Contrastive Learning for Valid Code Generation

Dipti Lohia and **Jung Hoon Lim**
College of Information and Computer Sciences
University of Massachusetts Amherst

Abstract

Large-scale language models have shown great results across a lot of different tasks. One such field of research is code-based tasks such as code clone detection, defect detection, and Natural language to code generation. The existing approaches reveal that although these models can yield precise outcomes with particular inputs, the output may exhibit significant variability when subjected to insignificant perturbations in the input. Our objective is to conduct a comprehensive study to quantify the inherent inconsistency of existing code-based natural language processing (NLP) models. We will also determine what perturbations have what kind of effect on the performance of existing models.

```
1  #make below code smaller:
2  yusuke_power = {"Yusuke Urameshi": "Spirit Gun"}
3  hiei_power = {"Hiei": "Jagan Eye"}
4  powers = dict()
5  for dictionary in (yusuke_power, hiei_power):
6      for key, value in dictionary.items():
7          powers[key] = value
8  print(powers)
9
10
11 #dictionary comprehension
12 powers = {key: value for key, value in (yusuke_power, hiei_p
13 print(powers)
14
```

Figure 1: The following code snippet is from the open ai playground, for the codeX model. Lines 1 to 8 give the prompt to the model and lines 11 to 13 is the output produced by the model.

1 Introduction

1.1 Task Description

The dependence on language models has witnessed a remarkable surge in recent times, particularly in aiding programmers with software engineering tasks. These tasks include querying the language model to produce the most relevant code, conversion of code from one language to another, detecting if two code snippets are clones of each other, etc. Despite being trained on vast amounts of data, these models may still generate inconsistent outputs when presented with even slight variations in the input prompt. This questions the robustness of the existing language models. Here is a small example of what we mean by model robustness. Figure 1 shows a Python code snippet that utilizes a loop to add items to a dictionary, which was given to CodeX. When asked to make the code 'smaller', CodeX produced a dictionary comprehension version of the code, as depicted in the output. In Figure 2, we provided the same input but with additional newline characters. Surprisingly, the output was identical to the input, except for the removal of only one of the newline characters. For this project, we will first conduct a quantitative evaluation of the robustness of existing large models for three code-related tasks. These tasks are the code summarisation, the next token prediction, and the code clone detection tasks.

We will later concentrate on specific modifications that have been made to the code and carefully examine how alterations or perturbations to the code impact the output of the existing models.

1.2 Motivation and Limitations of Existing Work

As shown in the above figure, we see an unexpected behavior from existing models. The models are sensitive to small changes in the prompt. We, therefore, want to solve this drawback by making the existing models more robust.

1.3 Proposed Approach

We have divided the project into two phases. Phase 1 is conducting a study to quantify the robustness issue and Phase 2, we will focus on determining how the output is affected with specific types of per-

```
#make below code smaller:
yusuke_power = {"Yusuke Urameshi": "Spirit Gun"}
hiei_power = {"Hiei": "Jagan Eye"}
powers = dict()

for dictionary in (yusuke_power, hiei_power):


    for key, value in dictionary.items():
        powers[key] = value

print(powers)

#make below code smaller:
powers = dict()

for dictionary in (yusuke_power, hiei_power):

    for key, value in dictionary.items():
        powers[key] = value

print(powers)

```

Figure 2: The following code snippet is from open ai playground tool. Lines 1 to 12 is the prompt provided to the codeX model. The prompt has some additional new line characters as compared to Figure 1. Lines 14 to 22 provide the model output.

turbations instead of them being randomly applied to the code. Earlier we were planning on improving existing models by adding contrastive loss and then using soft prompt tuning or fine tuning technique. But since, the changes we observed were not extremely drastic(as shown in the future sections) therefore, we assumed that using contrastive examples would not improve model performance.

For our project, we focused on creating a dataset with small perturbations so as to test the existing code task models on how they perform. We tested on total of 4 existing models across the 3 tasks we defined earlier. We considered code written in python language due to the ease of availability of tools that will help us in data transformations.

2 Related Work

(Jain et al., 2020) proposed ContraCode, which pretrains a network by providing contrastive examples so that the model learns the semantics of the code instead of syntax. They observed that RoBERTa model was sensitive to code edits. The accuracy degraded significatly under adverserial perturbations.

