Evaluating Accuracy and Robustness and Mitigating Hallucinations in Salesforce CodeGenie AI Code Generation Tool

by
Christian Natajaya
Jeremy Szeto
Ryan Chien
Syam Sundar Kirubakaran
Shrey Mathur

Advisors:
Xiang Anthony Chen
Katie Stasaski
Yingbo Zhou

University of California, Los Angeles
Henry Samueli School of Engineering and Applied Science

August 30, 2023

Abstract

Artificial intelligence-based text generation is currently at the forefront of natural language processing research and technology, thanks in large part to the introduction and rise in popularity of ChatGPT. Along with this significant recent explosion in text generation technology, there have been methods created to evaluate the text for accuracy, comprehension, and syntax, along with other metrics. Artificial intelligence-based code generation, however, lacks a comprehensive and thorough set of these evaluation metrics. To illustrate just one problem that may be caused by this, accurate code generation for mathematical equations is critical for various fields including education, research, and application development. Therefore, there is a need for reliable code generation that can assist students, professionals, and researchers in implementing mathematical models in their work. Thus, our capstone project revolved around identifying holistic evaluation metrics that can provide insights beyond only accuracy to better facilitate downstream tasks. Using the CodeGen2.5 code autocompletion engine developed by Salesforce, we passed perturbed code functions as inputs to the model and measured the Levenshtein edit distance and embedded token similarity scores of the corresponding outputs against the ground truth reference code and unperturbed input generated code to test the accuracy and robustness of the CodeGen2.5 large language model (LLM). Our results showed trends that the model is not very accurate when evaluating against reference code as ground truth, and it is also not robust to different types of input perturbations at small degrees of perturbation. From these results we can reasonably conclude that the CodeGen2.5 model is susceptible to any amount of perturbation, highlighting the importance of robustness as an evaluation metric when analyzing performance of CodeGen2.5, and more broadly, LLMs in general. Additionally, our results show that the model struggles to generate accurate code or hallucinates when prompted to write functions specifically intended to calculate the results of mathematical equations, display graphical plots, or import popular Python libraries, such as numpy and pandas, with uncommon aliases.

Acknowledgements

We would like to thank each and every one of the individuals and organizations who played a part in helping us complete this capstone project and report.

First and foremost, we would like to thank our industry advisors, Katie and Yingbo, for their guidance, availability, and subject matter expertise. Their patient support has been a testament to both Salesforce's as well as their own personal commitment to education and research excellence.

We would also like to thank Dr. Chen, our academic advisor, for linking us with the Salesforce representatives, as well as for his insight and dedication. His advice directed us in picking a meaningful problem and refining the scope of our work.

We thank the Masters of Engineering program's staff and faculty for their instruction and advocacy throughout the course of our studies. We could not have made it this far without them.

Finally, we would like to thank our classmates, whose shared experiences, diverse viewpoints, and constructive criticism ultimately improved the completeness and breadth of our capstone experience.

Completion of our project was made possible with the assistance of these amazing people and organizations. We hope that our work honors the inspiration of the people named above just as much as it honors our own.

Table of Contents

Chapter One: Problem Definition

1.1 Overview and Motivation

The current state-of-the-art evaluation metric for computer-generated code is CodeBLEU, which uses n-gram matching based on code syntax to measure the accuracy of the generated code. CodeBLEU's reliability is derived from comparing the correlation coefficients of accuracy alongside quality scores assigned by programmers. So while CodeBLEU makes sense in theory and computation, it lacks practicality when it comes to real-world application. Other metrics have been developed, such as CodeBERTScore, which relies on token embeddings to evaluate similarity between code generated from models. However, similar to the flaws with CodeBLEU, CodeBERTScore lacks in applied practicality and has not shown solid linkage between theory and results. With these clear shortcomings in the current state-of-the-art evaluation metrics, our group sought to find a more reliable and holistic code generation evaluation metric.

1.2 Problem Statement

The fundamental issue with state-of-the-art evaluation metrics is that they are too heavily based on accuracy or "correctness" of the generated tokens. In code syntax much more so than in natural language syntax, accuracy can be highly variable, with a large number of factors that could impact the metric. For instance, different problem types such as arithmetic versus planning problems could have wide ranging outcomes that would vary the accuracy results. Different phrasings of the same input prompt could have the same meaning in the context of natural language, but the code generation model may misinterpret the modified input, leading to generation of an output with high variance. For large language models (LLMs) setting different values for parameters, such as temperature and top-p, that affect the model's creativity can also deliver varying accuracy results. In summary, accuracy does not always encompass reliability and trustworthiness of the generated code; somewhat incorrect code might still have correct logic and somewhat accurate code might have incorrect logic.

1.3 Proposed Solution

To propose a solution addressing this problem, for our capstone project we wanted to create a set of holistic evaluation metrics for Salesforces' code autocompletion tool, CodeGenie (aptly named after the CodeGen AI model that it is based on). Our specific goal was to create metrics that Salesforce developers would be able to consult and further use to assist in improving their CodeGen LLM such as through prompt engineering or architecture, feature, or parameter updates. We elected to evaluate the model on two criteria: accuracy and robustness.

1.3.1 Mitigating Hallucinations

Using our criteria for accuracy, we can find subdomains where the model is inaccurate, as these subdomains could be a result of hallucinations. As an addition to our investigation into evaluation metrics, we propose a solution to create a fine-tuned dataset to retrain CodeGen2.5 where it underperforms in accuracy. Refer to Chapter Three: Methods for how we create this dataset. Since there are potentially quite a large range of subdomains that could underperform in accuracy, it would be unfeasible to explore each and every single one of these subdomains. For the purpose of this paper, we found that CodeGen2.5 doesn't work well with mathematical equations in particular, so we focus on the subdomain of mathematical equations. Overall, our goal is to develop a tool to create a fine-tuned dataset which can help mitigate inaccuracies of CodeGen2.5 while ensuring robustness for that particular subdomain.

1.4 Definition of Key Terms

1.4.1 Accuracy

Accuracy can be defined as the state of correctness, and in the terms of code generation evaluation, how closely the generated code resembles the reference code. In order to calculate accuracy, we should not only consider the textual correctness but also the syntax and semantics of the output, as a function that "looks" different from a reference output might still functionally behave the same.

1.4.2 Robustness

Robustness is defined as how well the code generated from the model performs when perturbations and noise are added to the input prompt. Robustness of generated code can be benchmarked by identifying the percentage of accurate results produced despite having noisy input prompts. Noisy input prompts can include incomplete and ambiguous prompts, or other forms of input imperfections. By assessing the robustness of generated code, developers and researchers can gauge the code's ability to handle real-world scenarios where input data may not always be perfect. This evaluation helps identify areas for improvement, refine algorithms, and enhance the overall reliability and performance of CodeGen2.5.

1.4.3 Hallucinations

Hallucinations in LLMs refer to instances where the model generates text that appears coherent and contextually relevant but contains factual inaccuracies or information that is not grounded in reality. These inaccuracies arise due to the model's tendency to over-optimize for patterns in training data, resulting in outputs that may seem plausible but lack accurate content, potentially leading to the spread of misinformation or biased information.

Chapter Two: Literature Review

2.1 CodeBLEU

The current state-of-the-art solution for code generation evaluation is CodeBLEU [1]. It is derived from BLEU, an n-gram word matching score used to evaluate natural language. Simply put, CodeBLEU is the BLEU score adjusted for syntax found in code. To encourage the argument for why this difference needs to be highlighted, here are three primary differences between natural language and code: limited keywords versus millions of words, tree structure versus sequential structure, and unique instructions versus ambiguous semantics. In code, there are reserved words required for programming, which leads to fewer sets of words to match for as opposed to the millions of words found in natural language. Additionally, code has a tree based structure that follows a specific, standardized, and systematic syntax tree and natural language follows a sequence that can be ambiguous and variable semantic at times. These differences make analyzing code uniquely difficult from natural language.

