Understanding Code Semantics: An Evaluation of Transformer Models in Summarization

Debanjan Mondal[*]   **Abhilasha Lodha**[*]   **Ankita Sahoo**[*]   **Beena Kumari**[*]
University of Massachusetts Amherst
{debanjanmond,alodha,asahoo,beenakumari}@umass.edu

Abstract

This paper delves into the intricacies of code summarization using advanced transformer-based language models. Through empirical studies, we evaluate the efficacy of code summarization by altering function and variable names to explore whether models truly understand code semantics or merely rely on textual cues. We have also introduced adversaries like dead code and commented code across three programming languages (Python, Javascript, and Java) to further scrutinize the model's understanding. Ultimately, our research aims to offer valuable insights into the inner workings of transformer-based LMs, enhancing their ability to understand code and contributing to more efficient software development practices and maintenance workflows.

1 Introduction

Code summarization is a task that involves generating coherent and semantically relevant summaries that effectively describe the intended function of the software. In the dynamic realm of software development and maintenance, an adept grasp of program functionalities is of paramount importance. In this context, the integration of natural language summaries derived from source code emerges as a potent instrument, streamlining developers' efforts and augmenting program comprehension.

While current state-of-the-art code summarization models are developed and evaluated on clean and curated datasets, the real-world coding environment is far from standardized. Developers often deviate from standard coding practices, leading to inconsistent naming conventions. Additionally, actual codebases often feature commented sections, serving as legacy code or reserved for future use cases. Our research aims to simulate these real-world scenarios and assess whether models truly comprehend the inherent code semantics, rather than merely relying on textual cues.

The prevailing approaches to code summarization typically employ an encoder-decoder framework, encompassing the conversion of code into a hidden space and its subsequent transformation into natural language. For instance, CodeT5 (Wang et al., 2021), a unified pretrained encoder-decoder Transformer model, leverages the semantics encoded in identifiers. In this research, we investgate the effectiveness of these models by tweaking the function and variable names in the existing code summarization datasets. Furthermore, we introduce additional challenges, such as commented code and dead code, to elevate the complexity of data samples and scrutinize the models' summarization processes. Dead code refers to unreachable code segments, devoid of functional importance, which language interpreters (e.g., Python and Javascript) ignore. We seek to evaluate whether models effectively disregard such code segments. All our experiments are reproducible and we will release our code and data upon publication.

The driving motivation behind this research lies in enhancing code comprehension and reducing the efforts entailed in software development and maintenance. By unraveling how Language Models comprehend code, we aim to contribute insights that pave the way for more effective software development practices. Our study, through experimentation and analysis, strives to provide valuable directions for improving the capabilities of Language Models in understanding and summarizing code, despite the challenges posed by real-world coding scenarios.[*]

2 Related Work

Automated code summarization is a useful tool for software developers and has been an active re-

---

[*]Equal contribution.

[*]Our code is publicly available at: Github

serach field for quite some time. Recently large language models have shown significant improvements in natural language tasks. Inspired by this, several pretrained language models have been developed for the programming language tasks. The encoder-decoder models have been found to be more successful in Programming Language (PL) tasks, whereas fully decoder models perform significantly better in Natural Language (NL) domain.

Models like CodeBERT (Feng et al., 2020), PLBART (Ahmad et al., 2021), GraphCodeBERT (Guo et al., 2021), CodeT5 (Wang et al., 2021), CoTexT (Phan et al., 2021) have shown impressive performances in the CodeXGLUE (Lu et al., 2021) benchmark. Unlike natural language, it's necessary to capture the rich code semantics in the programming language. Most Programming Language Models (PLMs) in this domain are pretrained on a large corpus of NL-PL pair in several programming languages with a masked token prediction objective. To capture the code semantics, various models have used different approaches. For example, CodeT5 uses an additional masked identifier prediction and GraphCodeBERT incorporates the data flow extracted from the code. These PLMs have shown impressive results in downstream tasks like code summarization. Ahmed and Devanbu (2022a) explored the code summarization in project-specific domain. Sun et al. (2022) used an extractive and abstractive framework from source code summarization. Ahmed and Devanbu (2022b) showed that multilingual training can amply performance for low resource languages in different downstream tasks including code summarization. (Chen et al., 2022) provided further insights for low resource languages like Ruby.