Jiang et al. (Jiang et al., 2020) try to throw light on how much knowledge does language models have. They argue that models in fact have a lot of knowledge but are unable to translate that knowledge into a desirable output due to inappropriate prompts that are provided to them at the time of querying. They propose mining and paraphrasing based methods to generate high quality prompts.

(Webson and Pavlick, 2021) propose that models learn a lot even if the given prompts are irrelevant or made intentionally misleading. This raises the question if the models actually understand prompts like how humans do. They also conclude that models do learn faster using prompts and this is an area which can be explored. In our approach, we will also be using soft prompt tuning in context of code related tasks. This above mentioned paper also talks about the performance degradation of some models like ALBERT and T0 due to prompts that removed punctuations whereas models like T5 had a boost in it's performance.

CodeXGlue (Lu et al., 2021), developed by Microsoft, is a benchmark dataset that can be used for 10 code related tasks which fall under these four categories: Code-Code, Text-Code, Code-Text and Text-Text. They created a platform which contains BERT-style, GPT-style and Encoder-Decoder models which can be used by developers and researchers for model evaluations and comparisons.

3 Experiments

3.1 Datasets

3.1.1 CodeSearchNet

We plan to use CodeSearchNet (Husain et al., 2019), a public dataset with about 6 million functions from open-source code written in Go, Java, JavaScript, PHP, Python, and Ruby. In particular, we plan to start with Python source codes. The dataset has about 3.8 million, 490 thousand, and 52 thousand codes for the train, valid, and test set written in Python.

3.1.2 PoolC Code Clone Detection Dataset

We used a dataset that is registered in HuggingFace as "PoolC/i-fold-clone-detection-600k–5fold" dataset where $i \in \{1, 2, 3, 4, 5\}$ which is a dataset for the code clone detection task in Python. We used only a small portion of this dataset from this dataset where approximately 50% of the pairs were labeled as being similar. We observed that the two codes in each pair of this dataset differed significantly and based on the type of codes, and based on the comments and the types of functions, we conjecture that the codes were gathered as different solutions for the same question in coding interview preparation websites.

3.1.3 Transformations

We plan to use the LibCST package, which provides a concrete syntax tree of a Python source code, to slightly modify the source codes while preserving the semantics of the code. Through the concrete syntax tree, we can add, modify, or remove expressions such as classes, functions, whitespaces, assignments, import statements, and parenthesis in an automated way.

- **Adding characters**
  Adding new line character : We insert new lines of code for each new line character in the code
  Adding commas : We add commas at the end of lists, tuples, sets, and dictionary declarations
  Adding white spaces : We add one white space character in applicable locations of the code.

- **Removing characters**
  Remove new line character: We remove empty lines
  Removing white spaces: We remove existing white space characters from the code snippet.
  Remove commas: We remove commas at the end of list, tuple, set, and dictionary declarations

- **Removing texts with semantics**
  Remove comments: We remove comments and unassigned strings. When processing code snippets for summarization tasks, this helps to ensure that the resulting summary is based solely on the code's structure and logic rather than any additional information provided through comments.
  Remove unused imports: We remove unused imports. Imported names can have an effect on the understandability of codes for humans, which may be similarly applied to language models.

- **Style transfer (1)**
  Comprehensions to loop: Comprehensions are a concise way to create a new list, set, or dictionary by performing computations on iterable objects and filtering the results based on conditions. We changed comprehensions to a series of for loops and if statements while considering nested comprehensions as well.

- **Style transfer (2)**
  For loop to while loop: We change for loop to while statement when it involves the range function.
  Lambda to Function: a lambda function is a small function with an arbitrary number of arguments and a single statement. We extract these lambda functions in the codes and transform them to use the "def" keyword.
  Combine statements: Variable assignments in separate statements are combined if the number of targets and values match, no values to be assigned are included in the targets, and the yield keyword is not used.

- **Change of variable names**
  Local variable name change: We change local variable names inside function definitions by first detecting possible names excluding attributes, function names, parameter names, and non-local or global variable names since modifying these variables may break the codes. Then, we named variable names as $v0$, $v1$, ... if these variable names are not in the built-in scope, keywords, soft keywords, or used names.