CodeBLEU is calculated using a weighted combination of four parts: a weighted n-gram match (obtained by comparing the hypothesis code and the reference code tokens with different weights), a syntactic AST match (exploring the syntactic information of code), a semantic data-flow (considering semantic similarity between hypothesis and reference), and the standard BLEU score based on natural language. Studies show that this score is informative and useful for three primary reasons: the difference between CodeBLEU and standard BLEU is reliable and can differentiate code and natural language, the CodeBLEU score itself is reliable, and CodeBLEU is more correlated with human evaluation scores than traditional accuracy metrics.

For our project, we used CodeBLEU as a state-of-the-art reference for how evaluation metrics are applied to code generation. We ultimately opted to use Levenshtein edit distance and CodeBERT token embedding similarity scores to evaluate the CodeGen2.5 model to provide value and a new perspective outside of CodeBLEU evaluation.

2.2 CodeBERT

CodeBERT is a bimodal pre-trained model for programming language (PL) and natural language (NL) [2]. CodeBERT captures the semantic connection between natural language and programming language and produces token embedding representations to be used for downstream NL-PL tasks, including code generation tasks. Compared to RoBERTa, a purely natural language pre-trained model, CodeBERT consistently outperforms it for NL-PL downstream tasks. With a similar transformer model architecture to RoBERTa, CodeBERT is trained with two objectives: masked language modeling and replaced token detection. Using both an NL and PL generator, the NL-code discriminator is used to determine whether the generated tokens are “real” or “fake”. This process is more specifically shown in the figure below.

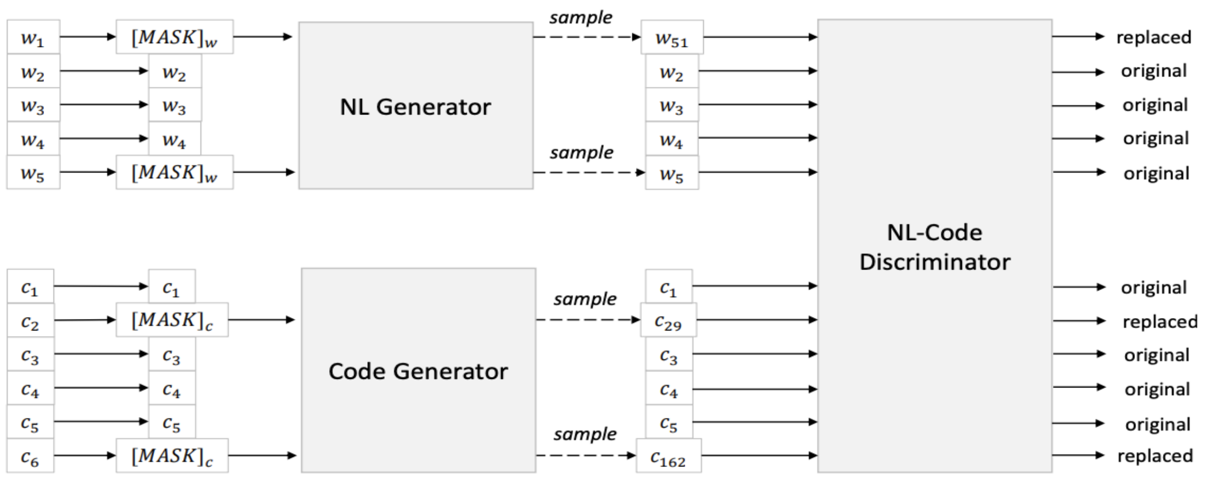

Figure 2.1: Image of how CodeBERT generates tokens for both NL and Code for pre-training objectives and how the discriminator detects plausible alternative tokens sampled from the generators, as well as produces general-purpose representations during fine-tuning

For our project, we make use of the CodeBERT embeddings to evaluate the similarity scores between the gold “correct” code file and the CodeGen model code file output. By making use of the semantic embeddings from CodeBERT, similarity scores can be more forgiving yet still accurate when evaluating code generation tools.

2.3 Code Robustness Evaluation for LLM

This literature introduces two key concepts when it comes to evaluating LLMs for code robustness: methodology for perturbing the input and current metrics used for evaluation [3]. They use two quantitative evaluation metrics to measure code generation: solved rate and averaged solved rate. Solved rate is calculated by taking the total number of correctly solved programs and dividing by the total number of programs. Average solved rate is calculated by averaging the solved rate for each program over the entire dataset. It is important to note that for our project, much of the dataset does not have a problem that requires solving. Therefore, we ultimately decided to opt for a solution that does not utilize solve rate.

The paper also hypothesized that perturbations to the code file can be categorized into two groups: problem formatting and prompt formatting. Problem formatting is described as manipulating the problem descriptions to explicitly represent the problem specifications. These modifications only change the superficial formats and don’t actually change the problem, therefore these are often labeled as superficial modifications. The second category is defined as prompt formatting, which manipulates the input text for the LLMs. The ways you can implement prompt formatting are by: specifying the programming language, providing input as a docstring or comments, appending instruction commands, defining a solve() function, providing import statements, or inserting/removing/preserving lines of code.

For the purposes of our project, the prompt formatting approach was more applicable and related. We were able to take some of these different implementations of prompt formatting into different perturbations we made to the CodeGen2.5 model input. We were also able to brainstorm different hallucinations the CodeGen2.5 model could make based on the different types of prompt formatting.

2.4 CodeBERTScore

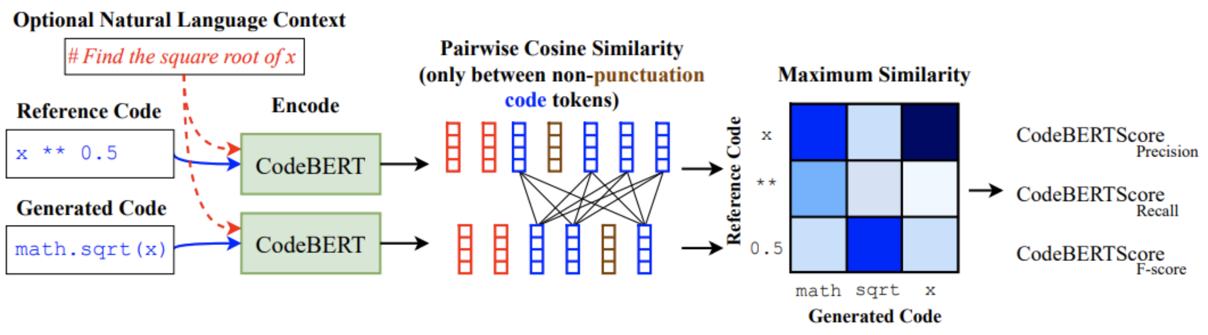

Figure 2.2: Diagram how CodeBERTScore is calculated between generated code and reference code

CodeBERTScore computes a soft similarity score between each token in the generated code file and the gold reference file using the contextual encodings of large pretrained models (such as CodeBERT) [4]. CodeBERTScore first encodes the generated code and the reference code independently with pretrained models. The soft similarity score is computed by taking the dot-product similarity between the encoded representations for each token from the generated code against the reference code; this process is illustrated in the above figure. The results from the paper show that CodeBERTScore is highly correlated with both human preference metrics and functional correctness metrics, both of which are factors that our group considers crucial to code evaluation.

Similar to the research performed in this paper, we utilized the CodeBERT token embeddings to calculate similarity scores between a reference code file and a perturbed code file from the CodeGen2.5 model.

2.5 CodeGen

Our project revolved around the evaluation of Salesforce’s code generation tool, CodeGen (specifically CodeGen2.5) [5]. The first version of CodeGen was released in March 2022, with the goal of automating the coding process, and generating a computer program that satisfies the user’s specific intent. Generally, a code generation model in compiler architecture refers to the process of translating the intermediate representation of a program into the target machine code. It generates efficient and optimized code that can be executed by the target hardware; Salesforce has implemented their own version of this in their CodeGen tool and CodeGenie Visual Studio Code extension.

The code generation model can be broken into six stages:

1. Lexical Analysis: The source code is divided into tokens. Each token represents a meaningful unit such as keywords, identifiers, operators, or literals.

2. Syntax Analysis: The parser analyzes the tokens and checks if they conform to the grammar rules of the programming language. It builds a parse tree or an abstract syntax tree (AST) representing the structure of the program.

3. Semantic Analysis: The semantic analysis phase verifies the correctness of the program's semantics. It checks for type errors, undefined variables, and other semantic rules. The output of this phase is an annotated AST.