As indicated by Guo et al. (2021) in GraphCodeBERT, indicators play a key role in code summarization. However, it's more desirable that our model relies more on code semantics and syntax rather than method names and identifiers. Developers follow their own naming conventions which can affect the model performance. In a closely related work, Sontakke et al. (2022) have shown that Semantic Preserving Transformations like removing code comments, replacing function names and local variable names to generic names significantly affects the BLEU score of the models like PLBART. We want to extend this exploration to other models like CodeT5, CodeBERT etc. We also aim to increase the scope of semantic preserving transfor-

mations by including dead code and commented code to check the model's understanding of the code.

3 Dataset

There are several different code summarization datasets available. But we prefered CodeXGLUE (Lu et al., 2021) over others for these reasons -

- CodeXGLUE has been meticulously deduplicated, as demonstrated in (Shi et al., 2022). This ensures that any duplication within the dataset does not artificially inflate performance metrics.

- CodeXGLUE offers a wide range of six languages. This allowed us to conduct experiments and compare results across different programming languages.

We can categorize the six languages available in CodeXGLUE into three groups based on the size of their combined datasets (including train, validation, and test sets). Please note that the following information pertains to the combined dataset size:

- In the High Resource category, both Python and PHP have approximately 300,000 code-summary pairs each.

- In the Mid Resource category, Java and Go consist of around 180,000 code-summary pairs.

- In the Low Resource category, Javascript comprises 65,000 pairs, while Ruby only has 27,000 code-summary pairs.

To analyze the impact of data transformation across resource categories, we selected one language from each category. Therefore, for our experiments, we utilized the languages Python, Java, and Javascript. The detailed statistics about train, validation and test splits is presented in Table 1.

3.1 Data Transformation

We will focus on code transformations that will preserve the code functionality. In the programming paradigm, this is known as **obfuscation**. In this study, we focused on 3 kinds of transformations. These are visually explained in Figure 1:

- **Renaming Identifiers:** Although the software development industry emphasizes the importance of meaningful and descriptive names

for functions and variables, developers often use random function and variable names. To replicate such scenario, we replaced function and variable names with generic but unique names. However while doing so, we had to keep in mind that the control flow of the overall program should not be affected. We leveraged the Abstract Syntax Tree (AST) of the source code to identify and edit identifiers. Implementation details and interesting corner cases vary across programming languages and will be discussed subsequently.

- **Commented Code:** It is very common in software engineering to encounter commented codes inside a function. These may be legacy codes which are not used anymore, or code snippets that might be used in future. To simulate such situation, we added commented codes. For each source code, we randomly sampled a function within the same data split, created commented version of it and added it after a function definition. Finding a suitable place to add comments is tricky and sometimes it can potentially change the program functionality. For our experiments, we add the comments starting from the next line after function definition.

- **Dead Code:** Adding code after return statements is another transformation that we explored in our experiments. Python and Javascript interpreters ignore anything added after return statements, and we wanted to check if the models have developed the ability to do so. Since Java compiler throws error if we add anything after return, we excluded Java from this study.

Code transformation implementation details about specific programming languages have been presented in A.2.

4 Models and Evaluation

4.1 Models

In this study we ran our experiments on 2 models:

- **CodeT5:** Upon examining the leaderboard of CodeXGLUE[2], and comparing metrics from various research papers, we discovered that

Languages	Transformation	Train Data	Validation Data	Test Data
Python	Original	251820	13914	14918
	Renamed Identifiers	251820	13914	14918
	Commented Code	251820	13914	14918
	Dead Code	251820	13914	14918
Javascript	Original	58025	3885	3291
	Renamed Identifiers	38254	2730	2157
	Commented Code	38254	2730	2157
	Dead Code	21897	1548	1213
Java	Original	164923	5183	10955
	Renamed Identifiers	164888	5182	10953
	Commented Code	164923	5183	10955

Table 1: Dataset Size Information for different splits.

