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

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).

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[1] 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-

[1]https://github.com/

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

Table 2: Dataset statistics

summary pair of Java in CodeSearchNet depicted in example (a) of Table 3, has comments in the code which have textual correlations with the summary.

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

Example	Example (a) Java Code with Comments from CodeSearchNet	Example (b) Java Code-Summary pair present in both 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 present in both 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 captital letters. Print the converted text.
PLBART Summary	Reads input and prints the maximum value of t .	The main entry point for this class .

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)

from Github and/or StackOverflow[2] with pre-training objectives such as token masking, deletion, 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.

[2]https://stackoverflow.com/

Approaches exploiting Program Analysis Information (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 (Ahmad et al., 2020; 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. 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).

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 Liguori et al. (2021), shows that parsing NL with generic variable names achieves better synthesis. Karmakar & Robbes (2021) 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. Yefet et al. (2020); Rabin et al. (2021) 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. Bielik & Vechev (2020) 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. Srikant et al. (2021) 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 ($\sim$ 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 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. Changing the operators would bring in change in the control flow dependencies

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
EXP-Tr-SDT	15.33	8.28	23.20	-0.21	19.41	-0.07	00.05	15.58	20.13	-0.83	15.07	3.38	4.36

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).

of the code and thus would lead to structural code changes. $\sim$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[3] and ast[4] packages. We detect the function and variable names by AST construction. The logical and arithmetic operators are detected by using regex[5]. SIT[6] is trained on TL-CodeSum and Python dataset and PLBART on CodeSearchNet. For having comparisons across the models, we fine-tune pre-trained PLBART[7] with TL-CodeSum and Python, where the codes are tokenized using the Tree-sitter tokenizer [8]. For fair comparison, we retain the hyper-parameter settings (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 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[9].

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).

[3] https://github.com/c2nes/javalang
[4] https://docs.python.org/3/library/ast.html#
[5] https://github.com/python/cpython/blob/3.10/Lib/re.py
[6] https://github.com/gingasan/sit3
[7] https://github.com/wasiahmad/PLBART
[8] https://github.com/tree-sitter/tree-sitter
[9] https://cloud.google.com/translate/automl/docs/evaluate#bleu

Example A	**Qualitative analysis for EXP-Te-SPT**			

Python code from Test set	Original Function
	def normalize(x):
	if x:
	def removeescape(matchobj):
	return matchobj.group(0)[1:]
	x = _simpleescapes(removeescape, x)
	return x.lower()
	else:
	return x
			Function after SPT
			def func(var1):
			if var1:
			def removeescape(matchobj):
			return matchobj.group(0)[1:]
			var1 = _simpleescapes(removeescape, var1)
			return var1.lower()
			else:
			return var1	

Summary	GT Original	normalizes x . normalize x .	GT EXP-Te-SPT	normalizes x . return zero if value is a x .
BLEU	70.71		21.10	
Example B	**Qualitative analysis for EXP-TrTe-SPT**			
Python code from Test set	Original Function def test_prewitt_h_mask(): np.random.seed(0) result = filters.prewitt_h(np.random .uniform(size=(10, 10)), np.zeros((10, 10), bool)) assert_allclose(result, 0)		Function after SPT def func(): np.random.seed(0) var1 = filters.prewitt_h(np.random .uniform(size=(10, 10)), np.zeros((10, 10), bool)) assert_allclose(var1, 0)	
Summary	GT Original	horizontal prewitt on a masked array should be zero . horizontal prewitt on a masked array should be zero .	GT EXP-TrTe-SPT	horizontal prewitt on a masked array should be zero . vertical sobel on a masked array should be zero .
BLEU	100.0		78.60	
Python code from Train set	Original Function def test_sobel_v_mask(): np.random.seed(0) result = filters.sobel_v(np.random .uniform(size=(10, 10)), np.zeros((10, 10), bool)) assert_allclose(result, 0)		Function after SPT def func(): np.random.seed(0) var1 = filters.sobel_v(np.random .uniform(size=(10, 10)), np.zeros((10, 10), bool)) assert_allclose(var1, 0)	
Summary	GT	vertical scharr on a masked array should be zero .	GT	vertical scharr on a masked array should be zero .
Example C	**Qualitative analysis for EXP-Te-SDT**			

Python code from Test set	Original Function
	def inplace_swap_column(X, m, n):
	if (m < 0):
	m += X.shape[1]
	if (n < 0):
	n += X.shape[1]
	if isinstance(X, sp.csc_matrix):
	return inplace_swap_row_csr(X, m, n)
	elif isinstance(X, sp.csr_matrix):
	return inplace_swap_row_csc(X, m, n)
	else:
	_raise_typeerror(X)
			Function after SDT
			def inplace_swap_column(X, m, n):
			if (m > 0):
			m -= X.shape[1]
			if (n > 0):
			n -= X.shape[1]
			if isinstance(X, sp.csc_matrix):
			return inplace_swap_row_csr(X, m, n)
			elif isinstance(X, sp.csr_matrix):
			return inplace_swap_row_csc(X, m, n)
			else:
			_raise_typeerror(X)	

Summary	GT Original	swaps two columns of a csc/csr matrix in-place . swaps two rows of a csc/csr matrix in-place .	GT EXP-Te-SDT	swaps two columns of a csc/csr matrix in-place . swaps two rows of a csc/csr matrix in-place .
BLEU	70.71		70.71	

Table 5: Qualitative Analysis with the SIT model trained on Python dataset. FN, VN, Operators

We understand that BLEU score, though widely used as a metric for code summarization (Shia et al., 2022), is not an appropriate metric to 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

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 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*, pp. 2655–2668, Online, June 2021a. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.211. URL https://aclanthology.org/2021.naacl-main.211.

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

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

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

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

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Pavol Bielik and Martin T. Vechev. Adversarial robustness for code. In *ICML*, 2020.

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

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

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

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

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

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

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

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Anjan Karmakar and Romain Robbes. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1332–1336. IEEE, 2021.

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

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

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

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

Pietro Liguori, Erfan Al-Hossami, Vittorio Orbinato, Roberto Natella, Samira Shaikh, Domenico Cotroneo, and Bojan Cukic. Evil: Exploiting software via natural language. *arXiv preprint arXiv:2109.00279*, 2021.

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

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

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

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

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

Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552, 2021.

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

Ensheng Shia, Yanlin Wangb, Lun Dub, Junjie Chenc, Shi Hanb, Hongyu Zhangd, Dongmei Zhangb, and Hongbin Suna. On the evaluation of neural code summarization. 2022.

Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. Generating adversarial computer programs using optimized obfuscations. *ArXiv*, abs/2103.11882, 2021.

Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*, pp. 2–13, 2020.

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

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

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

Hongqiu Wu, Hai Zhao, and Min Zhang. Code summarization with structure-induced transformer. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 1078–1090, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-acl.93. URL `https://aclanthology.org/2021.findings-acl.93`.

Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4:1 – 30, 2020.

Weizhe Yuan, Graham Neubig, and Pengfei Liu. Bartscore: Evaluating generated text as text generation. *Advances in Neural Information Processing Systems*, 34, 2021.

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

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.

Yuxiang Zhu and Minxue Pan. Automatic code summarization: A systematic literature review. *arXiv preprint arXiv:1909.04352*, 2019.

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