4. Intermediate Code Generation: The intermediate code generation phase translates the annotated AST into an intermediate representation (IR) code. The IR is a platform-independent representation that captures the essence of the program's behavior.

5. Optimization: Analyzes the IR code and applies various optimization techniques to improve the efficiency of the generated code. It eliminates redundant computations, reorders instructions, and performs other transformations to produce optimized IR code.

6. Code Generation: The final stage is the code generation itself. The code generator takes the optimized IR code and maps it to the target machine's specific instructions. It generates the actual machine code, including instructions for memory access, arithmetic operations, control flow, and function calls.

The code generation model aims to produce efficient and correct machine code that closely matches the behavior of the original program. It involves making decisions about register allocation, instruction selection, and memory management to optimize the performance of the generated code. For our project, we used a quantized version of the CodeGen2.5 model to generate code output. Reasons for this choice can be found in the Chapter 3: Methods section below.

2.6 Interpretability of Existing Pre-trained Code Models

Code representation learning has primarily been illustrated by programmers through code embeddings [6]. It aims to encode the code semantics into vector representations and is the basis for models for code intelligence. This paper addresses the issue that very little progress has been made regarding the interpretability of existing pre-trained code models. There has been little research done on the *why* and *how* these models work and what features they can capture. The authors of the paper conduct structural analysis into CodeBERT and GraphCodeBERT from three different perspectives, all analyzing how the models learn syntactic structure: attention analysis, probing on the word embedding, and syntax tree analysis. The paper found that pre-trained code models can learn grammatical/syntactic information, after results showed that the encoders for the transformers had learned the syntactic properties within the hidden layers.

This paper highlights one of the primary limitations of our research study. Without fully understanding how a pre-trained code model learns the syntactic structure of the code input to how it outputs it for downstream tasks, we will remain uncertain about how reliable using similarity scores for the token embeddings is.

2.7 Mitigating Hallucinations

Despite the remarkable success of LLMs in generating coherent text, their tendency to hallucinate certain parts of text limits their widespread adoption in real-world applications [7]. In order to improve text generation by LLMs, the authors propose an approach to identify and mitigate hallucinations. In the detection stage, the authors first identify the important concepts, calculate the model's uncertainty on them, and then validate the correctness of the uncertain concepts by retrieving relevant knowledge from the internet. In the mitigation stage, the hallucinated sentence is repaired using the retrieved knowledge as evidence. We referred to this paper to gain insights into strategies for mitigating hallucinations. Inspired by their approach, we attempted to extract data from the web and use it to generate the ground truth output.

2.8 Code Synthesis using Large Language Models

This paper explores the capabilities of current generation LLMs for program synthesis in general-purpose programming languages. The authors evaluated a collection of such models on two new benchmarks, MBPP (Mostly Basic Programming Problems) and MathQA-Python, in both the few-shot and fine-tuning regimes [8]. They found that LLMs perform extremely well in generating short Python programs. The largest models can synthesize solutions for up to 59.6% of the problems from MBPP using few-shot learning, and the largest fine tuned model achieved 83.8% accuracy on the MathQA-Python dataset. This inspired us to use the AI21 Jurassic 2 LLM for code generation, given the extracted mathematical equations, to detect hallucinations.

Chapter Three: Methods

3.1 Data Collection and Manipulation

We used the Code Search Net dataset to obtain samples to test our evaluation metrics on and gain insight on the performance of the CodeGen2.5 model. For the purpose of our evaluation we filtered only a subset of the Python examples from this dataset, passing the function name and docstring of a single data point as an input into CodeGen2.5 and comparing the generated output with the reference function (ground truth). Additionally, we perturbed the input as described below, inserting, deleting, or substituting tokens and generating a new, perturbed output. For each data point we compared the perturbed output with the reference function as a measure of accuracy, and also against the originally generated function (baseline) as a measure of robustness. This was measured by both the normalized Levenshtein edit distance as well as CodeBERT token similarity score for each of the aforementioned accuracy and robustness cases, and the values collected for each data point we sampled were averaged per perturbation and plotted in the figures in Section 3.3 Results [2, 4].

3.1.1 Dataset: Code Search Net

The Code Search Net dataset contains 2 million code and docstring pairs scraped from open source repositories on GitHub for several programming languages (Go, Java, Javascript, PHP, Python, and Ruby [9]. Data points from this dataset consist of the code of a function, along with its associated documentation and metadata such as the repository it was extracted from and programming language. As described above, we sampled this dataset for data points to run our evaluation metrics on.

3.1.2 Model: CodeGen2.5

CodeGen2.5 is the latest iteration in Salesforce's CodeGen family of autoregressive language models that can be used for program synthesis [10]. The model itself builds upon CodeGen2 and is trained on BigCode's StarCoderData dataset, which contains over 750GB of code in 86 different programming languages – an equivalent of approximately 250 billion tokens [11]. Salesforce primarily uses this model in the form of a Visual Studio Code extension internally to assist developers in writing code for a large project with complex data structures; the extension tool can take context from other open files as well as information from the currently active file as input to generate code autocompletions. More information on the specifics of this model and its architecture can be found in Chapter 2: Literature Review.

Quantized Model

Due to the computational resources required to run Salesforce's CodeGen2.5 engine we chose to use a weight-quantized version of the model that was efficiently compressed and could run in a Google Colaboratory T4 GPU instance while still being able to perform very close to the level of the uncompressed CodeGen2.5 model [12]. Briefly, quantization is a technique that can be used in machine learning applications that reduces the number of bits required to represent a floating-point number by converting a high-precision floating-point value to a lower-precision floating-point value or even an integer. This helps to reduce both the computational cost and memory loads of model inferencing, reducing the total size of the model and increasing the speed of computation. More information about the specific model we used can be found on its associated HuggingFace page [12].

3.1.3 Perturbation

To investigate the robustness of CodeGen2.5, we perturbed the inputs before passing them into the model. Then, we observed how the perturbations changed the similarity of the output code generated by CodeGen2.5 by comparing the CodeBERT embeddings of the CodeGen2.5 output generated by a perturbed input to the CodeBERT embeddings of the originally generated, unperturbed, output as well as the CodeBERT embeddings of the ground truth reference code [2].

To perturb the input, we first tokenized the input to CodeGen2.5. At a high level, this process converts a string into a list of tokens (which correspond to substrings of the total input string), then further converts the list of tokens into a list of `token_id`s that correspond to each particular substring token. We then passed this token list into one of three different functions that we created for the purpose of perturbing the tokenized input: `deletion_mask`, `insertion_mask`, and `substitution_mask`. For each of these functions, we passed in as arguments the tokenized input and a degree of perturbation, which is the number of tokens that would then be perturbed by each function. The functions are described below:

- **`deletion_mask():`** An index number is selected anywhere at random from 0 to the length of the tokenized input. The token_id at the sampled index is popped from the list of tokens. We repeat this as many times as the perturbation degree.

- **`insertion_mask():`** An index number is selected at random anywhere from 0 to the length of the tokenized input. An insertion value is selected at random anywhere from the minimum to the maximum range of the token_id. We then insert the insertion value to the token list at the sampled index. We repeat this as many times as the perturbation degree.

- **`substitution_mask():`** An index number is selected at random anywhere from 0 to the length of the tokenized input. A substitution value is selected at random anywhere from the minimum to the maximum range of the token_id. We replace the token at the selected index with the substitution value. We repeat this as many times as the perturbation degree.

The perturbed inputs were then passed into CodeGen2.5, and the resultant outputs for each code sample and degree of perturbation per type of perturbation were evaluated for both code accuracy and code robustness as described in the following sections.

3.2 Evaluation Metrics

3.2.1 Levenshtein Edit Distance

Levenshtein edit distance is a measure of similarity between two strings, defined as the minimum number of changes that is necessary to transform one string into another. This is used to evaluate the accuracy of generated output when compared against reference output, providing additional insight to simply evaluating against token similarity because the CodeBERT encoder itself might not be robust and generate “nonsensical” latent representations with small changes in input. This could lead to false negatives since a low similarity score (and subsequently a low robustness score) could be due to CodeBERT instead of the CodeGen2.5 output itself).