CodeT5 (Wang et al., 2021) achieves state-of-the-art results for the code summarization task on this benchmark. Due to limitations in the size of our available GPUs, we opted to utilize the CodeT5 small[3] (60M parameters) and base[4] (223M parameters) models, while excluding the CodeT5 Large model (770M parameters) from our analysis. It is worth noting that CodeT5 incorporates an identifier aware denoising objective during its pretraining, making it more inclined to utilize textual cues from identifiers. We wanted to evaluate its robustness in our experiments.

- **CodeBERT:** For comparison across different architectures, we chose the CodeBERT (Feng et al., 2020) (173M parameters) model. Unlike CodeT5, CodeBERT is an encoder only model which is pretrained on the CodeSearchNet (Husain et al., 2020) dataset. For sequence-to-sequence generation problems like code summarization, the authors provide an Encoder-Decoder framework where they initialize the encoder with the pretrained CodeBERT, but they randomly initialize the decoder with a transformer model. Note that the decoder weights are not trained during the pretraining phase. For our experiments we use the CodeBERT Base[5] model.

The finetuning code utilized for our project was obtained from the public GitHub repository of CodeT5.[6] The CodeT5 authors also included the finetuning code for CodeBERT. We reused the finetuned checkpoint of CodeT5 base that

---

[2] https://microsoft.github.io/CodeXGLUE/

[3] https://huggingface.co/Salesforce/codet5-small
[4] https://huggingface.co/Salesforce/codet5-base
[5] https://huggingface.co/microsoft/codebert-base
[6] https://github.com/salesforce/CodeT5

Transformation Type	Original Code	Modified Code
| Renaming Identifiers | ```python
def standard(cls, element):
    # We remove all special
    # characters and return the
    # formatted string.

    return (Regex(element,
        cls.regex_replace,
        replace_with="@funilrys")
        .replace()
        .replace("@funilrys", ""))
``` | ```python
def func1(arg1, arg2):
    return Regex(arg2,
        arg1.regex_replace,
        replace_with='@funilrys')
        .replace()
        .replace('@funilrys', '')
``` |
| Dead code | ```python
def collection_to_df (collection):
    return
    pd.concat([record.series
        for record in collection],
        axis=1).T
``` | ```python
def collection_to_df (collection):
    return
    pd.concat([record.series
        for record in collection],
        axis=1).T

    style = tag.get('style')
        if style and
        'text-decoration:underline'
        in style:
            tag.wrap(self.soup
            .new_tag('u'))
``` |
| Commented Code | ```python
def str_lower(x):
    sl =
    _to_string_sequence(x).lower()
    return
    column.ColumnStringArrow(sl.b
    ytes, sl.indices, sl.length,
    sl.offset, string_sequence=sl)
``` | ```python
def str_lower(x):
    # def fromRaw(cls, skype=None,
    # raw={}):
    #     return cls(skype, raw,
    #     **cls.rawToFields(raw))

    sl =
    _to_string_sequence(x).lower()
    return
    column.ColumnStringArrow(sl.b
    ytes, sl.indices, sl.length,
    sl.offset, string_sequence=sl)
``` |

Figure 1: Examples of different kind of transformations discussed in Section 3.1.

was available. As for CodeT5 small and Code-BERT, we performed the finetuning process on clean CodeXGLUE data, using the hyperparameters specified in the repository. The finetuned checkpoints on clean data for every model and every language serve as our baseline. Note that, the BLEU scores (Papineni et al., 2002) of our finetuned checkpoints on the clean CodeXGLUE test data has slight discrepancies with the BLEU scores reported in the original papers. However, the difference was not more than one decimal point. This may be attributed to the difference between GPU architectures. We report the scores mentioned in the original papers as the train clean - test clean scores.

5 Experiments

For all our experiments, we use the NVIDIA Tesla M40 GPUs. While finetuning for all types of data (Clean, Corrupted and Combined), we used the clean data finetuning hyperparameters that were available in the CodeT5 repository. However, for CodeT5 Base model, we had to reduce the batch size from 48 to 16 to fit in our GPU. All other hyperparameters remain the same. With this setup,

we perform the following experiments.

- **Identifier Corruption:** We conducted individual finetuning of the models using clean, corrupted, and combined train data (clean + corrupted). Subsequently, we evaluated these three types of models on both clean and corrupted test data. We used CodeBERT, CodeT5 Small and CodeT5 Base models for this experiment.

