Code Summarization: Do Transformers Really Understand Code?

Anonymous ACL submission

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-

Java code	Original Function
	private void append(StringBuilder buffer,double[] data,String prefix, String separator,String suffix){
	buffer.append(prefix);
	for (int i=0; i < data.length; ++i) {
	if (i > 0) {
	buffer.append(separator);}
	buffer.append(data[i]);}
	buffer.append(suffix);}
			Function after SPT and SDT
			private void func(StringBuilder var1,double[] var2,String var3 ,String var4,String var5){
			var1.append(var3);
			for(int i=0;i > var2.length; −−i){
			if (i < 0) {
			var1.append(var4);}
			var1.append(var2[i]);}
			var1.append(var5);}

Summ.	Model	SIT	PLBART
	GT	append a text representation of an array to a buffer .	append a text representation of an array to a buffer .
	Original	appends the given string representation of all elements. a concatenates with the given prefix.	appends the given double array to the given buffer.
	EXP-Te-SPT	compute the given string.	func(double[] var1,double[] var2,string var3,string var0)
	EXP-Tr-SPT	append a string listing of format.	appends a double array to the buffer.
	EXP-TrTe-SPT	append a string ref to the specified stringbuffer.	appends a string representation of a double array to the string builder.
	EXP-Te-SDT	appends the given string representation of all elements.	appends the given double array to the given buffer.

Python code	Original Function
	def GetEntityViaMemcache(entity_key):
	entity = memcache.get(entity_key)
	if (entity is not None):
	return entity
	key = ndb.Key(urlsafe=entity_key)
	entity = key.get()
	if (entity is not None):
	memcache.set(entity_key, entity)
	return entity
			Function after SPT and SDT
			def func(var1):
			var2 = memcache.get(var1)
			if (var2 is not None):
			return var2
			var3 = ndb.Key(urlsafe=var1)
			var2 = var3.get()
			if (var2 is not None):
			memcache.set(var1, var2)
			return var2

Summ.	Model	SIT	PLBART
	GT	get entity from memcache if available.	get entity from memcache if available.
	Original	returns a key that can be used for entity.	retrieves an entity from memcache.
	EXP-Te-SPT	returns an instance of c.	return the value of var2.
	EXP-Tr-SPT	cache keys for azure entities.	retrieves an entity from memcache.
	EXP-TrTe-SPT	cache keys in memcache.	returns the value of a memcache key.
	EXP-Te-SDT	returns a key that can be used for entity.	retrieves an entity from memcache.

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).

ferent 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[1] 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

[1]https://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-Tr-SPT**, 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 programmatically 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

PL & Dataset	Python				Java TL-CodeSum				Python CSN		Java CSN		
Model	SIT		PLBART		SIT		PLBART		PLBART		PLBART		Avg
Method	BLEU	Drop	BLEU	Drop	BLEU	Drop	BLEU	Drop	BLEU	Drop	BLEU	Drop	Drop
Original	34.11*	-	25.53	-	45.76*	-	20.61	-	19.30#	-	18.45#	-	-
EXP-Te-DL	23.61	10.5	22.99	2.54	19.34	26.42	16.08	4.53	19.30	0.00	18.45	0.00	10.99
EXP-Te-SPT	15.35	8.26	16.10	6.89	10.46	8.88	11.11	4.97	11.95	7.35	12.25	6.20	7.09
$SPT - FN$	18.26	5.35	16.61	6.38	12.81	6.53	12.64	3.44	15.27	4.03	14.26	4.19	4.99
$SPT - VN$	18.39	5.22	21.20	1.79	14.08	5.26	14.96	1.12	17.15	2.15	16.91	1.54	2.85
$SPT - CC$	23.61	0.00	22.80	0.19	19.34	0.00	16.08	0.00	17.61	1.69	18.22	0.23	0.35
EXP-Tr-SPT	18.25	5.36	20.68	2.31	13.77	5.57	14.68	1.40	18.44	0.86	18.25	0.20	2.62
EXP-TrTe-SPT	20.78	2.83	18.63	4.36	16.76	2.58	13.17	2.91	15.43	3.87	15.40	3.05	3.27
EXP-Te-SDT	23.57	0.04	22.98	0.01	19.29	0.05	16.00	0.08	18.92	0.38	18.24	0.21	0.13

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).

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 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 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.

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.

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

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021a. 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.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021b. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

YunSeok Choi, JinYeong Bak, CheolWon Na, and Jee-Hyong Lee. 2021. Learning sequential and structural information for source code summarization. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 2842–2851.

Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

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

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*.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 184–195.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806. IEEE.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*.

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

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020. Retrieval-augmented generation for code summarization via hybrid gnn. In *International Conference on Learning Representations*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

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.

Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. 2021. Cotext: Multi-task learning with code-text transformer. *arXiv preprint arXiv:2105.08645*.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Weizhen Qi, Yeyun Gong, Yu Yan, Can Xu, Bolun Yao, Bartuer Zhou, Biao Cheng, Daxin Jiang, Jiusheng Chen, Ruofei Zhang, et al. 2021. Prophetnet-x: Large-scale pre-training models for english, chinese, multi-lingual, dialog, and code generation. *arXiv preprint arXiv:2104.08006*.

Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. *arXiv preprint arXiv:2108.12987*.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407.