A change to the string is defined as an insertion, deletion, or substitution of a character in the string; each of these changes results in an increment to the total edit distance. For the purposes of our project, we normalized the edit distance by dividing the total score by the length of the reference output string. This way, the resulting data is independent of the length of the input into the CodeGen2.5 model and trends can be extracted from the data independent of the length of the reference code of each sample.

Using this normalization technique, an edit distance of 0.0 indicates that there are no changes in the generated output string with respect to the reference output string, and an edit distance of 1.0 is equivalent to requiring a change in every single character in the generated output a single time in order to match the reference output. It follows that an edit distance greater than 1.0 indicates additional modifications to the generated output beyond the length of the reference output, namely the insertion of new characters. An edit distance between 0.0 and 1.0, therefore, represents a generated output that contains characters matching some, but not all, of the characters in the reference output and is also of similar relative length to the reference output. The edit distance is calculated using the following equation:

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}\big(\operatorname{tail}(a), \operatorname{tail}(b)\big) & \text{if } a[0] = b[0], \\ 1 + \min \begin{cases} \operatorname{lev}\big(\operatorname{tail}(a), b\big) \\ \operatorname{lev}\big(a, \operatorname{tail}(b)\big) \\ \operatorname{lev}\big(\operatorname{tail}(a), \operatorname{tail}(b)\big) \end{cases} & \text{otherwise} \end{cases}$$

Figure 3.1: Levenshtein edit distance equation

To evaluate accuracy of the model, we measured the edit distance from the generated output to the ground truth output taken from the training dataset, and to evaluate robustness of the model, we measured the edit distance from the output generated by perturbed inputs to the generated output without any perturbation masks applied.

3.2.2 Similarity Scores

Together with edit distance, the similarity score is a metric used to evaluate accuracy and robustness. Generally speaking, how well an LLM performs can be understood better by having a thorough idea on how well the encoder encodes the input into latent vectors in embedding space (CodeBERTScore is an example of this) [4]. This understanding can be brought about by comparing the embedding spaces of the current tool that we are trying to benchmark against the state of the art LLMs currently in use which can perform similar operations given the same input. We can also have specific output reference code (a ground truth) corresponding to the given input and generate embeddings from this, comparing it to the embeddings of the generated code via methods such as pairwise or piecewise embedding.

We used an ensemble similarity score to judge the accuracy and robustness of CodeGen2.5. As with Levenshtein edit distance, to evaluate accuracy of the model, we measured the similarity of the embeddings of the generated output to the embeddings of the ground truth output taken from the training dataset. To evaluate robustness of the model, we measured the similarity of the embeddings of the output generated by perturbed inputs to the embeddings of the generated output without any perturbation masks applied.

To calculate a similarity score, we first encoded the input code strings using the CodeBERT tokenizer [2]. Just like the CodeGen2.5 encoder, this converts a string of code into a list of substring tokens, which is then converted into a list of `token_id`s that correspond to each substring token. The list of `token_id`s are then embedded into the latent representation space with the CodeBERT encoder. For our analysis, we extracted only the first element of the latent representation tensor. This element is equivalent to the code embedding relative to the

`<clstoken>` and will have the same dimensions regardless of the code string that was encoded, such that flattening or padding the tensors for the purposes of comparison is not necessary.

The final similarity score is the average of the cosine similarity, Pearson correlation, and Spearman rank correlation, which are explained in greater detail below.

Cosine Similarity

Cosine similarity measures similarity between two non-zero vectors defined in an inner product space by taking the dot product of both vectors. The cosine similarity is calculated using the following equation:

$$\text{cosine similarity} = S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2 \cdot \sum\limits_{i=1}^{n} B_i^2}},$$

Figure 3.2: Cosine similarity equation

Pearson Correlation

The Pearson correlation coefficient is a number between –1.0 and +1.0 that measures the strength and direction of the relationship between two variables. The Pearson correlation coefficient is calculated using the following equation:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

Figure 3.3: Pearson correlation equation

Spearman Rank

Spearman rank correlation indicates the strength and direction of association between two ranked variables by measuring the monotonicity of the relation between two variables. The Spearman rank correlation is calculated using the following equation:

$$r_s = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)},$$

Figure 3.4: Spearman rank correlation equation

3.3 Results

On the following pages are results gathered using the methods described above. They are a visualization of the accuracy and robustness performance of the CodeGen2.5 model as measured by our evaluation metrics of Levenshtein edit distance and ensemble similarity score over varying degrees of perturbation from 5 to 50. Each data point in the graph represents the average value of the output resultant evaluation metrics across every sample from the dataset that was passed into CodeGen2.5 as an input. For accuracy results, refer to Figures 3.1 and 3.3, which evaluate the model performance against the reference code function. For robustness results, refer to Figures 3.2 and 3.4, which evaluate the model performance against the initially generated code output from using the sampled function name and docstring as input.

Each type of perturbation is tracked and labeled with its respective action (the deletion label refers to a deletion perturbation, the insertion label refers to an insertion perturbation, and the substitution label refers to a substitution perturbation). The baseline curve in the accuracy graphs represents the output generated by an unperturbed input function and docstring.

Analysis of the following results is described in Chapter 4: Analysis.

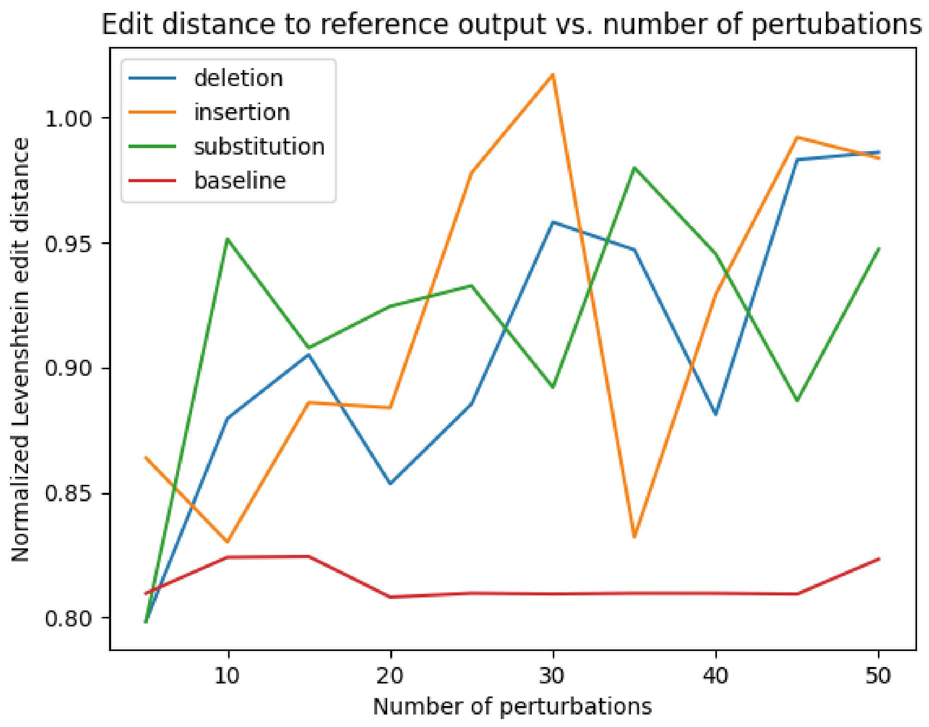

Figure 3.1: Accuracy with varying degrees of perturbation as measured by Levenshtein edit distance against the ground truth code