- **Commented Code Corruption:** For this corruption type, we performed separate finetuning of the model using clean data and commented code corrupted data. Finally, we evaluated these two types of models on both clean data and commented code corrupted data. We only evaluated the CodeT5 Small and CodeT5 base models in this particular setup.

- **Dead Code Corruption:** Similar to the previous corruption type, we carried out separate finetuning of the model using clean data and dead code corrupted data. Subsequently, we evaluated these two types of models on both clean data and dead code corrupted data. Once

again, we only evaluated the CodeT5 Small and CodeT5 base models in this setup.

5.1 Evaluation

Evaluation metrics are crucial for assessing the effectiveness of a code summarization model. In our study, we utilized both automatic and human evaluations.

1. **BLEU** (Papineni et al., 2002): It is a precision-based metric that measures the overlap between the words (and/or n-grams) in the machine-generated summaries and the human reference summaries. We calculated smoothed BLEU-4 (considering upto 4-grams) scores for each of the generated summaries and then averaged across all the summaries. We used the same evaluation script as CodeT5 which in turn reused the original evaluation script provided in the CodeXGLUE (Lu et al., 2021) benchmark. The BLEU scores obtained from different experiments for Python, Java, and JavaScript are presented in Table 2, 3, and 4.

2. **Human Evaluation:** Additionally, we performed a manual evaluation by annotating 200 random samples for both Python and JavaScript. Further details of the human evaluation process are discussed in the Section 7.

6 Discussion

Our experiments try to answer the following essential questions on models' understandability.

6.1 Research Question 1: *Does a model trained on clean data perform well on the identifier corrupted data?*

For all languages and models, the performance of a model trained on clean data tends to diminish when faced with corrupted test data, as compared to its performance on clean test data. The drop in performance for all models and languages is at least 4 points in terms of BLEU score. This phenomenon may be attributed to the model's reliance on textual hints found within function and variable names during training, rather than grasping the true essence of the code's functionality and achieving generalization. The comparison is visually shown in Figure 2.
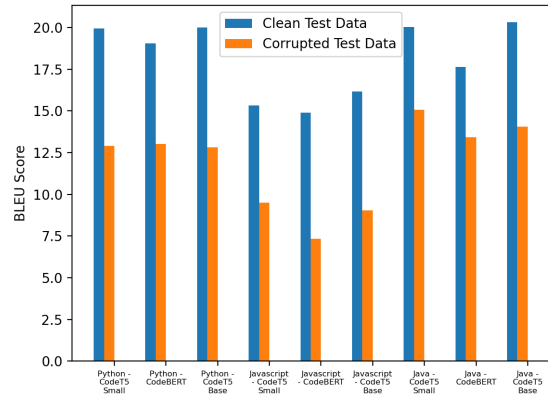

Figure 2: Comparing performance on clean and corrupted test data for models trained on clean data.

6.2 Research Question 2: *Does the model trained on identifier corrupted data perform well on clean data?*

A surprising observation arises when examining the performance of the model trained only on corrupted data. It demonstrates a commendable level of proficiency not only on corrupted test data (which is expected) but also on clean test data. This is visually explained in Figure 3. For all languages and models, the performance on clean test data between the model trained on clean data and corrupted data is less than 1 point in terms of BLEU score, except for CodeBERT in javascript. We hypothesize that when we train on the corrupted data, the model is forced to understand the code functionality in a generalized manner, thereby enabling it to perform well even in the clean dataset.
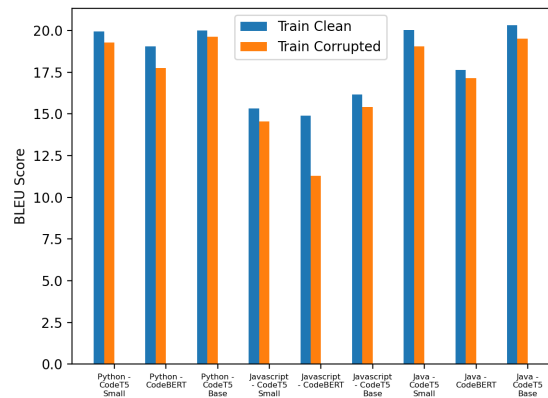

Figure 3: Comparing performance on clean test data and for models trained on clean data and corrupted data.

