William & Mary
CHARTERED 1693

W&M ScholarWorks

Undergraduate Honors Theses

Theses, Dissertations, & Master Projects

5-2024

Code Syntax Understanding in Large Language Models

Cole Granger
William & Mary

Follow this and additional works at: https://scholarworks.wm.edu/honorstheses

Part of the Artificial Intelligence and Robotics Commons, Data Science Commons, and the Programming Languages and Compilers Commons

Recommended Citation

Granger, Cole, "Code Syntax Understanding in Large Language Models" (2024). *Undergraduate Honors Theses.* William & Mary. Paper 2181.
https://scholarworks.wm.edu/honorstheses/2181

This Honors Thesis -- Open Access is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Undergraduate Honors Theses by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Code Syntax Understanding in Large Language Models

WILLIAM & MARY

CHARTERED 1693

Cole Granger

Department of Computer Science

The College of William and Mary

Supervisor

Denys Poshyvanyk, Ph.D.

In partial fulfillment of the requirements for the degree of

Bachelor of Arts in Computer Science

May 18, 2024

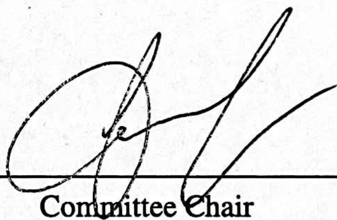
Code Syntax Understanding in Large Language Models

A thesis submitted in partial fulfillment of the requirement for the degree of Bachelor of Arts in Computer Science from William & Mary

by

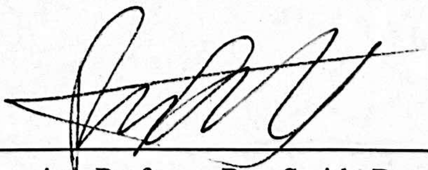
Cole Granger

Accepted for Honors



Committee Chair
Professor Denys Poshyvanyk, Computer Science
William & Mary

Professor Andreas Stathopolous, Computer Science
William & Mary



Associate Professor Ron Smith, Data Science
William & Mary

William & Mary
May 18th, 2024

Acknowledgements

I would like to express my deepest appreciation to my advisor, Professor Denys Poshyvanyk, for his invaluable belief in me. I could not have undertaken this journey without him first allowing me to work with his research group. I would also like to thank my committee members Professor Andreas Stathopoulos for his help in pursuing a computer science degree, and Professor Ron Smith, for his mentorship and wisdom.

This endevour would not have been possible without the guidance and mentorship of Dipin Khati. Your willingness to guide me through complex problems demonstrated a level of belief in my capabilities that was deeply motivating.

Special thanks should also go to all the members of the SEMERU group for their guidance and assistance on this project, as well as their mentorship during my time working under them as a research assistant.

Lastly, I would like to thank my family for all their support through the years. Their belief in me has kept my motivation high during this project.

In loving memory of Arthur Moore, whose love for learning inspired so many.

Abstract

In recent years, tasks for automated software engineering have been achieved using Large Language Models trained on source code, such as Seq2Seq, LSTM, GPT, T5, BART and BERT. The inherent textual nature of source code allows it to be represented as a sequence of subwords (or tokens), drawing parallels to prior work in NLP. Although these models have shown promising results according to established metrics (e.g., BLEU, CODEBLEU), there remains a deeper question about the extent of syntax knowledge they truly grasp when trained and fine-tuned for specific tasks.

To address this question, this thesis introduces a taxonomy of syntax errors, and a labeld set of LLM generated code containing syntax errors. The taxonomy was organized into Simple and Complex errors, describing the level of structual degredation caused by the syntax errors. We explored these over three different NLP datasets: Mostly Basic Python Problems (MBPP), the Code/Natural Language Challenge (CoNaLa), and HumanEval. With CoNaLa and MBPP having the task ofcode generation from natural langauge, and HumanEval having the task of code completion.

We ran a total of 4,941 prompts into the Mistral-7b-instruct-v.2 model, and encountered 130 syntax errors, or a 2.6% error rate. When we restirct the samples to python code only, the error rate increases to 2.9%. The most common simple error was an extra token, a space, added to the result. The most common complex error broke the assign relationship.

Contents

1 Introduction **1**

2 Background and Related Work **3**

2.1 History of Deep Learning 3

2.2 Large Language Models 6

2.2.1 Attention 7

2.2.2 Multi-head attention 8

2.3 Deep Learning for Software Engineering 8

3 Methodology **10**

3.1 Research Questions 10

3.2 Syntax Error Taxonomy Rational 10

3.2.1 Simple Errors 12

3.2.2 Complex Errors 13

3.3 Model and Datasets 15

3.3.1 Mostly Basic Python Problems 15

3.3.2 HumanEval 16

3.3.3 Code/Natural Language Challenge 16

3.4 Evaluation Setup 16

4 Results **18**

4.1 Cleaning and Validation 20

4.2 MBPP 23

4.2.1 Front Prompt Treatment 24

4.2.2 End Prompt Treatment 25

4.2.2.1 Statistical Significance 27

4.2.3 Samples of Interest and Removed Samples 28
4.3 CoNaLa 28
4.3.1 Rewritten 28
4.3.1.1 Train 28
4.3.1.2 Test 29
4.3.2 Intent 31
4.4 HumanEval 32

5 Discussion **34**
5.1 Main Findings 34
5.1.1 RQ1: Common Error Types 34
5.1.2 RQ2: Difficultly of Software Engineering Tasks 35
5.1.3 RQ3: Prompt Changes 35
5.1.4 RQ4: Task Differences 35
5.2 Research Implications 35
5.3 Study Limitations 36
5.4 Future Research 36

6 Conclusions **38**

References **39**

Chapter 1

Introduction

Many different means of automatic program synthesis have been explored to date. From foundational first order logic based systems, to state of the art natural language based models. In recent years, the field of Machine Learning for Software Engineering has emerged. [1] The emergence of large language models, [2] has resulted in increased research into natural language to source code generation tasks. These models have been relatively successful according to established metrics such as BLEU and CodeBLEU. [3], [4] Several models such as CodeBERT [5] , CodeLlama [6], PyMT5 [7], and other models have proven effective for transfer learning across several software engineering tasks. [1], [8] Nonetheless, despite promising results from those metrics, there remains a deeper question of the extent of code syntax knowledge these models truly grasp when trained and fine-tuned for software engineering tasks. Understanding how these models understand syntax could help provide another method for causal analysis of neural code models. [9] In this research, we seek to systematically evaluate the propensity of syntax errors across those tasks.