We apply these transformations in random order. Given a variable $temp$ between 0.0 and 1.0, the transformations are applied with the probability of $temp$ whenever possible for all transformations except for modifying new line characters and whitespaces.

To check that our implementations for the transformations are valid, we tested that all of the transformations work even after applying the transformations to 110 thousand Python codes in our implementation and external packages used for the transformations.

3.2 Code Tasks

This section outlines the code-related tasks we selected for our experiment and presents the corresponding results. Our analysis includes qualitative assessments and quantitative assessments for summarisation and code generation and code clone detection, offering a comprehensive evaluation of our findings.

3.2.1 Code Summarisation

Given a Python code snippet, our objective is to produce a summary or a one line description of what that piece of code does. For our evaluation of existing model, we use codeT5 model from hugging face by Salesforce. This model is fine tuned

on the codeSearchNet dataset for code summarisation task.

Experiment Conducted

We take the existing model and feed the model with the transformed data of different temperatures. These temperatures are percentage of perturbations/transformations which are made to the code.

Experiment 1: In this experiment, we conducted an analysis using the codeT5 model by feeding it with the transformed data. The transformed data was obtained by applying a combination of multiple transformations. As mentioned earlier in the report, we performed this experiment at four different temperatures. The primary objective of Experiment 1 was to validate whether small variations in the data could lead to under performance of the model.

Experiment 2: For Experiment 2, we specifically focused on applying a particular transformation to the data and then feeding that transformed data into the model. The purpose of Experiment 2 was to analyze which specific changes had the most significant effect on breaking the model's performance. By systematically evaluating the performance of the codeT5 model on different datasets, each representing a specific change, we aimed to identify the changes that rendered the model most susceptible to failure.

Below, we provide a detailed report of the results obtained from both experiments.

Results and Analysis

Qualitative Assessments

Overall, we found that the results did not significantly underperform as per our hypothesis. However, upon closer examination, we identified some discrepancies in the results. For example, in Table 1, Row 4, we noticed that the reported results in each column varied slightly from each other. Some reports included additional information, such as the *random variate algorithm*, while others omitted important details like the *centered rectangle* from the ground truth. Interestingly, we also observed instances where the perturbations actually led to improved results. This can be seen in Row 1 of Table 1, where the output for $temp = 1$ included mentions of $type_a$ and $type_b$, which are similar to classes a and b of the ground truth, whereas the other outputs did not include such information.

Quantitative Assessments

Figure 11 illustrates the BLEU(Papineni et al., 2002) scores obtained in our analysis, where the reference file is the output of the model when no perturbations are applied (temperature = 0), and the prediction files are generated by the model at temperatures 0.2, 0.4, 0.7, and 1. We observe a consistent decline in the BLEU scores as the temperature increases. This decline indicates a decrease in the model's ability to generate outputs that closely match the reference texts.

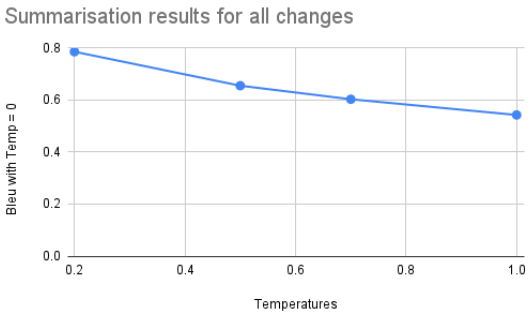

Figure 3: Bleu score between model outputs at different temperatures and when output of the model when no input is not perturbed.

Furthermore, we calculated the BLEU scores of the model's outputs at different temperatures using the ground truth of the model as the reference. The results show that the BLEU scores did not vary significantly, and the unigram precision score remained in the range of 0.326 to 0.327. This suggests that the model's outputs were relatively close to the ground truth.

However, the noticeable decline in the BLEU scores compared to the output at temperature 0 highlights the model's reduced stability in generating consistent answers as the temperature increases.

