Contrastive learning for valid code generation

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

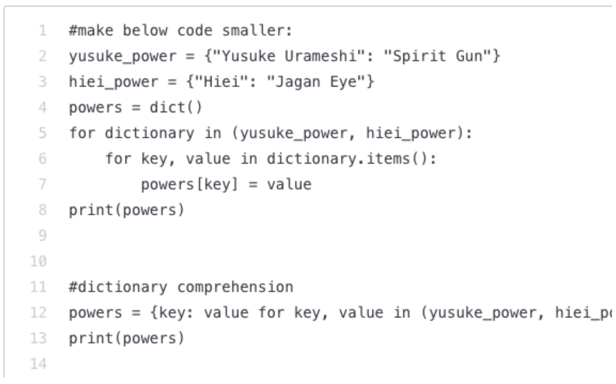
Abstract

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, 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 which we targeted in our previous step.

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 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 robustness on existing large models for three code related tasks. These tasks are



```
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
```

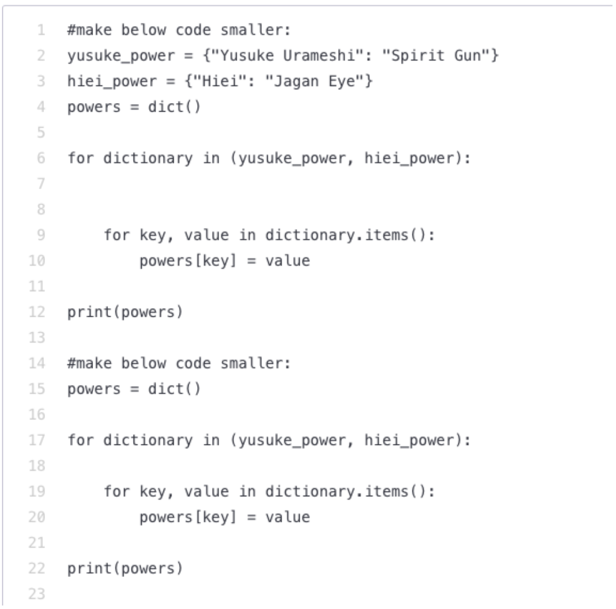
Figure 1: The following code snippet is from the open ai playground, for the codeX model. Lines 1 to 8 give the prompt to the model and lines 11 to 13 is the output produced by the model.