In order to answer these questions, in this paper, we introduce a curated study of different syntax errors encountered by Large Language Models across two software engineering tasks, natural language to source code, and code completion. In order to facilitate syntax error understanding, we created a taxonomy of syntax errors. This taxonomy consisted of two major categories, Simple and Complex errors, with several subcategories each. Simple errors are defined as misspellings, or single token mistakes.

Complex errors are defined as keyword misuse, multiple consecutive errors, or breaking a syntactic relationship present in python code syntax. [10]

We tested a total of 4,941 samples over various software tasks and found an overall error rate of 2.6%, however if we limit the samples to python code only (4,442) our error rate increases to 2.9%. Our greatest contribution is the creation of a taxonomy of syntax errors, which serves as a basis for future research.

Chapter 2

Background and Related Work

We briefly discuss previous research exploring syntactical understanding in large language models that we seek to build off of with this study. Definitions, Introduce Topics, Previous work.

2.1 History of Deep Learning

The historical idea of "machine learning", or learning a function from some training data that can map new input data to output values, is at least 200 years old. [11]. The earliest form of which, the least squares linear regression, was published by Legendre in 1805 and Gauss in 1809. [12].

A more modern definition is: "Machine learning (ML) is a branch of artificial intelligence (AI) and computer science that focuses on the use of data and algorithms to enable AI to imitate the way that humans learn, gradually improving its accuracy." [13] According to IBM, a few of the most common machine learning algorithms are:

- Neural Networks: "Neural networks simulate the way the human brain works, with a huge number of linked processing nodes. Neural networks are good at recognizing patterns and play an important role in applications including natural language translation, image recognition, speech recognition, and image creation."[13]

- Linear Regression: "This algorithm is used to predict numerical values, based on a linear relationship between different values. For example, the

technique could be used to predict house prices based on historical data for the area."[13]

- Logisitic Regression: "This supervised learning algorithm makes predictions for categorical response variables, such as "yes/no" answers to questions. It can be used for applications such as classifying spam and quality control on a production line."[13]
- Clustering: "Using unsupervised learning, clustering algorithms can identify patterns in data so that it can be grouped. Computers can help data scientists by identifying differences between data items that humans have overlooked." [13]
- Decision trees: "Decision trees can be used for both predicting numerical values (regression) and classifying data into categories. Decision trees use a branching sequence of linked decisions that can be represented with a tree diagram. One of the advantages of decision trees is that they are easy to validate and audit, unlike the black box of the neural network." [13]
- Random forests: "In a random forest, the machine learning algorithm predicts a value or category by combining the results from a number of decision trees." [13]

This paper will focus on the subsection of machine learning that is "deep learning". This all began with Frank Rosenblatt's perceptron. [11], [14]. These perceptrons can adjust their output based on training data. If the intial output is incorrect, the weights can be adjusted. These were stacked to form Artifical Neural Networks. The intial version of these had many issues, including the inability to learn the XOR boolean function. [11], [15].

This continued until the introduction of the backpropagation algorithm. [11] By minimizing an error function, often called the *loss function*, this technique allowed for the use of gradient descent in order to iteratively update the weights in a multi-layer neural network. [16] The original perceptron proposed by Rosenblatt, used a step function such that it outputs a fixed value, either a 0 or a 1. [14]. This function can be represented as:

$$y = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

However, backpropagation requires that the output of the neural net be differentiable. The use of activation functions, such as the sigmoid, or a more modern function the ReLU [17], allow for the differentiation of the perceptron output. [16]

The following section is based largely on the online book written by Michael Nielsen. While the equations are simplified for this paper, he goes much more in depth on the specifics of the algorithm.

Let's consider a neural network with layers indexed by l. Each layer has weights $\mathbf{W}^{(l)}$ and biases $\mathbf{b}^{(l)}$. The output of each layer before activation is denoted as $\mathbf{z}^{(l)}$, and after applying the activation function σ, it is $\mathbf{a}^{(l)} = \sigma(\mathbf{z}^{(l)})$.

The process of backpropagation involves computing the gradients of a loss (sometimes called the cost) function L with respect to the weights and biases. In classification tasks, it is typically cross-entropy, and in regression it is typically squared error. These gradients are used to update the parameters during training.

$$\nabla L = (\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \cdots, \frac{\partial L}{\partial w_l})$$

The error at the layer (L) is given by:

$$\delta^{(L)} = \nabla_a L \odot \sigma'(\mathbf{z}^{(L)})$$

where $\odot$ denotes the element-wise product, and $\nabla_a L$ is the gradient of the loss function with respect to the activations of the output layer. This means that $\delta^{(L)}$ is a vector of length n, where n is the number of nodes at any given layer (L).