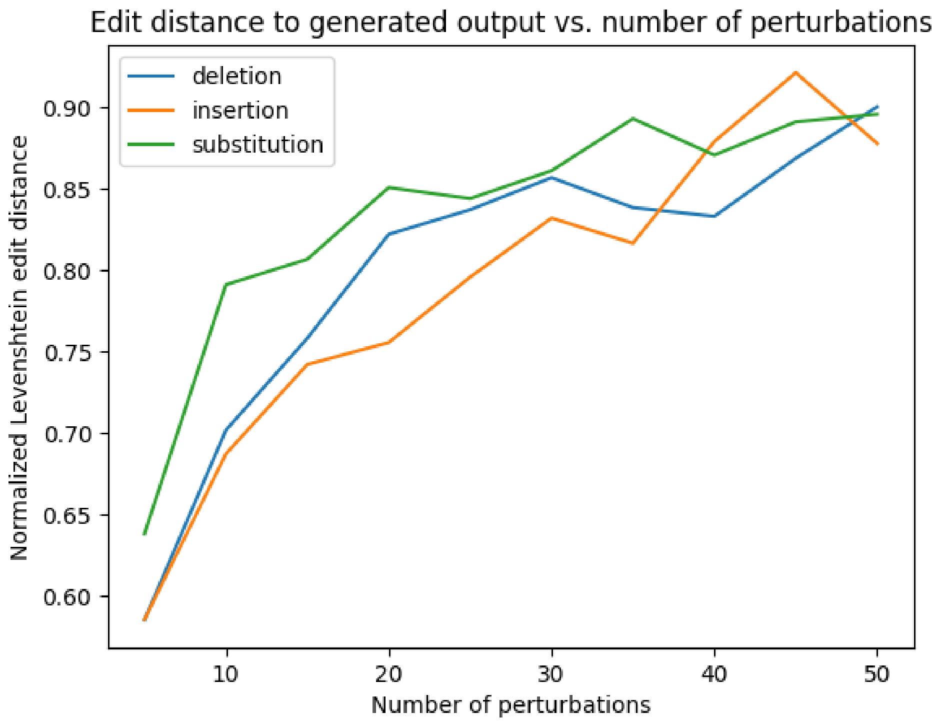

Figure 3.2: Robustness with varying degrees of perturbation as measured by Levenshtein edit distance against generated code

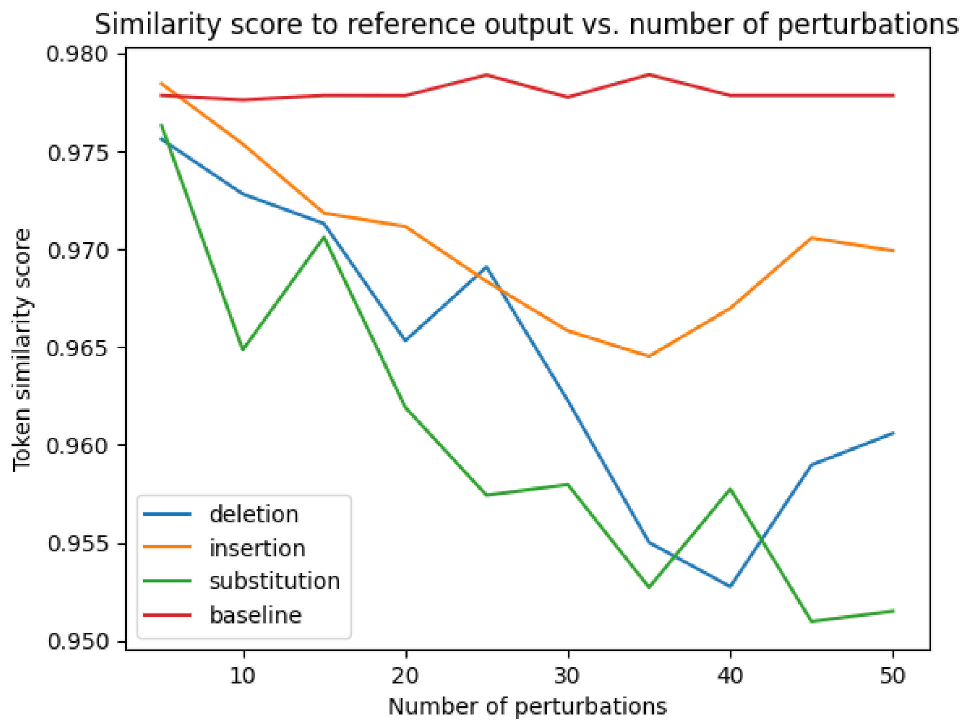

Figure 3.3: Accuracy with varying degrees of perturbation as measured by an average of Cosine, Pearson, and Spearman scores against the ground truth code

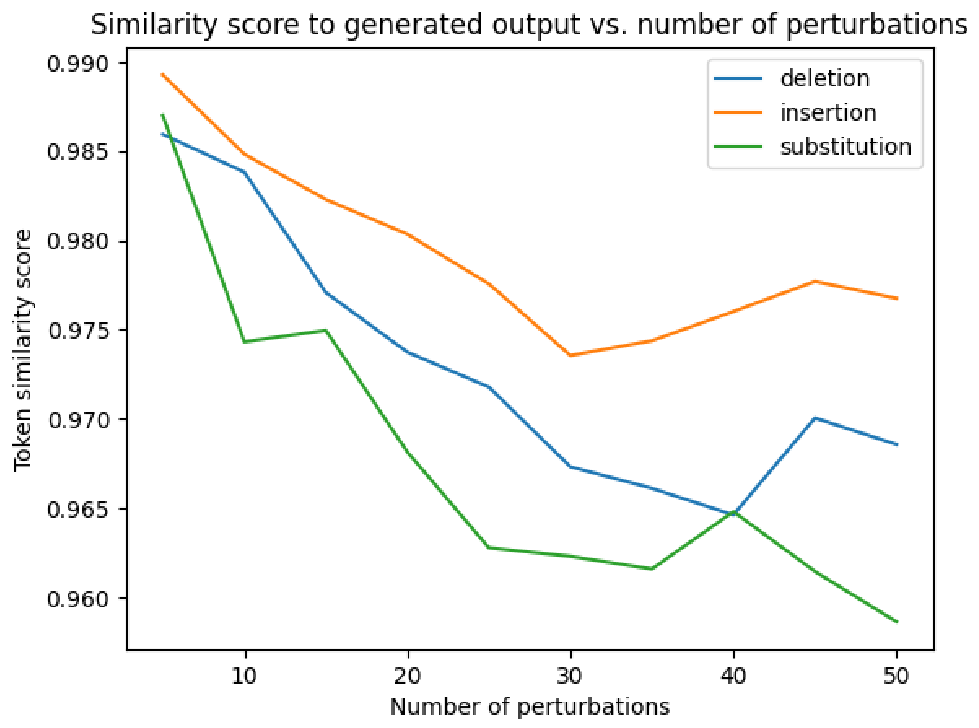

Figure 3.4: Robustness with varying degrees of perturbation as measured by an average of Cosine, Pearson, and Spearman scores against generated code

3.4 Mitigating Hallucinations

We attempted to mitigate hallucinations from CodeGen2.5 when generating code for mathematical equations by creating a fine-tuned dataset that gives concrete definitions for commonly used mathematical equations and perturbations of those definitions, following the stages outlined below.

3.4.1 Ground Truth Identification

To identify the hallucinations we used a ground truth table with four components to use as a concrete structure for code evaluation. Below is an example of the ground truth table:

Equation Name	Name & Number of Parameters	Parameter Values	Return Value
simple_interest	3 : P, N, R	10000, 3, 2.5	750

Table 3.1: Sample Ground Truth Table Entry

We manually created a list of 50 commonly used mathematical equations following the ground truth map, including 10 common and simple mathematical equations, such as simple and compound interest, and 40 equations with increasing complexity, such as quadratic equations and equations involving complex numbers. This results in a dataset that is diverse in terms of parameters and return values.

3.4.2 Code Formulation

The next step was to generate code that produced the correct return value in accordance with the ground truth. To this end, we used AI21 Jurassic 2 LLM [13] for code synthesis by passing prompts with the equation to the Jurassic 2 model. Then, we extracted the input and output variables using Abstract Syntax Trees, and used another prompt to ask the model to write a function to return the output given the input parameters.

The dataset contains correct functions that produce the desired results based on ground truth. For example:

```
def simple_interest(principal,time_period,rate_of_interest):
  simple_interest_calc_result = (principal * time_period * rate_of_interest) / 100
  return simple_interest_calc_result
```

3.4.3 Targeted Perturbations

For the generated functions, we generated variations using eight types of targeted perturbations while still maintaining the integrity of the operations performed in the function definition:

1. Variable name change
2. Data type change
3. Function name modification
4. Parameter order change
5. Parenthesis change
6. Return type change
7. Addition of comments
8. Addition and removal of print statements

These perturbations were accomplished by using Abstract Syntax Trees (ASTs) and Node Transformers [1]. We constructed the ASTs using the initial function definitions and performed the above perturbations using Node Transformers.