Wenhan Wang, Kechi Zhang, Ge Li, and Zhi Jin. 2020. Learning to represent programs with heterogeneous graphs. *arXiv preprint arXiv:2012.04188*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

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.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1385–1397. IEEE.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. *arXiv preprint arXiv:2103.11318*.

A Limitations of Code Summarization Datasets

Dataset	Language	Train	Valid	Test
Python	Python	57,203	19,067	19,066
TL-CodeSum	Java	69,708	8,714	8,714
CodeSearchNet	Python	251,820	13,914	10,955
CodeSearchNet	Java	164,923	5,183	14,014

Table 3: Dataset statistics for Python (Wan et al., 2018), TL-CodeSum (Hu et al., 2018b) and CodeSearch-Net(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 appications, 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

Example	Example (a) Java Code with Comments from CodeSearchNet	Example (b) Java Code-Summary pair common in Train & Test set of TLCodeSum

Code	static String normalizePath(String path) {
	StringBuilder sb = new StringBuilder(path.length());
	int queryStart = path.indexOf('?');
	String query = null;
	if (queryStart != -1) {
	query = path.substring(queryStart);
	path = path.substring(0, queryStart);
	}
	// Normalize the path. we need to decode path segments, normalize
	//and rejoin in order to
	// 1. decode and normalize safe percent escaped characters. e.g. %70 ->'p'
	// 2. decode and interpret dangerous character sequences. e.g. /%2E/ ->'/./' ->'/'
	// 3. preserve dangerous encoded characters. e.g. '/%2F/' ->'///' ->'/%2F'
	List<String>segments = new ArrayList<>();
	for (String segment : SLASH_SPLITTER.split(path)) {
	// This decodes all non-special characters from the path segment.
	//so if someone passes
	// /%2E/foo we will normalize it to /./foo and then /foo
	String normalized =
	UrlEscapers.urlPathSegmentEscaper().escape(lenientDecode
	(segment, UTF_8, false));
	if (".".equals(normalized)) {
	// skip
	} else if ("..".equals(normalized)) {
	if (segments.size() >1) {
	segments.remove(segments.size() - 1);}
	} else {
	segments.add(normalized);}
	}
	SLASH_JOINER.appendTo(sb, segments);
	if (query != null) {
	sb.append(query);}
	return sb.toString();}
		private static char[] zzUnpackCMap(String packed){
		char[] map=new char[0x10000];
		int i=0;
		int j=0;
		while (i <112) {
		int count=packed.charAt(i++);
		char value=packed.charAt(i++);
		do map[j++]=value;
		while (–count >0);
		}
		return map;
		}

Summary	Normalizes a path by unescaping all safe, percent encoded characters.	Unpacks the compressed character translation table.
Example	**Example (c) Python Code Summary pair common in train and test set of Python dataset**	**Example (d) C Code-Summary pair from CodeNet**

Code	def query_yes_no(question, default=u'yes'):
	valid = {u'yes': u'yes', u'y': u'yes', u'ye': u'yes', u'no': u'no',
	u'n': u'no'}
	prompt = {None: u'[y/n]', u'yes': u'[Y/n]', u'no': u'[y/N]'}.
	get(default, None)
	if (not prompt):
	raise ValueError((u"invalid default answer: '%s'" % default))
	while 1:
	sys.stdout.write((colorize(question, colors.PROMPT) + prompt))
	choice = raw_input().lower()
	if (default and (not choice)):
	return default
	elif (choice in valid):
	return valid[choice]
	else:
	printFailure(u"Please respond with 'yes' or 'no' (or 'y' or 'n').\n")
		#include <stdio.h>
		int main(void)
		{ int n;
		int ii;
		int i;
		float k1, k2;
		float x[4], y[4];
		scanf("%d", &n);
		for (ii = 0; ii <n; ii++){
		for (i = 0; i <4; i++){
		scanf("%f %f", &x[i], &y[i]);}
		k1 = (y[1] - y[0]) / (x[1] - x[0]);
		k2 = (y[3] - y[2]) / (x[3] - x[2]);
		if (k1 == k2){
		printf("YES\n"); }
		else {
		printf("NO\n");} }
		return (0);}

Summary	ask a yes/no question via raw_input() and return their answer .	There are four points: A(x1, y1), B(x2, y2), C(x3, y3), and D(x4, y4). Write a program which determines whether the line AB and the line CD are parallel. If those two lines are parallel, your program should prints "YES" and if not prints "NO".
Example	**Example (e) Python Code-Summary from CodeNet and summary generated by PLBART**	**Example (f) Java Code-Summary from CodeNet and summary generated by PLBART**

Code	while True:
	t = int(input())
	if t == 0:
	break
	tmp = [int(input()) for i in range(t)]
	res = [tmp[0]]
	for i in range(1,t):
	res.append(max(tmp[i], tmp[i]+res[i-1]))
	print(max(res))
		public class Main{
		public static void main(String[] args) {
		Scanner scan = new Scanner(System.in);
		String str = scan.nextLine();
		str = str.toUpperCase();
		System.out.println(str);
		}
		}

Summary	Given a sequence of numbers a1, a2, a3, ...,an, find the maximum sum of a contiguous subsequence of those numbers. Note that, a subsequence of one element is also a contiquous subsequence. The input end with a line consisting of a single 0.	Write a program which replace all the lower-case letters of a given text with the corresponding captial letters. Print the converted text.
PLBART Summary	Reads input and prints the maximum value of t .	The main entry point for this class .

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[9] with pre-training objectives such as token masking, dele-

[9]https://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.