For any layer l, the error $\delta^{(l)}$ is propagated backwards from layer $l+1$:

$$\delta^{(l)} = \left((\mathbf{W}^{(l+1)})^\top \delta^{(l+1)}\right) \odot \sigma'(\mathbf{z}^{(l)})$$

This means that the previous layer's error can be computed with the current layer, hence *backpropagation.*

The gradient of the loss function with respect to the weights and biases are

computed as follows:

$$\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} (\mathbf{a}^{(l-1)})^\top$$
$$\frac{\partial L}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}$$

Using these gradients, the parameters are updated using gradient descent:

$$\mathbf{W}^{(l)} = \mathbf{W}^{(l)} - \lambda \frac{\partial L}{\partial \mathbf{W}^{(l)}}, \quad \mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \lambda \frac{\partial L}{\partial \mathbf{b}^{(l)}}$$

where λ is the learning rate. [16], [18], [19].

Hoowever, intial usage of this algorithm of early multi-layer neural networks had several flaws. The larger the model, the less effective backpropagation is for learning. This is because the gradient of the loss function is split between neurons. This split causes the problem of a vanishing gradient. [20]

This basic architecture was expanded to fufill several use cases, such as image classification, sequence predicition, clustering, and pattern recognition. [21]–[23].

The most important usage of neural networks relating to this work is sequence modeling. The first recurrent neural networks were introduced in 1982 by John Hopfield. [24]. A significant improvement came with the introduction of the LSTM, which was designed to avoid the vanishing gradient problem. [22].

These models were foundational in the later development of Large Language models.

2.2 Large Language Models

While the idea of natural language processing can be tracked as far back as Alan Turing and his Turing Test. [25] The first major "language model" was the release of the n-gram model, which modeled language as a Markov process of n-word long sequences [26]. The aforementioned recurrent models dominated the natural language domain, with the introduction of the GRU used for statistical translation [27], and the Seq2Seq model. [28]

The release of the "Attention is all you Need" paper introducing the transformer architecture [2] is the foundation of large language models. It marks the

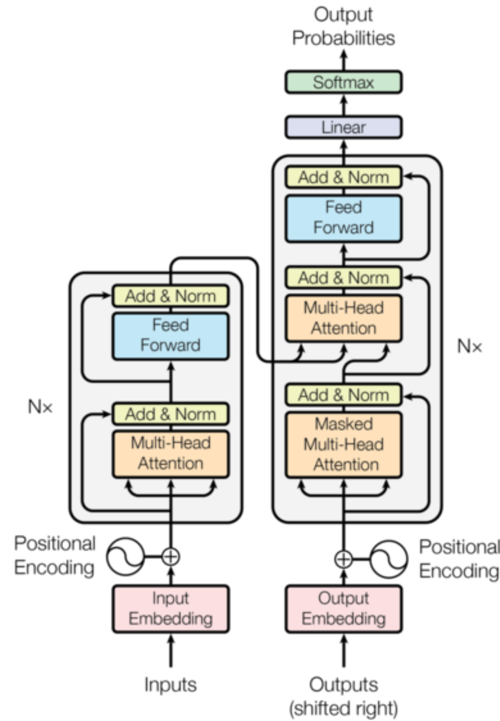


Figure 2.1 Transformer Architecture as proposed in "Attention is All You Need" by Vaswani et al.

shift away from recurrent based models to models solely focused on an attention mechanism.

2.2.1 Attention

Self-Attention, at a high level, is a method to compute an ouput by considering interactions between all words in a sentence, regardless of their position. The impact of each word on the output is determined by the strength of its relevance (attention) with other words. This allows the model to focus on different sections of a sentence. [29]

It is described as "An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key" in the original paper.

Self-attention is calculated by computing the dot product of the query with all keys, dividing by $\sqrt{d_k}$, then obtaining the ranked weights of the corrisponding values with a softmax function.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.1}$$

[2]

2.2.2 Multi-head attention

The multi-head attention function is described as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O \tag{2.2}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.3}$$

Multi-Head Attention involves running the attention mechanism multiple times in parallel. Each of these parallel operations is known as a "head." For each head, the input matrices Q (queries), K (keys), and V (values) are linearly transformed with different, learned linear projections. This creates different sets of queries, keys, and values for each head. Attention is then computed independently on each set of queries, keys, and values, allowing the model to capture different aspects of the information. The outputs of each head are concatenated and once again linearly transformed to produce the final values, which combines insights from each head.

2.3 Deep Learning for Software Engineering

Program generation has been a research goal of artificial intelligence as far back the 1950's. [30]–[33] This evolved from algorithmically transforming mathematical equations into machine code in the 1960's, to rule-based systems such as C Language Integrated Production System (CLIPS), then to UML diagram to code programs such as IBM Rational Rose in the 1990's. [34] With the advancement of computational resources, a field of Deep Learning for Software Engineering has emerged (DL4SE). The emergence of large language models and their subsequent application on code has revolutionized this field. [35], [36]

In recent years, it has become increasingly popular to train large language

models on source code. [5]–[7] Which have shown promising results in various software engineering tasks. [37] LLMs have increasingly been employed to generate source code, offering significant advancements in programming assistance, code completion, and even generating entire scripts or software modules. These models are trained on a diverse range of internet text, including substantial volumes of programming code across various languages, which enables them to understand and generate code effectively. Oftentimes, the datasets used to train these models can range from billions, to even trillions of tokens, pulled from public repositories such as GitHub.

Various ways of evaluating these models have been explored to various degrees of success. [1], [3], [9], [38]

Chapter 3

Methodology

3.1 Research Questions

To try and understand LLM syntax understanding we propose a series of research questions:

- RQ_1**:** What types of errors are most common in LLM generated syntax errors?
- RQ_2**:** Is there a specific software engineering task that causes more errors?
- RQ_3**:** How do prompt changes affect the amount of syntax errors?
- RQ_4**:** Is there any difference in syntax error frequency and type based on task type?