Figure 4 presents the BLEU scores obtained from Experiment 2, as mentioned earlier. In this experiment, we assessed the impact of different changes on the model's output. Notably, we observed that seemingly trivial changes, such as adding white spaces, altering variable names, and introducing new lines, had the most significant impact on the model's output. Surprisingly, changes that were anticipated to have a greater impact on the model's performance, such as transforming a for loop into a while loop, exhibited a comparatively lesser effect on the model's output. This finding suggests that certain changes, which may appear substantial, do not necessarily result in a significant alteration of the model's behavior.

Temp = 0.2	Temp = 0.5	Temp = 0.7	Temp = 1	Temp = 0	Ground Truth
Returns the KL function that is used to determine the KL.	Returns the KL function that is used to determine the KL.	Returns the KL function that is used to determine the KL.	Returns the KL of the type_a and type_b.	Returns the KL function that is registered for the given types.	Get the KL function registered for classes a and b.
Run and check the NVE v0 node.	Run Tenacity and check the response.	Run and check a single node in Tenacity.	Run Tenacity and check response.	Run and check the response of a single NVE request.	Grabs extra options like timeout and actually runs the request, checking for the result
Run the passset.	Run the n - term pass.	Run the circuit.	Run the circuit.	Run the passset.	Run all the passes on a Quantum Circuit
Resize an image to a new size using a random variate.	Resize an image by cropping it with a size.	Resize an image to a new size.	Resizes an image to fit the specified size.	Resize an image to a new size using a random variate algorithm.	Crop the image with a centered rectangle of the specified size
Creates a shell with a single node.	Creates a GUI shell.	Creates a GUI shell.	Display a single node in the shell.	Creates a shell with a single node.	Open a shell

Table 1: Outputs produced by codeT5 for different temperatures

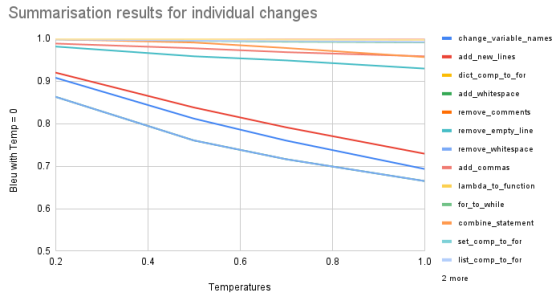

Figure 4: Bleu score between model outputs at different temperatures and when output of the model when no input is not perturbed for different individual changes made to the input

3.2.2 Code generation/ Next token prediction

The next token prediction task is a natural language processing task that aims to predict the next token. For code we aim to predict the next line of code given a partial input. In this case, we provide the model with the first half of a code snippet and expect it to generate the second half. To evaluate this task, we used the Codegen model available on Hugging Face, specifically the 350M-multi checkpoint. This model is pretrained on the BigQuery dataset, which contains code snippets from various programming languages, including C, C++, Go, Java, JavaScript, and Python. The dataset contains 119.2 billion tokens, making it a valuable resource for training and evaluating code generation models.

Experiment Conducted

For this task, we conduct the same experiment as that of the summarisation task. We split the test data into input into 2 parts such that the first half is also a code snippet, then we apply transformations to the first half of the input and feed that into the model.

Results and Analysis

Qualitative Analysis

The results of our experiment showed that the model didn't perform well with the perturbations. We saw drastic changes in the ground truth and the model output with the perturbed input . We have shown multiple examples of the output comparisons in Figure 5, 6.

Quantitative Analysis

The CODE-BLEU(Ren et al., 2020) scores were calculated by comparing the model's outputs with the ground truth of the input. Figure 8 visually depicts the impact of increasing temperature on the model's performance. It clearly illustrates a decline in performance as the temperature increases, indicating a negative effect on the model's ability to generate accurate outputs.