Train Data	Test Data	Python			Javascript			Java		
		CodeT5 Small	CodeBERT	CodeT5 Base	CodeT5 Small	CodeBERT	CodeT5 Base	CodeT5 Small	CodeBERT	CodeT5 Base
Clean	Clean	19.96	19.06	20.01	15.32	14.90	16.16	20.02	17.65	20.31
	Corrupted	12.92	13.03	12.81	9.51	7.34	9.04	15.06	13.42	14.05
Corrupted	Clean	19.28	17.75	19.64	14.55	11.30	15.41	19.05	17.16	19.50
	Corrupted	16.21	15.52	16.50	12.68	10.76	13.13	17.27	16.20	17.36
Combined	Clean	19.73	18.93	20.05	15.27	13.01	15.83	19.82	18.01	19.70
	Corrupted	16.17	15.76	16.51	12.46	11.29	12.86	17.14	16.19	17.44

Table 2: Smooth BLEU-4 scores for different train-test combinations for clean and identifier corrupted data. When models are trained using clean data, their performance deteriorates when tested on identifier corrupted data. However, when models are trained on a combination of both clean and corrupted data, they demonstrate satisfactory performance on both types of test data - clean and corrupted.

Train Data	Test Data	Python		Javascript	
		CodeT5 Small	CodeT5 Base	CodeT5 Small	CodeT5 Base
Clean	Clean	19.96	20.01	15.32	16.16
	Dead Code	18.55	19.83	15.20	15.52
Dead Code	Clean	19.74	18.66	14.69	15.32
	Dead Code	18.92	19.19	15.62	16.70

Table 3: Smooth BLEU-4 scores for different language and train-test combinations for dead code corruption.

6.3 Research Question 3: *How is the performance of the model trained on the combined data?*

Our combined dataset contained both clean data and identifier corrupted data. The model trained on a combined dataset exhibits impressive performance not only on clean data but also on identifier corrupted data. Its performance on the clean test data is very similar to and sometimes even surpasses the performance of the model only trained on the clean data. Similar observations are seen for the corrupted test data. Notably, this pattern is consistent across all the models and languages. This shows that if we curate our dataset correctly, the model can generalize across clean and corrupted datasets. The BLEU score performance comparisons are visually explained in Figures 4 and 5.

6.4 Research Question 4: *What is the effect of commented code perturbations on the model's capabilities?*

We observe that the model trained on a dataset consisting of commented code showcases comparable and impressive performance on both clean code data and commented code data. However, a model trained exclusively on clean data displays satisfactory performance on the clean test set, albeit experiencing a notable decline in performance when evaluated on the commented code test set (presented in Figure 6). A potential explanation is that the absence of comments in the clean code training data
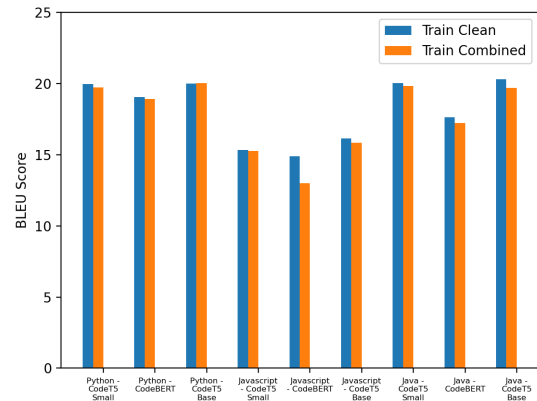

Figure 4: Comparing performance on clean test data for models trained on clean data and combined data.
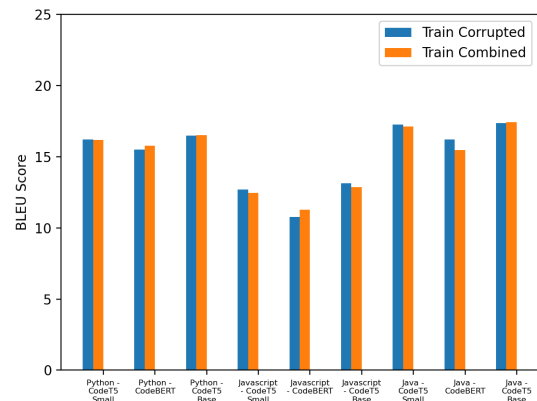

Figure 5: Comparing performance on corrupted test data for models trained on corrupted data and combined data.

prevents the model from learning the syntax associated with comments. However, when the model is trained on code that includes comments, it captures the code syntax information despite the comments not directly influencing the code's functionality.

