CODE SUMMARIZATION: DO TRANSFORMERS REALLY UNDERSTAND CODE?

Ankita Sontakke, Manasi Patwardhan, Lovekesh Vig, Raveendra Kumar Medicherla, Ravindra Naik, Gautam Shroff  
TCS Research, India  
{ankita.sontakke, manasi.patwardhan, lovekesh.vig, raveendra.kumar, rd.naik, gautam.shroff}@tcs.com

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 (SOTA) 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 (Ahmad et al., 2021a; Wu et al., 2021; Zügner et al., 2021; Phan et al., 2021; Qi et al., 2021; Elnaggar et al., 2021) towards automated code summarization adopts two primary approaches: (i) Fine tuned Language Models (LM) or (ii) Deep models that inject Program Analysis Information (PAI) claiming to facilitate better understanding of program semantics. In this paper, we perform an empirical analysis to evaluate the code understanding capabilities of these models for summary generation. We assume that code summary is a reflection of not only the textual cues present in the code but also the underlying semantics of the code. Based on this assumption, we hypothesize that any semantics preserving code transformations would have minimal effect on code summaries, whereas code transformations that change the underlying logic of the input code would alter the summaries meaningfully to reflect and capture the change in the logic. Conversely, our observations lead to rejection of these hypotheses. Following observations of our studies may prove useful for the code summarization research community:

1. The BLEU scores of existing code summarization models reported on public datasets are very low (in the range of 11.17 to 26.53) (Wang et al., 2021), especially for out-of-domain data (in the range of 5.45 to 7.85) 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 by replacing meaningful function and variable names with generic vocabulary words, 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 summarization models performing 'short-cut' learning by primarily relying on the inductive biases from meaningful function and variable names. Such heavy reliance may be acceptable for domain specific application codes, where the developer ensures the usage of meaningful function and variables names. However, when code summarization heavily depends upon the underlying algorithm or logic implemented in the code such reliance would be detrimental for code comprehension.

3. Training with codes after semantic preserving transformations leads to no improvements in BLEU over the original models. This indicates that the models are extremely reliant on textual cues and are unable to understand the underlying logic of the code when these are removed. This highlights the
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).

need for designing better training strategies to facilitate the understanding of the code logic, such as self-supervision with both semantic preserving and disrupting transformations.

4. Transformations which change the semantics of the code in terms of perturbing the operators in expressions (arithmetic and logic) and thus impacting the high-level logic 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 in terms of having exactly same code-summary pairs in the splits leads to a large drop in the BLEU scores (average 11), highlighting the need for carefully designing datasets. Datasets should completely avoid code overlaps across the splits not only in terms of codes having same surface forms but also codes which belong to a same project sharing semantics. Datasets should also facilitate learning of code semantics and prevent over reliance on textual correlations.

2 RELATED WORK

2.1 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. These datasets have the following lacuna:

1. Having code comments as a part of the source code: CodeSearchNet (Husain et al., 2019) have code comments in the codes and need pre-processing to avoid biases. For example, the code-
2. Data leakage: TL-CodeSum (Hu et al., 2018b) and Python (Wan et al., 2018) datasets have data leakages with duplicate code-summary pairs across train and test splits. Examples (b) and (c) in Table 3 depict example Java and Python code-summary pairs present in the train and test splits of the mentioned datasets.

3. Meaningful function and variable names having textual correlations with the words in the summary: As depicted in Table 1, the code snippets in the current datasets have meaningful function and variable names that have textual correlations with the Ground Truth (GT) summaries, leading to an inductive bias.

4. Highly abstract summaries that are divorced from the code logic: 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 from CodeNet (Puri et al., 2021) in example (d) in Table 3 provides a problem description of the complete code summarizing the underlying logic of the code.