In Figure 7, we present the results of the CODE-BLEU scores for individual changes in the input, compared to the ground truth. Notably, we observed that specific changes had a significant ef-

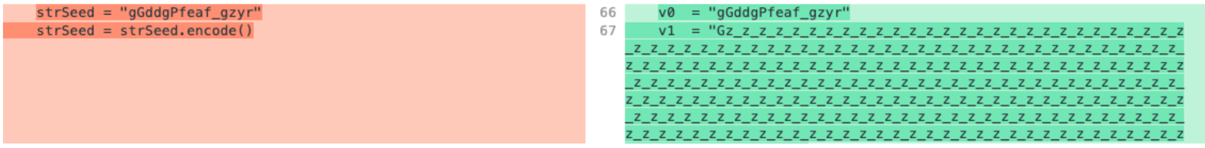

Figure 5: The image shows the output of the model with both augmented and non-augmented input. The orange highlighted part is the output when the input has no perturbations and the green highlighted part is when the input has perturabtions of temp = 0.7

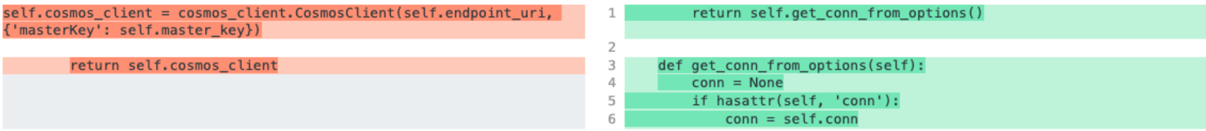

Figure 6: The orange highlighted part is the ground truth and the green highlighted part is model output when only new lines have been added to at temp =1

fect on the model's output. Adding new lines and removing empty lines had the most substantial impact, followed by the combine statement and changes in variable names.

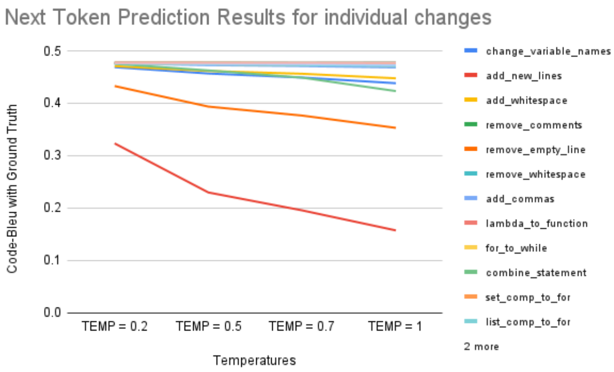

Figure 7: Bleu score between model outputs at different temperatures and ground truth for different individual changes made to the input

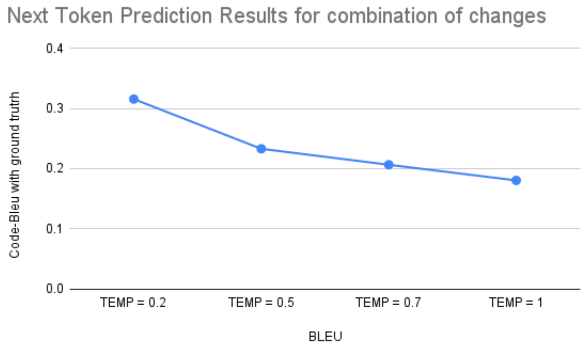

Figure 8: Bleu score between model outputs at different temperatures and ground truth

3.2.3 Code clone detection

The code clone detection task aims to classify whether two codes are clones of each other. For this task, we fine-tuned a two pre-trained encoders, which are the CodeBERT base model (Feng et al., 2020) and the encoder of CodeT5(Wang et al., 2021), in a Siamese architecture. Let $Encode(code)$ denote the [CLS] vector output of the encoder where $code$ is inputted in the format of "[CLS], tokenized $code$, [SEP]" to the model. Also, let $Dropout(v)$ and $Linear(v)$ be the output of a dropout layer with drop probability of 0.1 and when v is inputted and the output of v when passed through a linear layer with an input size of 768 and an output size of 768 respectively. Given two codes $code_1$ and $code_2$, we calculate the logit between the two codes for classification as the dot-product of $Linear(Dropout(Encode(code_1)))$ and $Linear(Dropout(Encode(code_2)))$. We use the sigmoid of the logit as the prediction probability and calculate the loss using the binary cross entropy loss over the prediction probability.

We fine-tuned the model with CodeBERT base and CodeT5 encoder with 400 thousand pairs of codes with learning rate of 1e-6 and 1e-5 respectively. Batch size of 8 and a linear learning scheduling with warmup steps of 5,000 are applied.