The transformation is done using:

```
class transformerClass(ast.NodeTransformer):
  def visit_arg(self, node):
    return ast.arg(**{**node.__dict__,
'arg':perturbation_method_<id>(node.arg)})
  def visit_Name(self, node):
    return ast.Name(**{**node.__dict__,
'id':perturbation_method_<id>(node.id)})
```

Here, `perturbation_method_<id>` represents all possible perturbation methods listed above, giving us all possible perturbations for the given equation.

This procedure is repeated for all 50 equations which in total gives a code sample size of 400 code snippets.

3.4.4 Results

We compared a selection of CodeGen models to see if they hallucinate, which are explained in greater detail in Appendix C: Model Descriptions. To understand the hallucinations, we ran the same completion prompt on all four models to see which ones hallucinate. For the purposes of this report, we only include one example prompt and the corresponding result per model:

```
text = "def simple_interest(p, n, r):" # the prompt
input_ids = tokenizer(text, return_tensors="pt").input_ids
generated_ids = model.generate(input_ids, max_length=256)
print(tokenizer.decode(generated_ids[0], skip_special_tokens=True))
```

codegen-2B-mono

```
def simple_interest(p, n, r):
    return p * (1 + r / n) ** n

def compound_interest(p, n, r):
    return p * (1 + r / n) ** n - p

def main():
    p = float(input("Enter principle amount: "))
    n = int(input("Enter number of times interest is compounded: "))
    r = float(input("Enter rate of interest: "))
    print(f"Simple interest: {simple_interest(p, n, r)}")
    print(f"Compound interest: {compound_interest(p, n, r)}")

if __name__ == "__main__":
    main()
```

codegen-2B-multi

```
def simple_interest(p, n, r):
    return p * n * (1 + r) ** -1

def simple_interest_with_decay(p, n, r, d):
    return p * n * (1 + r) ** -1 * (1 - d)

def simple_interest_with_decay_and_reset(p, n, r, d, s):
    return p * n * (1 + r) ** -1 * (1 - d) * (1 - s)

def simple_interest_with_reset(p, n, r, s):
    return p * n * (1 + r) ** -1 * (1 - s)

def simple_interest_with_reset_and_decay(p, n, r, s, d):
    return p * n * (1 + r) ** -1 * (1 - s) * (1 - d)

def simple_interest_with_reset_and_decay_and_reset(p, n, r, s, d, s2):
    return p * n * (1 +
```

codegen-2B-nl:

```
def simple_interest(p, n, r):
    return p * (1 + r) ** n

def simple_interest(p, n, r):
    return p * (1 + r) ** n

def simple_interest(p, n, r):
    return p * (1 + r) ** n

def simple_interest(p, n, r):
    return p * (1 + r) ** n

def simple_interest(p, n, r):
    return p * (1 + r) ** n

def simple_interest(p, n, r):
    return p * (1 + r) ** n

def simple_interest(p, n, r):
```

codegen2-1B:

```
def simple_interest():
    """
    This function will return a simple interest rate.
    """
    return 0.01

def simple_interest_rate():
    """
    This function will return a simple interest rate.
    """
    return 0.01

def simple_interest_rate_with_fixed_rate():
    """
    This function will return a simple interest rate with a fixed rate.
    """
    return 0.01

def simple_interest_rate_with_fixed_rate_and_fixed_rate_spread():
    """
    This function will return a simple interest rate with a fixed rate and fixed rate 
spread.
    """
    return 0.01

def 
simple_interest_rate_with_fixed_rate_and_fixed_rate_spread_and_fixed_rate_spread_spread():
    """
    This function will return a simple interest rate with a fixed rate and fixed rate spread 
and fixed rate spread spread.
    """
    return 0.01

def simple_interest_rate_with_fixed_rate_and_fixed_rate_spread_and_fixed_rate_spread_
```

Chapter Four: Analysis

4.1 Accuracy and Robustness

To reiterate, we evaluated the accuracy and robustness of CodeGen2.5 using the Levenshtein edit distance and our similarity score metric. Edit distance is a measure of how different the generated code is from the reference code based on the actual words present in the output; similarity score does the same but takes into account syntax and semantics. Tracking both metrics allows us to get a better insight into what could cause the CodeGen2.5 output to be less accurate or less robust. We should be concerned if the edit distance is high and the similarity score is low, since this indicates that the generated code is semantically different from the reference code.

4.1.1 Accuracy: Edit Distance

For the sampled inputs, CodeGen2.5 was relatively inaccurate according to the edit distance. As seen in Figure 3.1, the Levenshtein edit distance started at around 0.8 with 5 perturbations applied to the input and approached 1.0 when 50 perturbations were applied to the input. As the degree of perturbation increased, the accuracy as measured by Levenshtein edit distance continued to trend higher as well. This means that CodeGen2.5 generated code became increasingly different to the ground truth output as the input was perturbed further from the original sample, which is in line with intuition.

4.1.2 Accuracy: Similarity Score

As seen in Figure 3.3, the similarity score started at and remained around 0.950 to 0.980 for all types of perturbations. This indicates that CodeGen2.5 remains semantically correct despite a high edit distance. Further in contrast to the edit distance, the similarity scores fluctuated around a mean that did not change much with increasing degree of perturbation. An increasing edit distance but a relatively constant similarity score indicates that the syntax and semantics of the CodeGen2.5 output remain intact, even though the generated code became increasingly different. Notably, the general, albeit small, decrease in similarity score is likely because perturbing tokens changed the code context significantly, leading to an output that is logically, and thus semantically, different from that of the reference code.

4.1.3 Robustness: Edit Distance

According to the edit distance, CodeGen2.5 exhibits poor robustness. As seen in Figure 3.2, inserting, deleting, and substituting even only 5 tokens significantly altered the generated output, with an edit distance greater than 0.5 for all types of perturbations. Higher degrees of perturbations also led to higher edit distances. Regardless of whether the perturbation was an insertion, deletion, or substitution, the edit distance increased by around a total of 0.3 when the degree of perturbations was increased from 5 to 50 tokens. Although this represents a significant increase in the edit distance, increasing the number of perturbations made a smaller impact than the impact of initially introducing a perturbation to the CodeGen input, which makes logical sense since further perturbations to the initially perturbed input are contributing to an input that may be already semantically or contextually different from the original input string.

4.1.4 Robustness: Similarity Score

Similar to the results of edit distance, CodeGen2.5 exhibits poor robustness to insertion, deletion, and substitution when measured by similarity scores. As seen in Figure 3.4, for insertion, the similarity score started at approximately 0.990 when a token was randomly inserted into the CodeGen2.5 input, and decreased by a maximum of 0.015 as the degree of perturbation increased from 5 to 50. For deletion, the similarity score started at approximately 0.986 when tokens were randomly deleted from the CodeGen2.5 input, and decreased by a maximum of 0.020 as the degree of perturbations increased from 5 to 50. For substitution, the similarity score started at approximately 0.987 when tokens passed as input into CodeGen2.5 were randomly substituted for tokens of random value, and decreased by a total of 0.030 as the degree of perturbations increased from 5 to 50. Once again, increasing the number of perturbations made a smaller impact than initially introducing a perturbation to the CodeGen2.5 input, as evidenced by the negligible change in token similarity scores.

4.2 General Findings

4.2.1 Fluctuation in the Accuracy Graphs

For our accuracy graphs where we compared the generated output to the reference ground truth code, both the edit distance and the similarity score fluctuated at higher degrees of perturbation for all three perturbations: deletions, insertions, and substitutions. This makes sense since the context of the input has the potential to vary much more significantly as more random `token_id`s are inserted, deleted, or substituted. In other words, there is a possibility that insertions, deletions, or substitutions with random `token_id` could preserve the original meaning behind the context, but it is much more likely that these perturbations would lead to a completely different context of the input, and therefore the similarity between outputs generated by CodeGen2.5 is more volatile at higher degrees of perturbation.