5. Domain specific summaries that are not obtainable from code requiring external knowledge beyond the code logic for summary generation: In the CodeNet dataset the problem descriptions come from a variety of domains. Thus, 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 3 to generate the illustrated GT 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. No out-of-domain splits: 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 3 from CodeNet and the corresponding GT and predicted summaries by PLBART trained on CodeSearchNet. Since there no domain overlap between these datasets, the predicted summaries do not match with the GT summary and most of the time are meaningless.

7. No datasets for legacy programming languages like COBOL: 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.

We perform our analysis on CodeSearchNet, TL-CodeSum and Python datasets for Python and Java programming languages. The data statistics are provided in Table 2.

### 2.2 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 and/or control flows 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 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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Language</th>
<th>Train</th>
<th>Valid</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python (Wan et al., 2018)</td>
<td>Python</td>
<td>57,203</td>
<td>19,067</td>
<td>19,066</td>
</tr>
<tr>
<td>TL-CodeSum (Hu et al., 2018b)</td>
<td>Java</td>
<td>69,708</td>
<td>8,714</td>
<td>8,714</td>
</tr>
<tr>
<td>CodeSearchNet (Husain et al., 2019)</td>
<td>Python</td>
<td>251,820</td>
<td>13,914</td>
<td>14,918</td>
</tr>
<tr>
<td>CodeSearchNet (Husain et al., 2019)</td>
<td>Java</td>
<td>164,923</td>
<td>5,183</td>
<td>10,955</td>
</tr>
</tbody>
</table>

Table 2: Dataset statistics
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.

Table 3: 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)). (a) Red color demonstrates the lexical match between the words in the code with the words in the ground-truth summary (b) and (c) shows example Java and python code-summary pairs which are present in both train and test splits of the mentioned datasets depicting leakage (d) Blue color demonstrates code logic implemented to detect parallel lines (e) and (f) showcases examples from CodeNet dataset and respective summary generated by PLBART model trained on CodeSearchNet, showcasing out-of-distribution performance (For details refer to Section 2.1).
Approaches exploiting Program Analysis Information (PAI) use LSTMs \cite{Hu18a,Alon18b,LeClair19a}, Transformers \cite{Ahmad20a,Wu21b,Zugner21a,LeClair19a,Zhang20a}, Graph Neural Networks (GNNs) \cite{Liu20a,LeClair19b,Wang20a} or a combination of these \cite{Choi21a,Shi21a} 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 \cite{Hu18a,Alon18b,LeClair19a}, pre-defined adjacency matrices with the edges as an inductive bias for the attention between nodes (tokens) \cite{Wu21b}, relative positional encodings between adjacent nodes \cite{Ahmad20a,Zugner21a} or feeding the Code Property Graphs (CPGs) to the model \cite{Liu20a}. Some studies also enhance these models by incorporating information retrieval techniques \cite{Li20a,Zhang20a,Liu20a}, where the prototype summaries of similar codes are retrieved from a database and are edited by using an encoder-decoder setting. For our analysis, we include one model from each of the above categories, namely PLBART \cite{Ahmad21a} and Structure Induced Transformers (SIT) \cite{Wu21b}.

2.3 Adversarial Program Perturbations

This section throws some light on the literature which explores the idea of investigating adversarial program perturbations to break the AI models built for code intelligence tasks. For program synthesis task, with NL as an intent \cite{Liguori21a}, shows that parsing NL with generic variable names achieves better synthesis. \cite{Karmakar21a} probe the codes to evaluate if the code representations provided by the pre-trained models encode the surface level, syntactic, structural and semantic code characteristics and points at the need for designing better pre-training strategies. \cite{Yefet20a,Rabin21a} defines an approach which learns adversarial examples having semantic preserving transformations, specifically variable renaming and adding dead code to the original program to attack the trained neural model to make incorrect predictions for the tasks of bug finding and predicting method names. \cite{Bielik20a} checks the robustness of neural models trained for predicting program properties by creating adversarial examples. As opposed to these approaches, we experiment with semantic preserving transformations for the code summarization task to evaluate the model dependency on meaningful function and variable names for summary generation. \cite{Srikant21a} identify the perturbations in terms of replacing the existing code tokens and the sites in the program to apply the perturbations to break sequence-to-sequence code summarization models. However, the summaries taken into consideration are of only few (~1-3) words. On the other hand, we aim at exploring the effect of semantic preserving along with semantic disrupting transformations on the predicted summaries, which are more elaborate in terms of token lengths. Based on the assumption that the summaries require the models to understand the underlying logic of the code and can not just rely on exploiting the inductive bias created by the meaningful function and variable names, we hypothesize that the semantic preserving transformations should have no major effect on the generated summaries, whereas the semantic disrupting transformations should negatively affect the summaries.