3.2 Syntax Error Taxonomy Rational

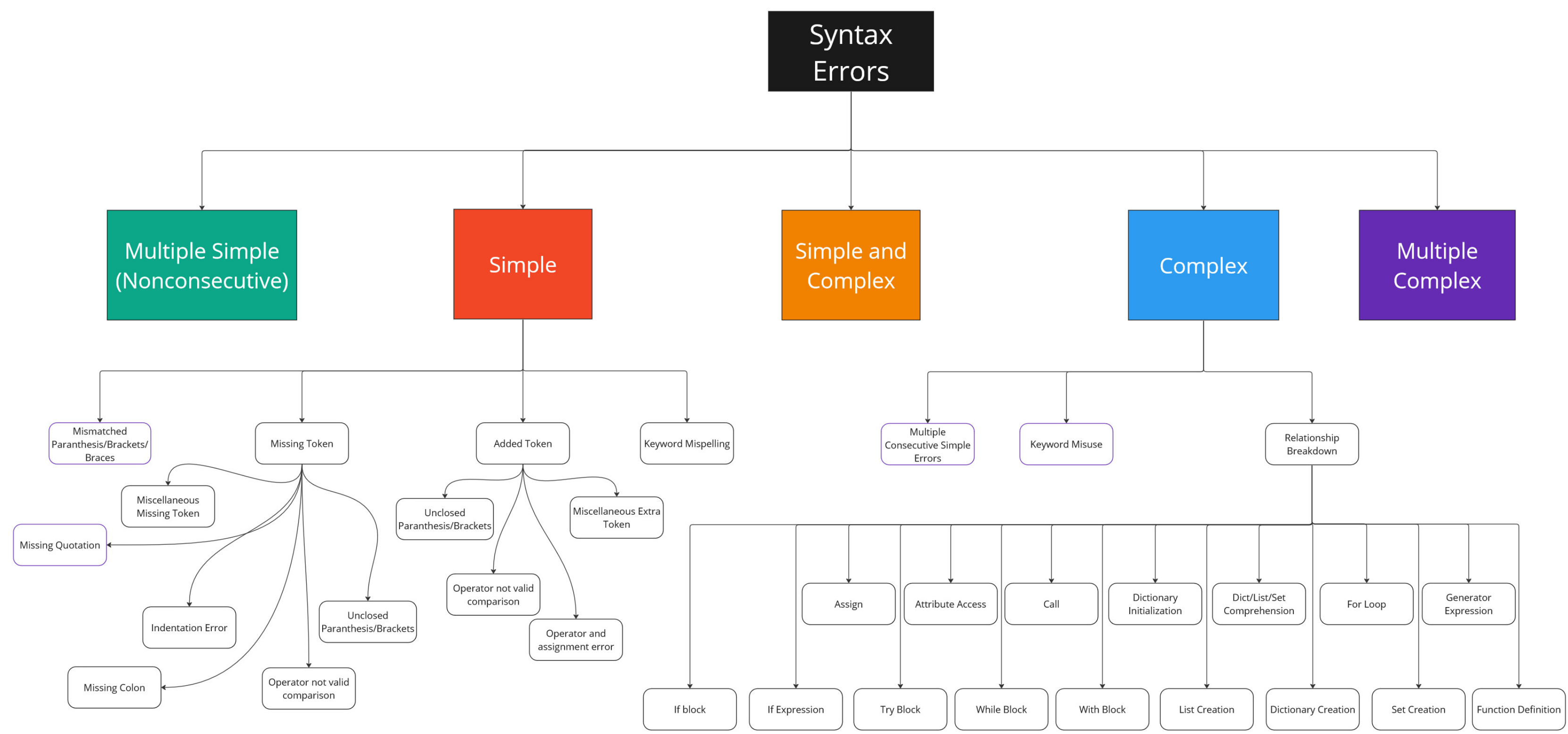


Figure 3.1 Proposed Syntax Error Taxonomy

In order to study the level of code syntax understanding of large language models, we need a classification metric for syntax error type and severity.

To this end, we propose a taxonomy of syntax errors. As seen in Figure-3.1 syntax errors are broken down into 5 main categories, Multiple Nonconsecutive Simple, Simple, Simple and Complex, Complex, and Multiple Complex. Three of these categories, Multiple Nonconsecutive Simple, Simple and Complex, and Multiple Complex, are limited to single lines only. This means that a sample can only be one of these classes, if multiple errors occur in the same line. If multiple errors occur within the same sample, but occur on different lines, each error would be labeled Simple or Complex. We design the taxonomy this way to draw a clearer line between potentially random token addition or removal, and a more severe lack of syntax understanding.

3.2.1 Simple Errors

We define simple errors as errors pertaining to a single token only, or a keyword misspelling. A few examples are:

- ```
  def geometirc_series_nth_term(first_ term, common_ratio, n):
      ...
      return first_term * (common_ratio ** (n - 1))

  ```

  Listing 3.1: Extra Token

- ```
      merged_iterator = iter(()))

  ```

  Listing 3.2: Unclosed Paranthesis/Extra Token

- ```


      for i in range(1 << n):
          s = bin(i)[2:]
          count[i] = int(sum([int(b) for b in list(zip(*[list(
  s)[::-1], list(s)])[0]))])

  ```

  Listing 3.3: Mistmatched Paranthesis
  ```

If two or more simple error occur non-consecutively in a sample, we define it as the class Multiple Simple Errors. However, If two or more syntax errors occur in succession, it is then classified as a complex error. Ex:

- ```
  def binomial_probability(n, k, p):
  ...
      n choose k = math.comb(n, k)
      return n choose k * pow(p, k) * pow(1.0 - p, n - k)
  
  ```

Listing 3.4: Multiple Simple Errors

- ```
      my_list.sort(key=lambda x: int(x.split(' (' ')[0].split(' ')[-1]))
  
