2 have the following lacuna:

1. CodeSearchNet (Husain et al., 2019) have code comments in the codes and need pre-processing to avoid biases. For example, the code-summary pair of Java in CodeSearchNet depicted in example (a) of Table 4, has comments in the code which have textual correlations with the summary.

2. TL-CodeSum (Hu et al., 2018b) and Python (Wan et al., 2018) datasets have data leakages. Examples (b) and (c) in Table 4 depict the Java and Python example code-summary pairs from the datasets which are common across the train and test splits.

3. As depicted in Table 1, current datasets have function and variable names that have textual correlations with the summaries, leading to an inductive bias.

4. As collected from Github repositories, the summaries of existing datasets (Table 1) are in the form of code-comment pairs where the code snippets are at function-level. For models to learn the underlying program logic, we need the code-summary pairs in the form of complete code with more abstract code-level summaries. For example the code-summary pair in the example (d) in Table 4, from the project CodeNet (Puri et al., 2021) provides a problem description of the complete code summarizing the underlying logic of the code.

5. The code summaries require external domain knowledge, which is not available in the source code. For example, in CodeNet dataset the problem descriptions come from variety of domains. It is impossible to predict the domain-specific components of the summaries from the codes as an input, which require external domain knowledge. For example, from the code illustrated in example (d) of Table 4, to generate the illustrated ground truth summary external domain knowledge in terms of the meaning of ‘parallel lines’ (lines having same slope and the definition of slope computation) is required.

6. As the existing datasets may not have domain overlaps, models trained on one dataset do not perform well on the other (out-of-domain data) as depicted by the codes in examples (e) Python and (f) Java in table 4 from CodeNet and the corresponding ground truth and predicted summaries by PLBART trained on CodeSearchNet. Since there no domain overlap between these datasets, the predicted summaries do not match with the ground truth and most of the time are meaningless.

7. The above listed code-summary dataset addresses only high-resource programming languages such as Python, Java, Javascript, PHP, Ruby, Go and C#. For practical applications, where there is a need to maintain and debug legacy codes we need datasets that would facilitate summarization of legacy languages such as COBOL.

**B Code Summarization Approaches**

Neural code summarization approaches can be majorly divided into: (i) Language Model (LM) based (ii) Deep models exploiting PAI to incorporate code semantics. LM based approaches such as PLBART (Ahmad et al., 2021b), CodeT5 (Wang et al., 2021), CoText (Phan et al., 2021), ProphetNet-Code (Qi
Table 4: Code-Summary Examples depicting lacuna of existing datasets (CodeSearchNet (Husain et al., 2019), TL-CodeSum (Hu et al., 2018b), Python (Wan et al., 2018) and CodeNet (Puri et al., 2021)).

et al., 2021), CodeTrans (Elnaggar et al., 2021), and CodeBERT (Feng et al., 2020), pre-train a LM on mono-lingual programming language data collected from Github and/or StackOverflow with pre-training objectives such as token masking, dele-

9https://stackoverflow.com/
tion, or infilling (Lewis et al., 2019). They are further fine-tuned on code-summary pairs to learn code-text alignment and infer summaries for unseen codes.

Approaches exploiting PAI use LSTMs (Hu et al., 2018a; Alon et al., 2018; LeClair et al., 2019), Transformers (Ahmad et al., 2020; Wu et al., 2021; Zügner et al., 2021; LeClair et al., 2019; Zhang et al., 2020), Graph Neural Networks (GNNs) (Liu et al., 2020; LeClair et al., 2020; Wang et al., 2020) or a combination of these (Choi et al., 2021; Shi et al., 2021) and inject PAI in the form of Abstract Syntax Trees (ASTs), data dependencies and/or control flows. The PAI is provided in the form of flattened ASTs using pre-ordered or structure based traversal (Hu et al., 2018a; Alon et al., 2018; LeClair et al., 2019), pre-defined adjacency matrices with the edges as an inductive bias for the attention between nodes (tokens) (Wu et al., 2021), relative positional encodings between adjacent nodes (Zügner et al., 2021) or feeding the Code Property Graphs (CPGs) to the model (Liu et al., 2020). Some studies also enhance these models by incorporating information retrieval techniques (Li et al.; Zhang et al., 2020; Liu et al., 2020), where the prototype summaries of similar codes are retrieved from a database and are edited by using an encoder-decoder setting.