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 II demonstrates the transformations.

\textbf{SPT} are the set of Semantic Preserving Transformations, which include (i) \textit{CC} removing the Code Comments from 17\% of the codes in CodeSearchNet (ii) \textit{FN} replacing meaningful user-defined Function Names with more generic (but unique) function names, and (iii) \textit{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. \textit{FN} and \textit{VN} are applicable to all codes in all the datasets.

\textbf{SDT}s 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. Changing the operators would bring in change in the control flow dependencies...
### Table 4: 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>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>Model</td>
<td>SIT</td>
<td>BLEU</td>
<td>Drop</td>
<td>BLEU</td>
</tr>
<tr>
<td>Original</td>
<td>34.11</td>
<td>-</td>
<td>25.53</td>
<td>-</td>
</tr>
<tr>
<td>EXP-Te-DL</td>
<td>23.61</td>
<td>10.5</td>
<td>22.99</td>
<td>2.54</td>
</tr>
<tr>
<td>EXP-Te-VN</td>
<td>18.39</td>
<td>5.22</td>
<td>21.20</td>
<td>1.79</td>
</tr>
<tr>
<td>EXP-Te-CC</td>
<td>23.61</td>
<td>0.00</td>
<td>22.40</td>
<td>0.19</td>
</tr>
<tr>
<td>EXP-Tr-SPT</td>
<td>18.25</td>
<td>5.36</td>
<td>20.08</td>
<td>2.31</td>
</tr>
<tr>
<td>EXP-TrTe-SPT</td>
<td>20.78</td>
<td>2.83</td>
<td>18.63</td>
<td>4.36</td>
</tr>
<tr>
<td>EXP-Tr-SDT</td>
<td>23.57</td>
<td>0.04</td>
<td>22.98</td>
<td>0.01</td>
</tr>
<tr>
<td>EXP-TrTe-SDT</td>
<td>15.33</td>
<td>8.28</td>
<td>23.20</td>
<td>-0.21</td>
</tr>
</tbody>
</table>

of the code and thus would lead to structural code changes. ~78%, 68%, 40% and 43% of codes in CodeSearchNet-Java and Python, TLCodeSum and Python datasets, respectively, 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 GT summaries, which are retained for both the transformations.

## 4 Experimental Setup

This section explains the performed experiments and the corresponding hypotheses in detail.

**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 exactly matches 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. We hypothesize that the 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, we hypothesize that the BLEU scores should improve.

**EXP-TrTe-SPT** Models trained with the SPT transformed Train Set are tested on the SPT transformed Test Set. Along similar lines of EXP-Tr-SPT, we expect improvements in the BLEU scores over unmodified trainset-testset results indicating 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, based on the assumption that the model uses code semantics for summarization, we hypothesize that the generated summaries are different from the original GT summaries, leading to a drop in the BLEU scores.

**EXP-Tr-SDT** Models trained on the SDT transformed Train Set retaining the original GT summaries are tested on the original Test Set. As SDT changes the code semantics, mapping such programs to the original summaries lead to noisy training. Thus, we hypothesize that the generated summaries on the original Test Set would be noisy leading to some drop in the BLEU scores.