- code summarisation
- code clone detection
- Next token prediction

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   #make below code smaller:
2   yusuke_power = {"Yusuke Urameshi": "Spirit Gun"}
3   hiei_power = {"Hiei": "Jagan Eye"}
4   powers = dict()
5
6   for dictionary in (yusuke_power, hiei_power):
7
8
9       for key, value in dictionary.items():
10          powers[key] = value
11
12  print(powers)
13
14  #make below code smaller:
15  powers = dict()
16
17  for dictionary in (yusuke_power, hiei_power):
18
19      for key, value in dictionary.items():
20          powers[key] = value
21
22  print(powers)
23
```

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.

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 perturbations 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 mpdel performance.

For our project, we will focus on creating a dataset with small perturbations so as to test the existing code task models on how they perform. Currently we will focus on testing only 3 existing models of the 3 tasks we defined earlier. We will also only consider 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 (about 65,000 pairs of codes) 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, it's important to avoid any potential bias that comments within the code may introduce. 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 one type of loop format to another. This involves replacing the loop's initialization, condition, and increment statements
  - *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 key-word.
  - *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, 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 first checked the results by using sample inputs. Then, we tested that all of the transformations work even after applying the transformations to some of the codes that are used for the transformations, which consist of about 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 for summarisation and code generation and quantitative assessments for 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. We have conducted our experiment on 4 different temperatures : 0.2, 0.5, 0.7, 1. We have reported the qualitative analysis of the results below.

Results and Analysis 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.

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
We take the existing model and feed the model with the transformed data of temp = 0.7 for now. We aim on extending it to all the temperatures (similar to the summarisation experiment). First, we split the input such that the first half is also a valid code snippet. Then we apply perturbations to the first half of the code and feed that into the model.

Results and 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 show an example of this in Fig 1. Therefore, in case of the next token prediction task our hypothesis is correct. We have evaluated results qualitatively till now. Further we will also calculate Code-bleu score of these result.

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 pre-trained CodeBERT base model (Feng et al., 2020) in a Siamese architecture. Let $Encode(code)$ denote the [CLS] vector output of the CodeBERT model where $code$ is inputted in the format of "[CLS], tokenized $code$, [SEP]" to the model. Also, let $Linear(v)$ be the output of v when passed through a linear layer with an input size of 768 and an output size of 768. Given two codes $code_1$ and $code_2$, we calculate the logit between the two codes for classification as the dot-product of $Linear(Encode(code_1))$ and $Linear(Encode(code_2))$. We use the sigmoid of the logit as the prediction probability and calculate

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

the loss using the binary cross entropy loss over the prediction probability. We fine-tuned our model using about 250 thousand instances of the PoolC dataset.

Experiment Conducted

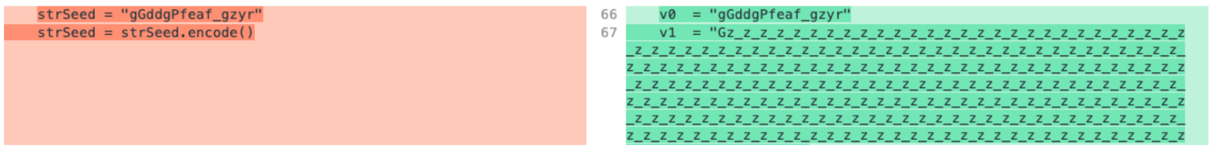
To evaluate the performance of our model, we found the threshold for logits that maximizes the f1 score using 30 thousand instances in the validation dataset of PoolC. Next, we quantified the inconsistencies of our model by using augmented codes as the input to the model. In particular, given $code_1$ and $code_2$, we modified only $code_1$ with groups of transformations and varying temperatures with a step size of 0.1 and observed how the prediction of whether $code_1$ and $code_2$ differs.

Results and Analysis

We used 10000 instances of code pairs for this part. For the ith instance, let $logit^i_{temp}$ indicate the logits between when only one code is augmented with temperature of $temp$. $temp$ being 0.0 means that no transformations have been applied.

The accuracy and the same prediction rate with varying transformation groups and temperature are shown in (a) and (b) of Figure 4. While accuracy is near 0.85, the same prediction rate was between 0.9874 and 1.0. Let $M(temp, group)$ be the mean of the absolute difference between logits of $logit^i_{temp}$ and $logit^i_{temp}$ where $i \in 1, \ldots, 10000$ given temperature $temp$ and a transformation group $group$. The standard deviations of $logit^i_{temp}$ and $M(temp, group)$ are shown in (c) and (d) of Figure 4 respectively. We could observe that the fine-tuned model is robust to the transformations applied.

For further investigation, we show the results of $M(temp, group)$ and the number of changes made when temperature is 1.0 in Table 2. We could see that the variable name changing had the biggest effect followed by removing comments while style transfer had a relatively small effect. The least change was made in removing characters and we conjecture that this is because adding new line or whitespace were ignored during tokenization. Among 156031 changes made for removing char-