  ```

Listing 3.5: Complex(Multiple Consecutive Simple)

3.2.2 Complex Errors

Complex errors are based upon syntactical relationships present in python code. [10] We are not tracking all relations or sub-relations from Shen et al. but we have taken a sample of the relation types listed in that paper to create a basis for our complex error types. Along with those relationship types, We added Multiple Consecutive Simple as mentioned above, and Keyword misuse.

The reasoning for including 'Multiple Consecutive Simple' as a complex error is to differentiate it from single token errors. Simple errors are based on singular tokens, so have two consecutive single token errors naturally removes it from this classification. Therefore, instead of creating another separate category, we elect to include it within complex errors. The presence of multiple consecutive single token errors also shows a more severe lack of understanding.

Relation	Example
Assign: Target $\rightarrow$ value	target = 10
Call: func $\rightarrow$ arg	function(args)
For: for $\rightarrow$ body	for target in iter: body

if:
if $\rightarrow$ else
	if condition:
	body1
	else:
	body2

if (conditional) expression:	target = value if bool else value
Dictionary Creation	{key: value}
List Creation	[value, value, value]
Set Creation	(value, value, value)
Dictionary/Set/List Comprehension	[for item in lst] {for key in dict} {for item in set}
Generator Expression	(value for idx in iter)

Try Block	try:
	body1
	except Exception:
	body2
	else:
	body3
	finally:
	body4

With Block	with open('file.txt') as f: content(f)
While: test $\rightarrow$ body	while condition: body
Function Definition	def func(arg1, arg2): body

Table 3.1: Relation Types Defined in our Taxonomy

3.3 Model and Datasets

We chose to use the Mistral-7b-Instruct-v2.0 for our testing. [39]

Model	Modality	MMLU	HellaSwag	WinoG	PIQA	Arc-e	Arc-c	NQ	TriviaQA	HumanEval	MBPP	MATH	GSM8K
LLaMA 2 7B	Pretrained	44.4%	77.1%	69.5%	77.9%	68.7%	43.2%	24.7%	63.8%	11.6%	26.1%	3.9%	16.0%
LLaMA 2 13B	Pretrained	55.6%	**80.7%**	72.9%	80.8%	75.2%	48.8%	**29.0%**	**69.6%**	18.9%	35.4%	6.0%	34.3%
Code-Llama 7B	Finetuned	36.9%	62.9%	62.3%	72.8%	59.4%	34.5%	11.0%	34.9%	**31.1%**	**52.5%**	5.2%	20.8%
Mistral 7B	Pretrained	**60.1%**	**81.3%**	**75.3%**	**83.0%**	**80.0%**	**55.5%**	**28.8%**	**69.9%**	**30.5%**	47.5%	**13.1%**	**52.2%**

Figure 3.2 Mistral 7b Eval from [39]

We were limited in the size of model we could run due to hardware requirements. Given the performance level comparable to 13B parameter models, it will provide a good basis model for testing.