Experiment Conducted

First, evaluate the performance of our model, we found the threshold for logits that maximizes the f1 score using 30 thousand instances in the validation dataset of PoolC.

Next, we quantified the inconsistencies of our model by using augmented codes as the input to the model. In particular, given $code_1$ and $code_2$, we modified only $code_1$ with groups of transforma-

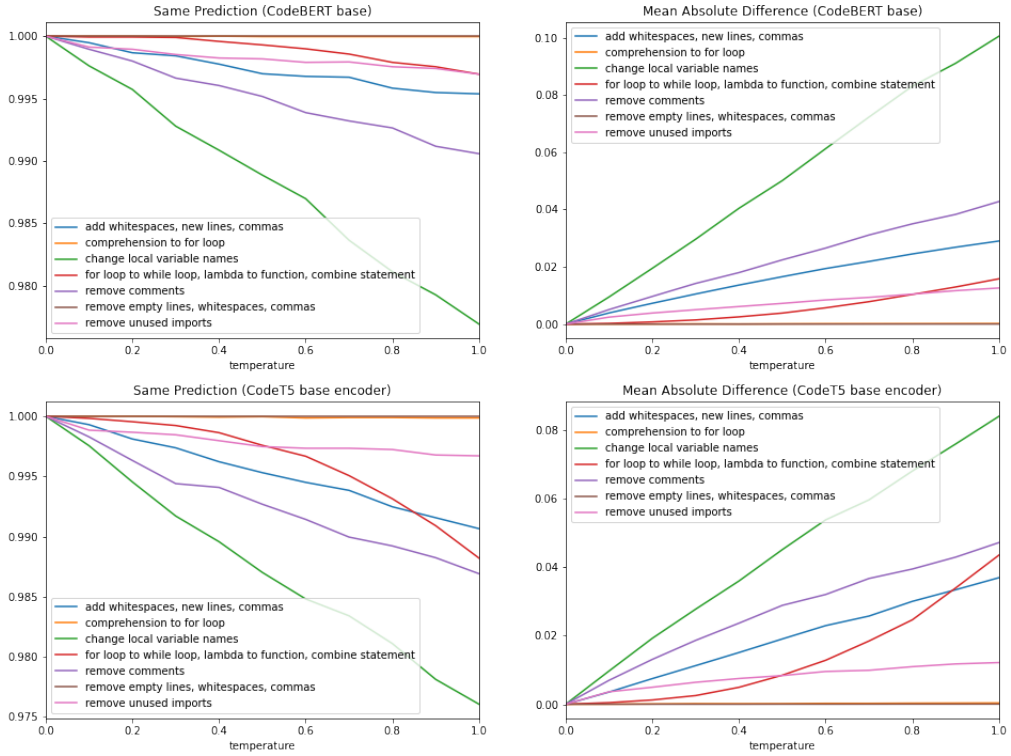

Figure 9: This figure shows quantitative results for the code clone detection task. The two upper and two lower graphs show the results when the CodeBERT base and the CodeT5 base encoder were used, respectively. The two left and two right plots show the same prediction rate and the mean absolute difference in logits, respectively.

```
import math
A,B,N, = map(int,input().split())
ret = 0
ans = 0
if B == 0:
  print(0)
  exit()
if B > N:
  base = N
else:
  base = B - 1
ret = math.floor(A*base/B) - A*math.floor(base/B)
print(ret)
----------------------------------------------------------------
data = [int(i) for i in input().split()]
print(f'{data[0] * data[1]} {data[0] * 2 + data[1] * 2}')
```

Figure 10: An example when the prediction for the code clone detection task changed by adding a comma. The upper code shows $code_1$ after transformation where the only difference compared to the original code is that the third comma in the second line was added. The lower code is $code_2$ and the two codes are labeled to be not clones.

```
def rally(arr):
        p = max(arr)
        v0 = 0
        best = float('inf')
        for i in range(1, p+1):
                v0 = 0
                for j in arr:
                        v0 += (j - i) ** 2
                best = min(best, v0)
        return best
n = int(input())
arr = list(map(int, input().split()))
print(rally(arr))
----------------------------------------------------------------
n = int(input())
x = list(map(int, input().split()))
ret = 1000000
for p in range(1, 101):
    ret = min(ret, sum((x - p) ** 2 for x in x))
print(ret)
```