Train Data	Test Data	Python		Javascript		Java	
		CodeT5 Small	CodeT5 Base	CodeT5 Small	CodeT5 Base	CodeT5 Small	CodeT5 Base
Clean	Clean	19.96	20.01	15.32	16.16	20.02	20.31
	Commented	16.15	16.26	14.61	14.21	15.06	18.83
Commented	Clean	19.07	17.90	15.00	15.07	19.77	20.23
	Commented	18.32	18.75	15.90	15.73	19.57	20.17

Table 4: Smooth BLEU-4 scores for different language and train-test combinations for commented code corruption.
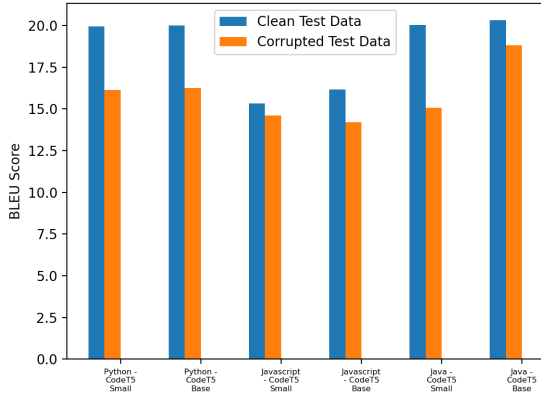

Figure 6: Comparing performance on clean and commented test data for models trained on clean data.

6.5 Research Question 5: *What is the effect of dead code perturbations on the model's capabilities?*

Upon analysis, we determine that a model trained on a dataset that includes non-functional dead code demonstrates impressive and similar performance when applied to both clean code and the aforementioned dead code. We make the same observation for a model trained on clean data and evaluated on both clean and non-functional code test datasets. We thus conclude that the addition of dead code doesn't have any significant impact on the generated summaries.

7 Human Evaluation

In order to gain deeper insights into specific errors, a manual evaluation was conducted on a randomly selected subset of 200 examples from the identifier corrupted dataset. The evaluation compared the performance of two CodeT5 Base models: the model trained on the combined dataset (including identifier corrupted codes) and the baseline model trained on clean data. The evaluation involved analyzing code, gold truth, baseline model summaries, and combined data trained model summaries. We

prepare one such set for both Python and Javascript (Refer Table 5). To avoid human bias, the annotators were given the summaries in a random order without any access to the information about which model generated them.

Language	Clean Data Model	Combined Data Model	Ties	Total
Python	21	143	36	200
Javascript	15	148	37	200

Table 5: Statistics of the aggregated manual evaluation data determining which model's summary was better.

The evaluation process included two individuals independently annotating the same set of data and marking the annotations as *Prediction 1*, *Prediction 2*, or *Tie* (both models). The chosen option implies that the selected annotation is more closely aligned with the gold truth and code. The inter-annotator agreement can be observed in the Table 6.

In cases where there was a disagreement between the annotators, *Ties* were resolved using the following strategies:

- If one annotation was marked as a *Tie* and the other was marked as *Prediction 1* or *Prediction 2*, we considered the *Prediction 1* or *Prediction 2* annotation as the final annotation in the aggregated dataset.

- If one annotation was marked as *Prediction 1* and the other as *Prediction 2*, the two annotators engaged in a discussion to reach a consensus for the final annotation for the summaries.

The observations revealed that the summaries generated by the model trained on the combined data were more relevant to the code and closer to the desired outcome (gold truth) compared to the baseline model. There were several notable issues with the summaries generated by the baseline model, which are discussed in the Figure 7.

The computed BLEU scores for the two models, where one was trained on clean data and the other on combined data, are as follows: For Python, the

bleu scores on the identifier corrupted dataset are **12.92** and **16.17**, while for Javascript, the scores are **9.04** and **12.86**, respectively. These scores align with the manual evaluation results, indicating that the model trained on combined data outperforms the model trained solely on clean data in code summarization.