We used three datasets to produce code samples.

- Mostly Basic Python Problems (MBPP)
- HumanEval
- Code/Natural Language Challenge (CoNaLa)

3.3.1 Mostly Basic Python Problems

Mostly Basic Python Problems was introduced in 2021 by a team at Google Research. It consists of 974 python problems designed to be solved by beginner level programmers, covering programming fundamentals, standard library functionality, and several other basic programming paradigms. [1] In our testing, we used the test set exclusively, which consists of 500 samples. We did not include the test cases in the prompt, as we are not testing code functional correctness just syntax. This dataset was used for the task of code generation from natural language. It represents expert level prompt creation, because of it's creation by programmers at Google.

We extended this dataset with a basic test for prompt engineering. We added the phrase 'Generate code only.' to either the front or the end of the prompt. For example, given a prompt in the MBPP dataset:

'Write a python function to find the missing number in a sorted array.'

We add the phrase 'Generate code only.' to the front:

'Generate code only. Write a python function to find the missing number in a sorted array.'

Or to the end:

'Write a python function to find the missing number in a sorted array. Generate code only.'

3.3.2 HumanEval

HumanEval is a dataset released alongside Codex, a GPT model trained on GitHub source code. It contains 164 hand-written samples containing a function signature, docstring, body, and several unit tests. [40] In our testing, we use the function signature and docstring only. We do not use HumanEval for its intended evaluation purpose, but rather as a basis from which to generate code samples from unseen docstrings. We do not use the provided test cases or reference code in the original dataset. Instead, we pass the provided function definition and docstring through our model to act as the code completion task. Although, we cannot verify that HumanEval was not included in the training set of our model.

3.3.3 Code/Natural Language Challenge

The Code/Natural Language Challenge is a dataset compiled in 2018 crawled from Stack Overflow. It includes 2,739 train and 500 test samples in it's curated version. [41] We use both the train and test datasets in our testing as a basis for generating code samples. We used both the intent, the original natural language crawled from stack overflow, and the rewritten intent, a crowd-sourced rewritten version of the original intent. We only used the test set of the original intent prompt.