**EXP-TrTe-SDT** Models trained with the SDT transformed Train Set and Tested with SDT transformed Test Set, both retaining the original GT summaries. Both Train and Test sets being noisy, this set-up does not provide any insights about change in the model performance after probing. Hence we do not perform this experiment.
To programmatically transform the codes, we use javalang, and ast packages. We detect the function and variable names by AST construction. The logical and arithmetic operators are detected by using regex. SIT is trained on TL-CodeSum and Python dataset and PLBART on CodeSearchNet. SIT and PLBART take ∼34 and 8 hours to train. Experiments on CodeSearchNet are performed with only PLBART as the program analysis information required for the SIT model is not available for this dataset.

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 GT 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 GT as compared to EXP-Te-SPT demonstrating the positive effect of an SPT transformed train set. Subtle change in the operators of ‘for loop’ in the Java code with EXP-Te-SDT as demonstrated in Table 1, completely disrupts the code logic. However, no change in the corresponding summaries showcase no influence of SDT.

Table 4 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. Comparable BLEU scores for SIT and PLBART models questions the benefit of infusing PAI 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. With EXP-Te-DL, the overall BLEU scores are in the range of 16-24, questioning their utility for real-life applications.

There is a further drop in BLEU (7.09) with EXP-Te-SPT showcasing the huge 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 pre-processing 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. With EXP-Tr-SDT the model trained with noisy pairs of data (SDT transformed code with original summary) showcase random behaviour with drop in BLEU for most of the cases, whereas for others we can observe a very slight increase in BLEU.

We perform qualitative analysis for the sampled test cases fed to the SIT model trained on Python dataset Table 5. High drop in BLEU for EXP-Te-SPT and EXP-TrTe-SPT is due to (i) GT summary having textual correlations with the function and variable names in the code. Training with original data leads to reliance on such correlations which are not available to exploit due to SPT transformed test data (Example A) (ii) Model trained with SPT transformed training data removes the bias of textual correlations. Such model, during inference, tends to copy the summary of a syntactically similar transformed code from the training set (Example B). No drop in BLEU for EXP-Te-SDT is due to GT summary having high reliance on textual cues in the code leading to model not learning to pay attention to the operators present in the code (Example C).

1https://github.com/c2nes/javalang
2https://docs.python.org/3/library/ast.html#
3https://github.com/python/cpython/blob/3.10/Lib/re.py
4https://github.com/gingasan/sit3
5https://github.com/wasiahmad/PLBART
6https://github.com/tree-sitter/tree-sitter
7https://cloud.google.com/translate/automl/docs/evaluate#bleu
We understand that BLEU score, though widely used as a metric for code summarization (Shia et al., 2022), is not an appropriate metric to measure the quality of the code summaries (Allamanis et al., 2018). BLEU only assigns credit to exact n-gram overlaps and does not take the sentence structure and the semantics into account. More importantly, BLEU is uncorrelated with the programmers comprehension of the source code (Stapleton et al., 2020) and the program correctness (Austin et al., 2021; Chen et al., 2021; Hendrycks et al., 2021). BERTscore (Zhang et al., 2019) or BARTscore (Yuan et al., 2021) metrics take sentence fluency and semantics into consideration. However, these metrics have not been evaluated in the context of source code summarization. The above analysis can be better supported by human evaluation (Shia et al., 2022; Zhu & Pan, 2019). However, considering the size of the test data (Table 2) and the experiments performed (Table 4), human evaluation is infeasible. Thus, the choice of BLEU score as the evaluation metric is the limitation of this paper. This points to the need of an appropriate evaluation metric for the task. We also plan to perform human evaluation on sampled data as the future work.

## 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 and well-curated datasets to facilitate code understanding. 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 illustrated transformations. We are working on extending it to generalize our observations.
REFERENCES


Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. 2018b.


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