8   Conclusion

By studying advanced code summarization models, we discover how making changes that preserve the meaning of the code affects the quality of the summaries they generate. Additionally, we provide evidence that if we train the large language models like CodeT5 properly, making changes that disrupt the meaning of function and variable names have little impact on the resulting summaries. Our observations remain consistent across three distinct programming languages: Java, Python, and JavaScript. These findings raise important concerns about how well these models truly understand code, highlighting the need for better training methods and carefully curated datasets that improve their understanding. We propose using different types of code transformations, such as introducing re-named identifiers, adding comments, or dead code, as ways to enhance the training of these models. Furthermore, there is an exciting opportunity to apply these findings to different programming languages, so that we can learn more about their general applicability.

Limitations

While this study presents valuable insights into code summarization using CodeBERT and CodeT5, certain limitations merit consideration.

Firstly, the experimentation focused exclusively on CodeBERT and CodeT5 due to practical GPU restrictions. While Large Language Model (LLMs) based approaches hold immense potential, their exclusion from the evaluation due to GPU limitations might restrict the generalizability of findings.

Secondly, the reliance on BLEU evaluation metrics, although widely used, introduces its own limitations(Roy et al., 2021). BLEU captures the word-level overlap between generated and reference summaries, but it may not holistically reflect the quality of the summary in all cases. The intricate semantics and contextual intricacies present in code may not be fully captured by BLEU scores alone.

Moreover, while human evaluation was conducted on a subset of 200 samples, the comprehensiveness of this assessment could have been further extended. A more expansive human evaluation, covering a broader array of code samples, could provide a richer understanding of the models' actual performance.

In future studies, overcoming these limitations could involve wider experimentation across a spectrum of Language Models, a more robust human evaluation, and exploring alternative evaluation metrics that better align with the complex nature of code summarization.

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.

Toufique Ahmed and Premkumar Devanbu. 2022a. Learning code summarization from a small and local dataset.

Toufique Ahmed and Premkumar Devanbu. 2022b. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1443–1455, New York, NY, USA. Association for Computing Machinery.

Fuxiang Chen, Fatemeh Fard, David Lo, and Timofey Bryksin. 2022. On the transferability of pre-trained language models for low-resource programming languages.

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.

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 Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin 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. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

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 Anibal, Alec Peltekian, and Yanfang Ye. 2021. Cotext: Multi-task learning with code-text transformer.

Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks.

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*. ACM.

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*.

Weisong Sun, Chunrong Fang, Yuchen Chen, Quanjun Zhang, Guanhong Tao, Tingxu Han, Yifei Ge, Yudu You, and Bin Luo. 2022. An extractive-and-abstractive framework for source code summarization.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pretrained 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.

A Appendix

A.1 Additional information on human annotation

Language	Raw Agreement	Cohen's Kappa
Python	74.5%	0.62
Javascript	82.0%	0.73

Table 6: Inter-annotator Agreement Statistics.

A.2 Data Transformation Procedure

A.2.1 Python

- **Renaming Identifiers:** We use Python's *ast* package to parse the code and create AST. Then we transform the ast using *NodeTransformer*[7] class. Finally the modified ast is unparsed and saved to the output file.

- **Commented Code:** Commented code is added after the function definition by randomly selecting a function code snippet from the same data split and adding comment symbols (#) before each line of the selected code snippet.

- **Dead Code:** The code in this section adds extra code snippets after return statements in Python source code. The extra code snippet is taken from the function body of a randomly selected function in the same data split. It uses the *libcst*[8] library to identify the location of the return statement.