3.4 Evaluation Setup

In order to extract syntax errors from Python code we use an Abstract Syntax Tree. This works because programming languages follow PEGs or a parsing expression grammar. These are similar to context-free grammars, in that it can break down a language into a parse tree. The main difference between a CFG and

a PEG, is that a PEG has exactly **one** correct parse tree, or none. This means that by parsing the code, we can detect syntax errors according to the grammar.

The limitations of normal Python syntax error detection present in most common IDEs is that it will stop parsing on the first syntax error encountered. In order to detect all present syntax errors in our generated samples, we created a simple Abstract Syntax Tree parser using the tree-sitter library. The reason we chose the py-tree sitter library is because it has the functionality to continue parsing even after encountering a syntax error. It reports whenever there is an error node in the tree, and the relevant string, line and character numbers, as well as the number of child nodes the error node has.

For each dataset, and prompt engineered version, we generated a subsequent response from our model. We then extracted all code from the responses, and verified the language. This extracted Python code was then passed into our AST parser, and the errors were reported.

Then we removed all errors that contained obvious console output, in the form of ' '. All other errors in the samples were maintained.

These were then saved as the 'raw errors'. These raw errors were then hand classifed based on the syntax taxonomy, and any errors that came about from text not intended to be code were removed. These hand-curated errors were called the 'true errors'.

Chapter 4

Results

Dataset	MBPP	MBPP-Front	MBPP-End	CoNaLa Rewritten Train/Test	CoNaLa Intent Test	HumanEval
Samples	500	500	500	**2777**	500	164
Python Samples	495	484	489	2335	475	N/A
Raw Error Samples	16	32	28	59	19	2
True Error Samples	14	29	19	**52**	12	1
Sum of Errors	18	36	27	68	13	1
Error Rate	3.64%	**5.8%**	3.8%	2.22%	2.4%	0.61%
Simple	14	25	21	28	8	0
Complex	4	3	6	40	5	1
Most Common Error Class	Simple	Simple	Simple	Complex	Simple	Complex
Most Common Error Type	Indentation Error	Indentation Error	Extra Token, \	Assign	Extra Token, \	List Comprehension

Table 4.1: Basic summary of results, showing key data points, such as simple and complex error counts, most common error type, and error rates per dataset.

4.1 Cleaning and Validation