4.2.2 Plateau in the Robustness Graphs

Both the edit distance and the similarity score approach a robustness plateau when the degree of perturbation is increased for all three perturbations: deletions, insertions, and substitutions. This indicates that after some degree of perturbation is reached, CodeGen2.5 starts generating the same output; it stops deviating from the ground truth code or the originally generated code. From this finding, it can be concluded that CodeGen2.5 is not robust to the first couple perturbations, but thereafter starts becoming more robust at higher degrees of perturbation.

4.3 Mitigating Hallucinations

While the CodeGen API works for a number of prompts, it struggles to generate correct code when asked to write functions to return results of some mathematical equations, such as functions to return roots of quadratic equations, calculate net present value, or find gravitational force using Newton's Law of Universal Gravitation. It also struggles with displaying graphical plots (specifically `seaborn` and `matplotlib`) and hallucinates when prompted to give different (uncommon) aliases to popular libraries, such as `numpy` and `pandas`. For example, when prompted to import numpy as ab, which is an atypical alias for numpy, the model still imported numpy as np, the common alias.

The observations and analysis below are based on input completion prompts into the various models for simple interest calculation as described in 3.4.4 Results:

codegen-2B-mono
The generated function formula: **`p * (1 + r / n) ** n`** is inaccurate and doesn't produce the results in the ground truth table. By definition, this is an inaccuracy as opposed to a hallucination.
codegen-2B-multi
This is a clear hallucination, as the model keeps building multiple functions by appending more context at the end of each function definition and adding to the body of the function.
codegen-2B-nl
The results observed here are similar to those for `codegen-2B-mono`; the model continues to generate tokens and repetitively creates multiple functions with the same function name. This is an example of both low accuracy and hallucination.
codegen2-1B
The results are similar to the output of `codegen-2B-multi`, as the model continues to build multiple functions by appending more context at the end of the function definition and building on the formula while returning a value of 0.1 on every function, which is a hallucination.

Chapter Five: Findings and Recommendations

5.1 Accuracy

Overall, our results show that CodeGen2.5 was relatively inaccurate for our method of sampling outputs, although it is semantically accurate as seen by a high similarity score. Specifically, providing the function name and the docstring seemed to be an insufficient amount of context required for CodeGen2.5 to generate accurate code. Perturbing the function name and docstring made CodeGen2.5 even more inaccurate although remaining semantically consistent, as evidenced by our results which show a generally increasing edit distance but a constant mean similarity score as the degree of perturbation increased.

Our recommendation is to more closely evaluate how Salesforce expects its users to use CodeGen2.5, particularly whether or not we should expect CodeGen2.5 to be able to generate code using just a function name and a docstring. If so, more work may need to be done to improve the accuracy of the model. However, if we only expect CodeGen2.5 to complete lines of code based on already existing code, and especially for known specific predetermined contexts as an internal utility as seems to be Salesforce's current use case for the code generation tool, this accuracy evaluation may be somewhat superfluous**.**

5.2 Robustness

Overall, our results show that CodeGen2.5 was not robust to small degrees of perturbation. Further, increasing the degree of perturbation caused the generated code to vary in robustness even more, albeit reaching a plateau at higher degrees. This makes CodeGen2.5 susceptible to adversarial attacks, such as random token deletion, insertion, and substitution. On this basis, our recommendation to Salesforce is to study the following question:

How can the architecture be made more robust to random perturbations in the input?

When designing the architecture of an LLM and training the model, developers should be equally concerned with robustness as they are concerned with accuracy. The method we developed in this project for evaluating the robustness of CodeGen2.5 can be extended to the domain of LLMs in general to address this issue. Particularly, one way to increase robustness could be to create a perturbation of the training dataset, so that the LLM is robust to adversarial attacks in the form of random noise to the input. The training dataset could be perturbed in a variety of ways, including changing the variable names, changing the function names, adding redundant lines of code, and reducing or increasing the length of code, without changing the logic of the input code. Another way to increase robustness could be to develop a new regularization loss for training the LLM, where the similarity score is used to measure the magnitude of the output deviation relative to the magnitude of the input deviation. Minimizing this regularization loss would optimize the LLM weights so that the model remains robust to small perturbations in the input.

5.3 Mitigating Hallucinations

The CodeGen API struggled to return results of common mathematical equations, display plots, and import libraries with uncommon aliases. Trying to improve the API's performance, namely generating accurate code using prompts, especially for equations, proved to be a challenging task. We tried a number of approaches, but each of them had limitations and could not guarantee accurate code generation when given a formula name as input.

5.3.1 Unexpected Insight

The below input prompt for all of the tested models:

```
inputs = tokenizer("# this function calculates simple interest",
return_tensors="pt").to(0) # prompt
sample = model.generate(**inputs, max_length=128)
print(tokenizer.decode(sample[0], truncate_before_pattern=[r"\n\n^#", "^'''",
"\n\n\n"]))
```

Actually generates the factually correct function:

```
# this function calculates simple interest
# input: principal amount, rate of interest, time
# output: simple interest
def simple_interest(principal, rate, time):
    return principal * (rate / 100) * time
```

While this is a pleasant finding, in order for the model to be as usable as possible for a broad audience it should be able to perform reasonably well on completion prompts using function definitions as input along with performing well on comments as input. By fine tuning the model with the generated dataset which has perturbed function definitions, we hope to reduce hallucinations and obtain factually correct results like the above from comment based prompts. Furthermore, we hope to extend this beyond the subdomain of mathematical equations to any code that the user wishes to generate using function definitions only.

5.4 Limitations

The primary limitation of our project is the unsupported linkage between code token embeddings and actual functionality of the code output. Although we observed similar scoring trends between the edit distance and similarity score with respect to number of perturbations, there still remains uncertainty over the reliability of this correlation. There hasn't been extensive research done on how changes to the embeddings reflect in changes to the output code [2, 6].

There is also possible bias with the limitations of our study to only Python code files. Under this limitation, we do not know how accurate our findings would be when applied to different coding languages, which have different syntax and semantics. In addition, the dataset used to train CodeGen2.5 is limited only to code repositories found on GitHub. This introduces some level of bias as well, since the data is only being collected from a single source, even though the source is diverse in use cases.

Another limitation we faced was more fundamental. As mentioned in the paper referenced in Chapter 2.6 that analyzes the interpretability of existing pre-trained code models, there is currently a gap in the understanding of the relationship between token code embeddings and the code token output. Because of this fundamental issue, we should be cautious about the reliability of the results of evaluating the similarity between token embeddings.

5.4 Future Work

From our discussions, more work can be done to improve the reliability and detail of our analysis and findings. These center around our methods of perturbing the CodeGen2.5 input and calculating the similarity score.

1. In our current analysis, we employed a stochastic method to perturb the CodeGen2.5 input. Specifically, the position and value of tokens we insert, delete, and substitute to the CodeGen2.5 input are selected at random. In the future, we could employ a deterministic, targeted method (non-random) to perturb the input to the CodeGen2.5 model. Below are the some examples of the scenarios in which we would implement this perturbation strategy:

    a. Perturbing import statements: For example, instead of "`import numpy as np`", we can change this to "`import numpy as ny`".
    b. Perturbing function names: For example, instead of "`def subtract(a, b):`", we can change this to "`def minus(a, b):`".
    c. Perturbing variable names: For example, instead of "`a = b + c`", we can change this to "`A = B + C`".

    We would then proceed with our accuracy and robustness analysis as before, using the Levenshtein edit distance and similarity scores. Doing this would yield more tangible results, where we can now evaluate how accurate and robust CodeGen2.5 is to known, non-random perturbations in input code strings.

2. As seen in our data analysis, the change in similarity score was negligible regardless of the degree of perturbation (on the order of magnitude -2). It seems strange that the semantic similarity of the generated output is greater than 0.95 despite all the perturbations made to the input. This does not provide us much insight, so in the future we could compare every single element of the latent representation tensor in our calculation. This would be accomplished by the following procedure:

   Since each code string has varying length, the number of elements in the `token_id` list and thus the dimensionality of the embedded representation tensor will vary for each code string. To be able to perform a cosine similarity between two embedded representations, we will need to ensure that both tensors have the same dimensionality. To do so, we will pad the end of the shorter `token_id` list with zeros to match the length of the longer `token_id` list. Doing so is equivalent to adding whitespace to the end of the shorter code string, which should not have an effect on the behavior of the code string.