A.2.2 Javascript

- **Renaming Identifiers:** To achieve this, the *esprima*[9] library was employed to obtain the AST structure. Some codes are excluded from our dataset because the library fails to obtain the AST for those specific codes. The AST was traversed using *Depth-First Search (DFS)* to extract node details of Identifiers related to variables and functions. Subsequently, the *estraverse*[10] library was utilized to traverse

---

[7]https://docs.python.org/3/library/ast.html#ast.NodeTransformer
[8]https://libcst.readthedocs.io/en/latest/
[9]https://www.npmjs.com/package/esprima
[10]https://www.npmjs.com/package/@types/estraverse

73

Corrupted Code	Type of issues in the summary	Gold truth	Summary generated by model trained on clean data	Summary generated by the model trained on combined data

function func0(variable0, variable1, variable2) {
    const variable3 = copyArray(this.subscribers(variable0, variable1));
    let variable4;
    const variable5 = variable3.length;
    if (variable5 === 0) {
        return false;
    }
    variable2 = preprocessorsPublishData(variable2);
    while (!!(variable4 = variable3.shift())) {
        _executeHandler(variable4, variable2, variable0, variable1);
    }
    return true;}
	Repetitive summary generated by **model trained on clean data**	Publish the event in one channel .	This function is the function that is called when the function is called .	Publishes data to all subscribers .

def func1(arg1, arg2):
    var1 = asyncio.get_event_loop()
    var2 = AsyncSatel(arg1, arg2, var1, [1, 2, 3, 4, 5, 6, 7, 8, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 25, 26, 27, 28, 29, 30], [8, 9, 10])
    loop.run_until_complete(stl.connect())
    loop.create_task(stl.arm('3333', 1))
    loop.create_task(stl.disarm('3333'))
    loop.create_task(stl.keep_alive())
    loop.create_task(stl.monitor_status())
    loop.run_forever()
    loop.close()
	Irrelevant / Generic summary of the code generated by **model trained on clean data**	Basic demo of the monitoring capabilities .	Same as func2 .	Example demo .

function func0(variable0, variable1, variable2) {
    if (Array.isArray(variable1)) {
        variable1.forEach(v => store[variable0].langs[v] = variable2);
        return;
    }
    store[variable0].langs[variable1] = variable2;
}
	Usage of variable and function names in the summary of **model trained on clean data**	Register Loader of different Language	set variable0 to variable1 with variable2	Adds a language to the store .

function func0(variable0, variable1) {
    var variable2 = lex();
    if (variable2.type !== (variable1 ? Token.Identifier : Token.Keyword) || variable2.value !== variable0) {
        throwUnexpected(variable2);
    }
}
	Irrelevant / incoherent summary generated in case of **Tie**	Expect the next token to match the specified keyword . If not an exception will be thrown .	ECMA - 262 12 . 2	ECMA - 262 13 . 3 . 3

def func1(arg1, arg2, arg3, *arg4, **arg5):
    if (arg2 is None):
        (arg2, var1) = self._get_client_creds_from_request(arg3)
    log.debug('Authenticate client %r.', arg2)
    var2 = (request.client or self._clientgetter(arg2))
    if (not var2):
        log.debug('Authenticate failed, client not found.')
        return False
    request.client = var2
    return True
	Relevant summary generated in case of **Tie**	Authenticate a non - confidential client .	Authenticate a client !	Authenticate a client .

Figure 7: Examples of different summaries generated by the models in manual evaluation.

the AST and rename each identified node accordingly. Finally, the modified code was generated using *escodegen*[11] and saved as the output.

- **Commented Code:** The commented code is inserted following the function signature. This is performed after getting the AST and using the AST to identify the end of function signature. The code used for commenting is chosen randomly from a collection of code snippets found in another JSON object, with comment symbols (//) added to each line.

---

[11] https://www.npmjs.com/package/escodegen

74

- **Dead Code:** Similar to commented code example, a function body code is randomly selected from a collection of code snippets in the same data split. This code is then appended to the original code after the return statement.

A.2.3 Java

- **Renaming Identifiers:** To modify the function names, and variable names, AST was generated for each code input samples using *JavaParser*[12] package. The AST was traversed to extract the function and variable name nodes, which was then modified to generalized names. It is taken care of to replace the occurrence of same variable and function names with the modified name throughout the code sample using a hash map.

- **Commented Code:** For adding commented code, we searched for the first opening curly braces "{" and the commented code was inserted within /* ... */ and added after the aforementioned curly braces. The commented code snippets were randomly sampled from the same data split of Java code samples.

- **Dead Code:** Addition of codes after return statement in Java throws compile error, therefore dead code was not added to Java code samples.

---

[12]https://javaparser.org/