Figure 11: An example when the prediction for the code clone detection task changed by changing local variable name. This example shows the code after transformation where variable ”v0” was changed from ”summ”. The lower code is $code_2$ and the two codes are labeled to be clones.

tions and varying temperatures with a step size of 0.1 using 30,000 pairs of codes. We observed how the logits and the predictions for whether $code_1$ and $code_2$ are clones vary. We also went through qualitative analysis by looking at some samples where the prediction or logits has changed significantly.

Results and Analysis

The number of changes made in the 30,000 samples that we used are 33526, 23948, 13843, 5343, 45277, 733945, and 462930 for Change of local variable names, Removing comments, Removing unused imports, Style Transfer(1), Style Transfer(2), Add characters, and Remove characters respectively. The graphs of the same prediction rate and the mean absolute difference in logits when temperature varies are shown in Figure 9. For some other metrics, the accuracy remained similar by being in the range of 83.8% - 85.2% for CodeBERT and 76.2%, and 77.0% for codeT5. This was the same in the standard deviation in absolute logit differences, as the standard deviation was between about 1.745 and 1.765 for the CodeBERT base and between 1.375 and 1.382 for the CodeT5 base.

The first thing that we could observe through Figure 8 was that the models fine-tuned in Siamese architecture were robust to the input changes. This was because even with the many changes made in the 30,000 samples, the same prediction rate remained high in all cases, and the absolute differences in logits were small compared to their respective standard deviations.

The second observation is that transformations that can change the semantics shown in natural languages, such as variable names or comments, had bigger effect on the model's robustness. The related groups of transformations are "Change of local variable names," "Removing comments," and "Removing unused imports." While the number of changes is typically small, both the same prediction rate and the mean absolute differences in logits remained relatively high for these three transformations.

Finally, we investigated the qualitative analysis. We observed some cases when adding a comma made a big difference in the predictions, and an example is shown in Figure 10. The logit changed from 0.512 to 0.701, changing the prediction incorrectly from False to True. An example of changing variable names is shown in Figure 11 where variable "summ" changed to "v0", and the prediction was changed correctly from False to True while the logit changed significantly from 0.206 to 1.19.

3.3 Software

We used the LibCST package to modify code while preserving its syntax. Regarding models, we employed pre-trained or fine-tuned natural language processing (NLP) models provided by HuggingFace. For deep learning tools, we utilized PyTorch, one of the Python packages that is compatible with HuggingFace models.

3.4 Conclusion

Our experiments revealed the susceptibility of models to small variations in the input. Notably, specific changes such as adding new lines and altering variable names consistently resulted in a drop in the model's performance across both the summarisation and next token prediction tasks. This suggests that the Codegen model, in particular, exhibited the lowest level of robustness among the tested models.

In the case of summarization, while qualitative analysis of a few examples did not indicate significant underperformance, quantitative analysis using BLEU scores highlighted a decline in performance. This observation emphasizes the need to consider both qualitative and quantitative evaluations to obtain a comprehensive understanding of the model's performance.

The code clone detection task, however, showed that our models are robust to input changes. Further investigation is needed since this may be because the code pairs in the dataset had low similarity with each other.

The findings from this study pave the way for future research aimed at enhancing the model's robustness. Researchers can explore techniques such as feeding the model with contrastive examples and examples with minimal changes to further improve its performance. By addressing the model's sensitivity to variations and focusing on robustness, we can advance the capabilities of the model and enhance its reliability in real-world applications.

We are currently preparing our work for submission to a relevant workshop. We are actively searching for a workshop that aligns with the focus and themes of our research to ensure the best fit for our work.

References

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*.

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*.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973*.

Zhengbao Jiang, Frank F Xu, Jun Araki, and Graham Neubig. 2020. How can we know what language models know? *Transactions of the Association for Computational Linguistics*, 8:423–438.

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.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

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*.

Albert Webson and Ellie Pavlick. 2021. Do prompt-based models really understand the meaning of their prompts? *arXiv preprint arXiv:2109.01247*.