```
    strSeed = "gGddgPfeaf_gzyr"
    strSeed = strSeed.encode()
```

```
    v0  = "gGddgPfeaf_gzyr"
    v1  = "Gz_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z
_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_
z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z
_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_
z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z
_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_
z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z_z
```

Figure 3: 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

acters, only 70 were removing commas and this may be the only changes that changed the model inputs.

3.3 Evaluation metrics

BLEU, proposed by (Papineni et al., 2002), is a metric that measures the similarity between reference translations and machine translated text. We can use this metric for code summarisation task. F1 score can be used for classification tasks, In this case we can use it for Code clone detection and Next Token prediction. BLEU-4 can also be used for code summarisation tasks as well. It has been established that BLEU scores are not the best way to evaluate the model, therefore, we can also use EM(exact match) and CodeBLEU (Ren et al., 2020) for code generation tasks.

3.4 Software

Our strategy involves utilizing the LibCST package to modify code while preserving its syntax. Regarding models, our plan is to employ pre-trained or fine-tuned natural language processing (NLP) models provided by HuggingFace. For deep learning tools, we have selected PyTorch, one of the Python packages that is compatible with HuggingFace models.

3.5 Timeline

- Week 1-3: ~~Literature review, learning new concepts~~
- Week 4-8: ~~Data preparation and baseline setup and evaluation on new data generated.~~
- Week 9-12: Evaluate the models performance on specific perturbations
- Week 13-14:Record the results and write final report.

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*.

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

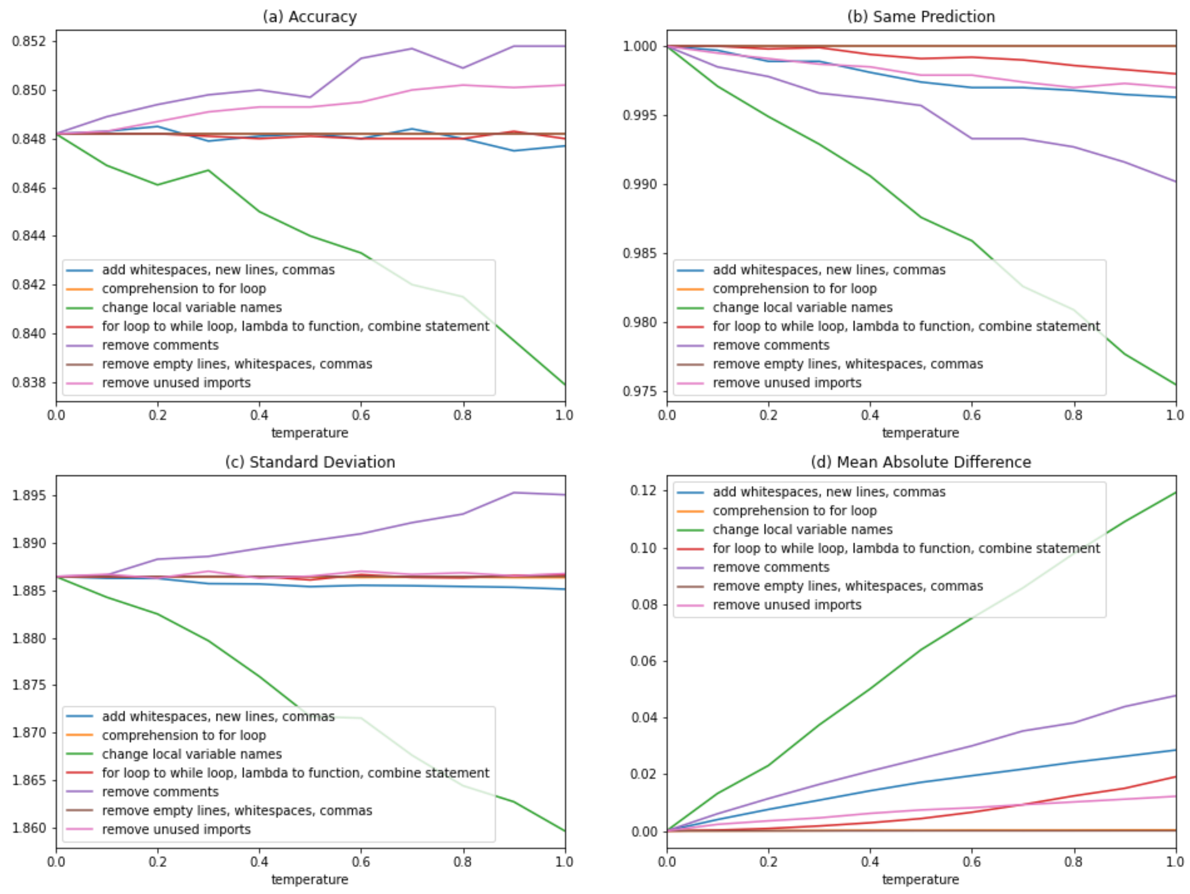


Figure 4: This figure shows quantitative results for the code clone detection task. (a) and (c) show the accuracy and the standard deviation of logits based on varying perturbation temperature. (b) and (d) show the same prediction rate and the mean absolute difference in logits with and without perturbation.

Group (group)	**Accuracy**	$\mathbf{M(temp, group)}$	**Number of changes**
None (original)	0.8482	0.0000	0
Adding characters	0.8477	0.0285	251370
Removing characters	0.8482	5.910e-5	156031
Removing comments	0.8518	0.0477	8011
Removing unused imports	0.8502	0.0122	4332
Style transfer (1)	0.8482	0.0003	2399
Style transfer (2)	0.8480	0.0191	14790
Change of local variable names	0.8379	0.1196	10832

Table 2: Values of $M(temp, group)$ and number of transformations applied for various groups when $temp$ is set to 1.0.