   We would then use the encoder to embed the `token_id` lists; this should return an embedded representation of the same dimensionality while preserving the meaning, syntax, and semantics of the original code string. We then calculate the cosine similarity, Pearson correlation coefficient, and Spearman correlation rank between two whole embedded representation tensors as described in Section 3.2 Evaluation Metrics, without having to only use the tensor embedding at the position corresponding to the `<clstoken>`.

3. In order to improve the API and overcome hallucinations, we need to come up with better techniques to build a dataset or improve the ones we tried. We can invest in data preprocessing techniques to clean and normalize equations extracted from sources like Wikipedia or images, reducing irrelevant characters and improving formula accuracy. Image Extraction Refinement can be another technique where we invest in refining techniques for extracting equations from images. This could involve using images that are guaranteed to contain only the relevant formula. Fine tuning CodeGen with this dataset can reduce the hallucinations currently made by the model on this subdomain. Another approach can be to train the model on existing code datasets, consisting of samples where code is used to compute and return results of mathematical equations. Datasets such as CodeSearchNet, CONCODE, CoNaLa and LeetCode problem datasets can be considered for this task [14, 15, 16]. A similar process can be extended to all subdomains to reduce the hallucinations made in other subdomains of code problems. By addressing these recommendations, it's possible to bridge the gap between CodeGen API's current capabilities and the desired correctness and versatility.

References

[1] S. Ren et al., “CodeBLEU: A method for automatic evaluation of Code synthesis,” arXiv.org, https://arxiv.org/abs/2009.10297 (accessed Aug. 24, 2023).

[2] Z. Feng et al., “Codebert: A pre-trained model for programming and natural languages,” ACL Anthology, https://aclanthology.org/2020.findings-emnlp.139/ (accessed Aug. 24, 2023).

[3] A. Shirafuji et al., “Exploring the robustness of large language models for solving programming problems,” arXiv.org, https://arxiv.org/abs/2306.14583 (accessed Aug. 24, 2023).

[4] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, “CodeBERTScore: Evaluating code generation with pretrained models of code,” arXiv.org, https://arxiv.org/abs/2302.05527 (accessed Aug. 24, 2023).

[5] E. Nijkamp et al., “Codegen: An open large language model for code with multi-turn program synthesis,” arXiv.org, https://arxiv.org/abs/2203.13474 (accessed Aug. 24, 2023).

[6] Y. Wan et al., “What Do They Capture? - A Structural Analysis of Pre-Trained Language Models for Source Code,” https://arxiv.org/, https://arxiv.org/pdf/2202.06840.pdf (accessed Aug. 24, 2023).

[7] N. Varshney, W. Yao, H. Zhang, J. Chen, and D. Yu, “A stitch in time saves nine: Detecting and mitigating hallucinations of llms by validating low-confidence generation,” arXiv.org, https://arxiv.org/abs/2307.03987 (accessed Aug. 25, 2023).

[8] J. Austin et al., “Program synthesis with large language models,” arXiv.org, https://arxiv.org/abs/2108.07732 (accessed Aug. 25, 2023).

[9] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “CodeSearchNet Challenge: Evaluating the state of Semantic Code Search,” arXiv.org, https://arxiv.org/abs/1909.09436 (accessed Aug. 24, 2023).

[10] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, “Codegen2: Lessons for training llms on programming and natural languages,” arXiv.org, https://arxiv.org/abs/2305.02309 (accessed Aug. 24, 2023).

[11] R. Li et al., “Starcoder: May the source be with you!,” arXiv.org, https://arxiv.org/abs/2305.06161 (accessed Aug. 25, 2023).

[12] T. AI, “TheBloke/CODEGEN25-7B-mono-GPTQ · hugging face,” TheBloke/Codegen25-7B-mono-GPTQ · Hugging Face, https://huggingface.co/TheBloke/Codegen25-7B-mono-GPTQ (accessed Aug. 24, 2023).

[13] R. MacManus, “AI21 Labs releases jurassic-2, its new large language model,” The New Stack, https://thenewstack.io/ai21-labs-releases-jurassic-2-its-new-large-language-model/ (accessed Aug. 24, 2023).

[14] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “Mapping language to code in programmatic context,” ACL Anthology, https://aclanthology.org/D18-1192/ (accessed Aug. 24, 2023).

[15] P. Yin, E. Chen, B. Vasilescu, and G. Neubig, “The code/natural language challenge,” CoNaLa, https://conala-corpus.github.io/ (accessed Aug. 25, 2023).

[16] G. Christ, “Leetcode Problem Dataset,” Kaggle, https://www.kaggle.com/datasets/gzipchrist/leetcode-problem-dataset (accessed Aug. 25, 2023).

Appendix A: Source Code

Source code for the methods and results for this project can be found at the following links:

Accuracy and Robustness Evaluation:
https://colab.research.google.com/drive/1P6jasNorKbfSnmkwTZgySQ7eM18INPcE?usp=sharing

Hallucination Mitigation:
https://colab.research.google.com/drive/189QE7kZqFsOVRQoka34TQgANe3v1VxpB?usp=sharing

Creating Targeted Perturbations:
https://colab.research.google.com/drive/1DDWwG_Q1gs5bBJS02VSnwo9XdEs74DV8?usp=sharing

Appendix B: Challenges

While trying to develop a method to mitigate hallucinations for popular equations, we faced a number of challenges when trying to extract the ground truth equations and convert it to its corresponding ground truth code. Specifically, the problems arose when we tried to scrape Wikipedia for the formulas.

1. While scraping formulas from Wikipedia, challenges arose in extracting and formatting the equations due to unwanted characters and symbols (see below). This made it difficult to extract the equation from the scraped text. Hence we decided to come up with a better approach for extracting the formulas.

   Input:
   ```
   topic = wikipedia.page("Newton's second law of motion")
   equations =
   BeautifulSoup(topic.html()).find_all('annotation')
   equations[0].text.split('{\\displaystyle ')[1][:-1]
   ```

   Output:
   ```
   {\\textbf {F}}={\\frac {d}{dt}}(m{\\textbf {v}})
   ```

2. The formula extraction from image approach required the downloaded images to only comprise the formula for accurate formula extraction. However, in reality, this was almost never the case, as a result of which we got some unnecessary characters in the formula extracted, or incorrect/ invalid equations. For example, in the case of Newton's Second Law, The downloaded image consisted of **`ΣF=ma`**. The extracted text was >) **`F=ma`**. In order to get the correct equation,we had to remove the unnecessary characters from the equation. Since we were getting different extracted strings for different images, we had to write custom code for each sample to extract the correct equations from the extracted text.

As a result, we decided to use AI21 Jurassic-2 pre-trained LLM to generate code, given the formula name. While both the single prompt and multi prompt approach gave correct results for some equations (such as Newton's second law of motion and Newton's universal law of gravitation), they didn't work for all (such as roots of quadratic equations), despite trying multiple different prompts. We realized that it was difficult to come up with a generic prompt that gives correct code for all equation names.

Appendix C: Model Descriptions

`codegen-2B-mono`

The first model considered for hallucination was codegen-2B-mono, where "mono" means the model is initialized with CodeGen-Multi 2B and further pre-trained on a Python programming language dataset, "2B" refers to the number of trainable parameters and the model bin is about 5.69 GB large.

`codegen-2B-multi`

The second model considered for hallucination was codegen-2B-multi, where "multi" means the model is initialized with CodeGen-NL 2B and further pre-trained on a dataset of multiple programming languages, "2B" refers to the number of trainable parameters and the model bin is about 5.69 GB large.

`codegen-2B-nl`

The third model considered for hallucination was codegen-2B-nl, where "NL" means it is pre-trained on the Pile (Bigcode dataset), "2B" refers to the number of trainable parameters and the model bin is about 11.30 GB large.

`codegen2-1B`

The fourth model considered for hallucination was codegen2-1Bl, Unlike the original CodeGen model family (i.e. CodeGen1), CodeGen2 is capable of infilling, and supports more programming languages. It has 1 Billion parameters and the model bin is about 4.13 GB large.