As mentioned in the methods section, after we generated our samples, they went through a phase of intial parsing and then cleaning. In order to pass the prompts into the model properly, an instruction template provided by Mistral was used. This included instruction tags: <s >and <\s \>as well as [INST] and [\INST]. [39] The remnants of this template was included in the model output, so it was removed. Then, any escaped newline or ’\’ characters were replaced with their unescaped versions. Then all code surrounded by ’ ` ` ` ’ or some variation of this pattern, was extracted. This is saved in the ’code’ column in the raw data. This code was then joined into a single string. This is saved in the ’all_code’ column. Then a respective json and csv file were saved as the ’raw errors’.

Then, all these raw errors were classified based on the proposed taxonomy, and any extraneous samples that were not meant to be executable code (such as console output, or example usage) were removed. The remaining errors were saved as the ’true errors’.

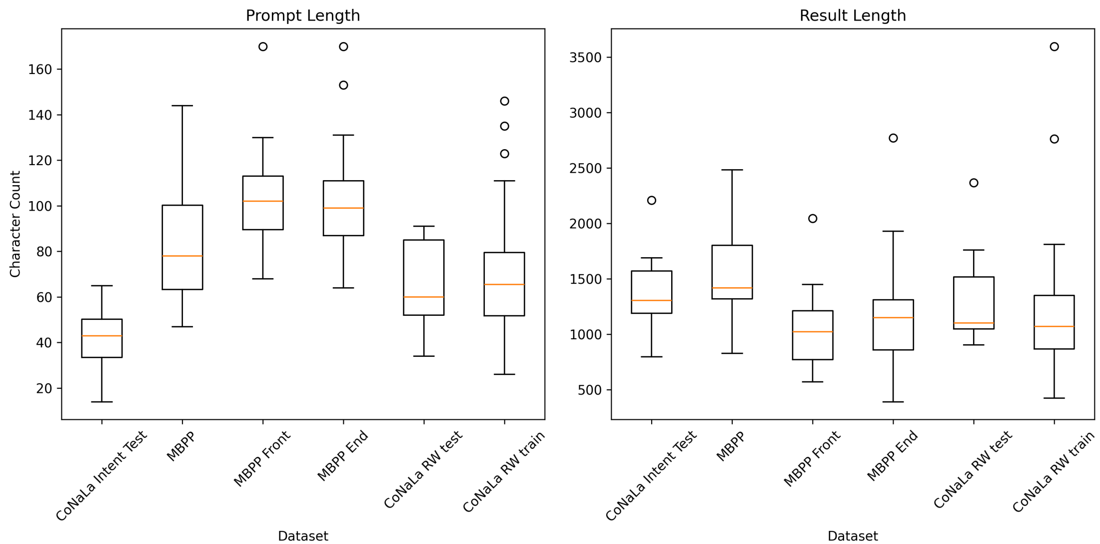


Figure 4.1 Prompt and Result length for samples that contained errors in each dataset.

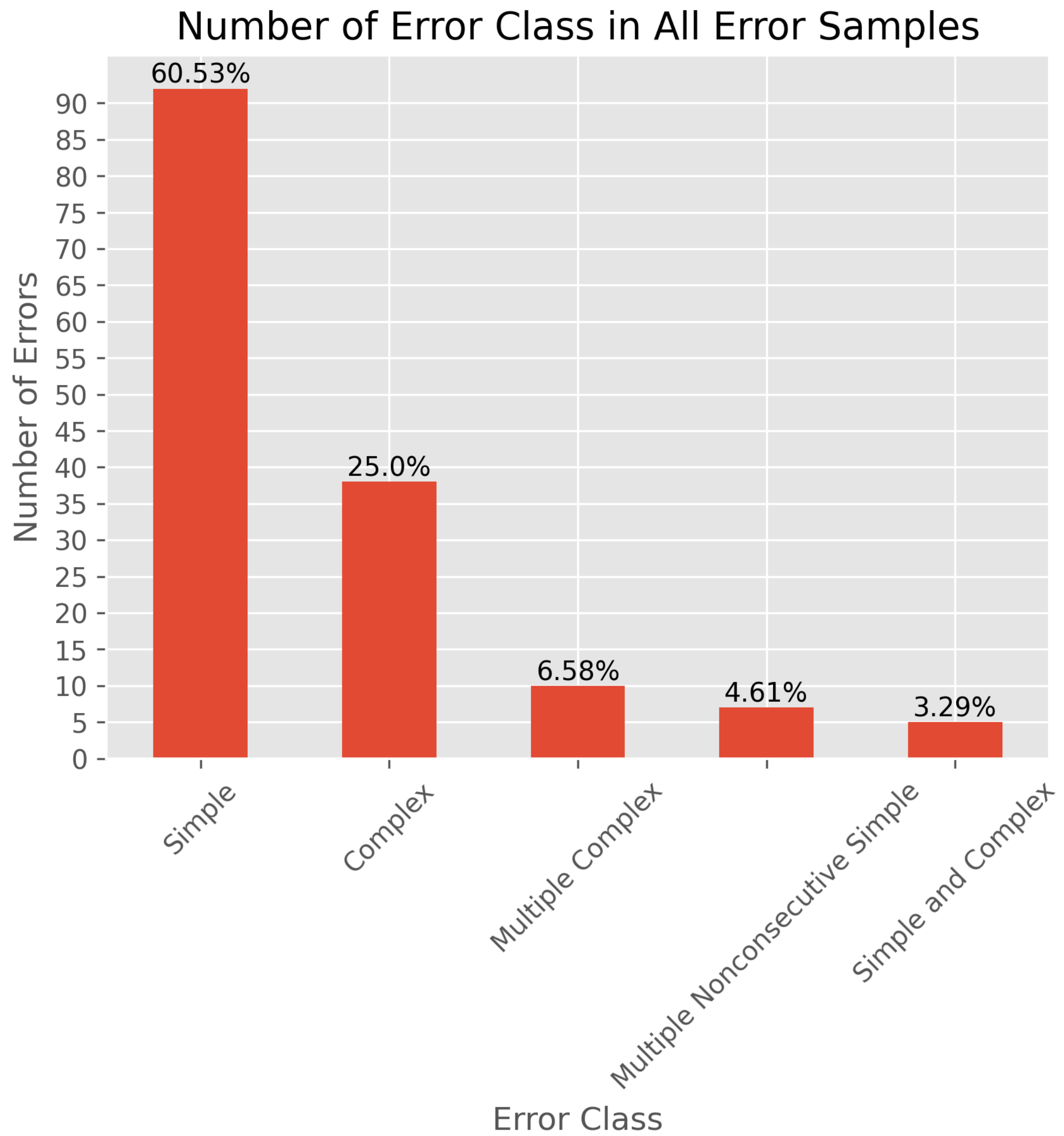


Figure 4.2 Error class count breakdown over all error samples.

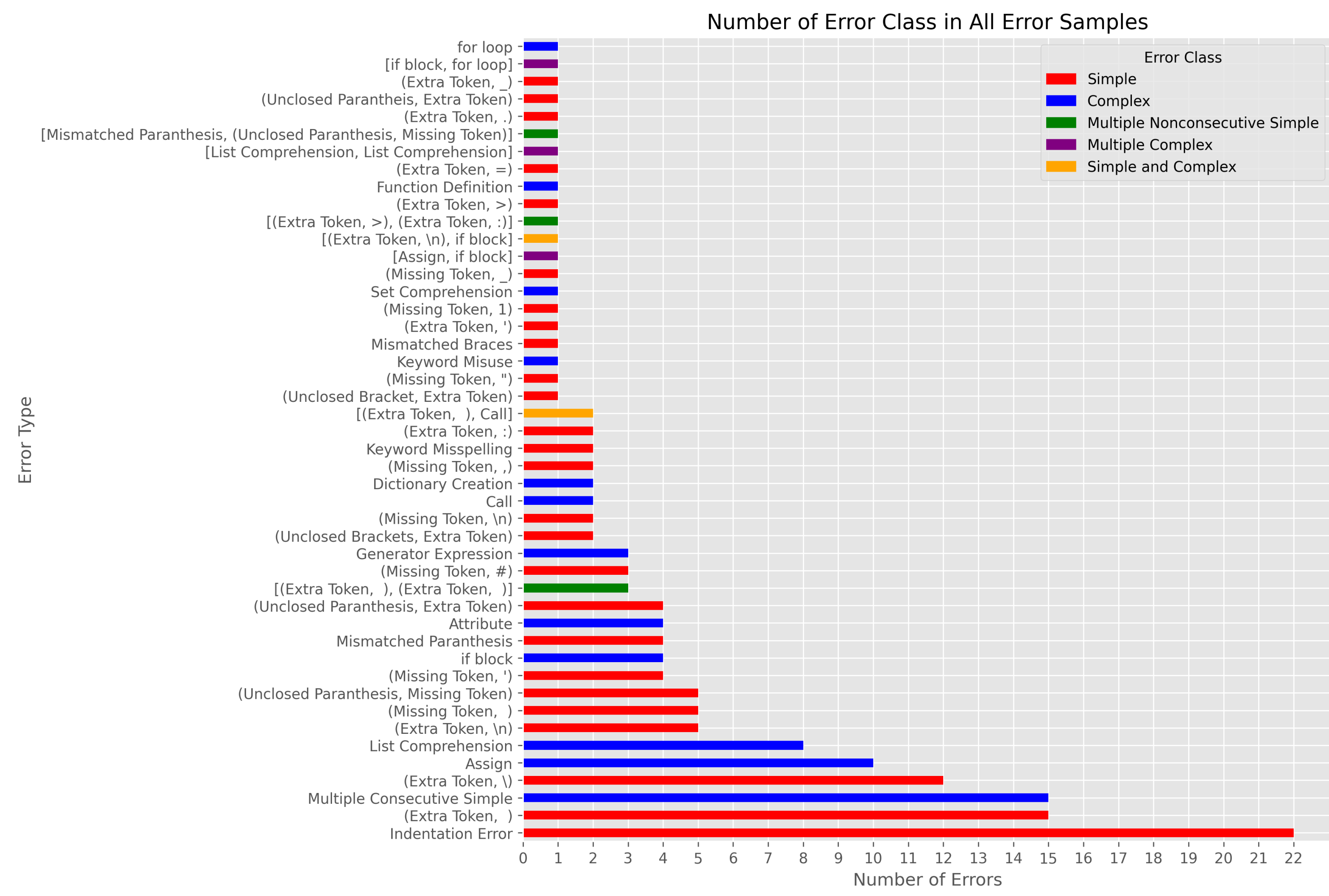


Figure 4.3 Error type breakdown over all error samples, color designated error class.

4.2 MBPP

Out of 500 samples generated, 495 were python code. There were a total of 16 raw error samples, after manual verification there were 14 true error samples. The total amount of errors, counting samples with multiple errors as multiple, was 18 with 12 Simple, 2 Complex, 1 Multiple Nonconsecutive Simple, and 1 Multiple Complex as shown in Figure 4.4. The most common error class was Simple, and the most common error type was Indentation Error as shown in Figure 4.5.

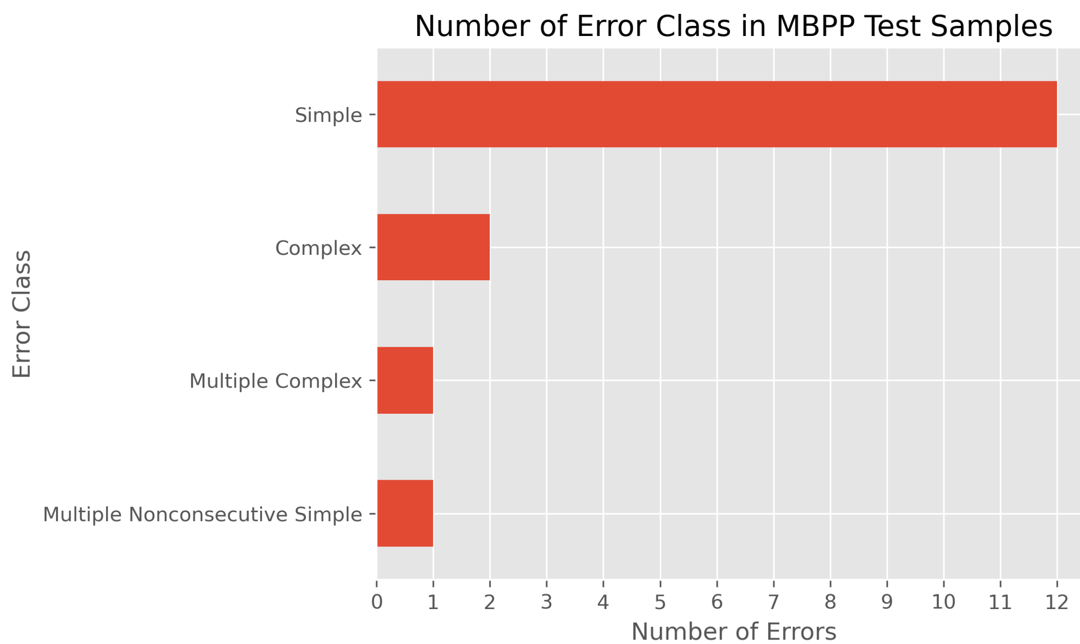


Figure 4.4 Error class count breakdown for Mostly Basic Python Problems, showing that simple errors contributed to 75% of all errors.

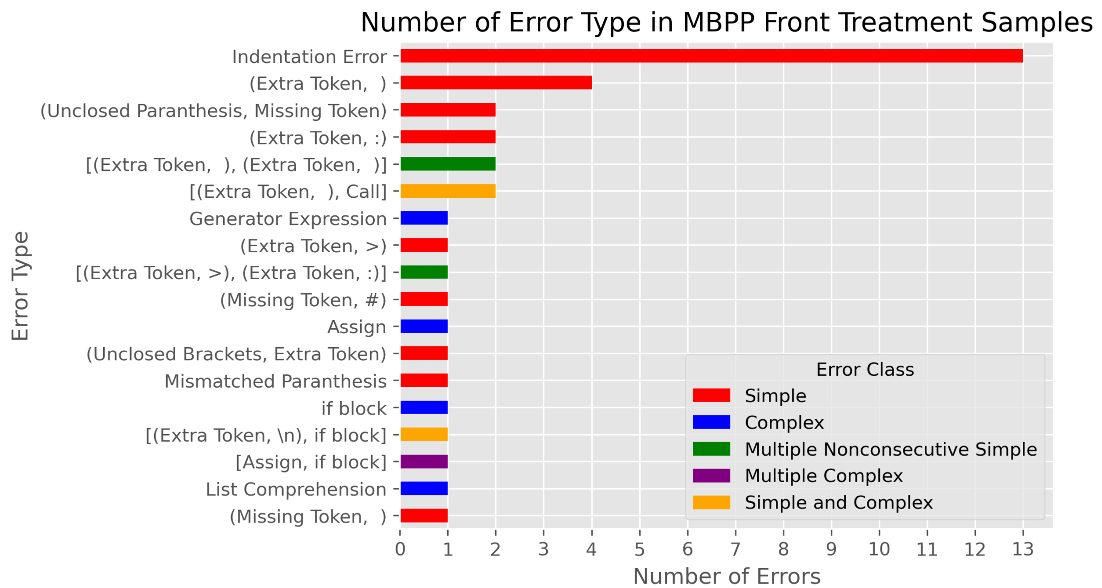


Figure 4.5 Error type count breakdown for Mostly Basic Python Problems, showing the most common error was an indentation error. Color designates error class.

4.2.1 Front Prompt Treatment

Out of the 500 samples tested, there were 484 python samples. There were 32 raw errors, and 29 true error samples. When we account for samples wtih multiple errors, there was a total of 36 errors overall as shown in Figure 4.7. This means the prompt caused a 100% and 107% increase in errors respectively.

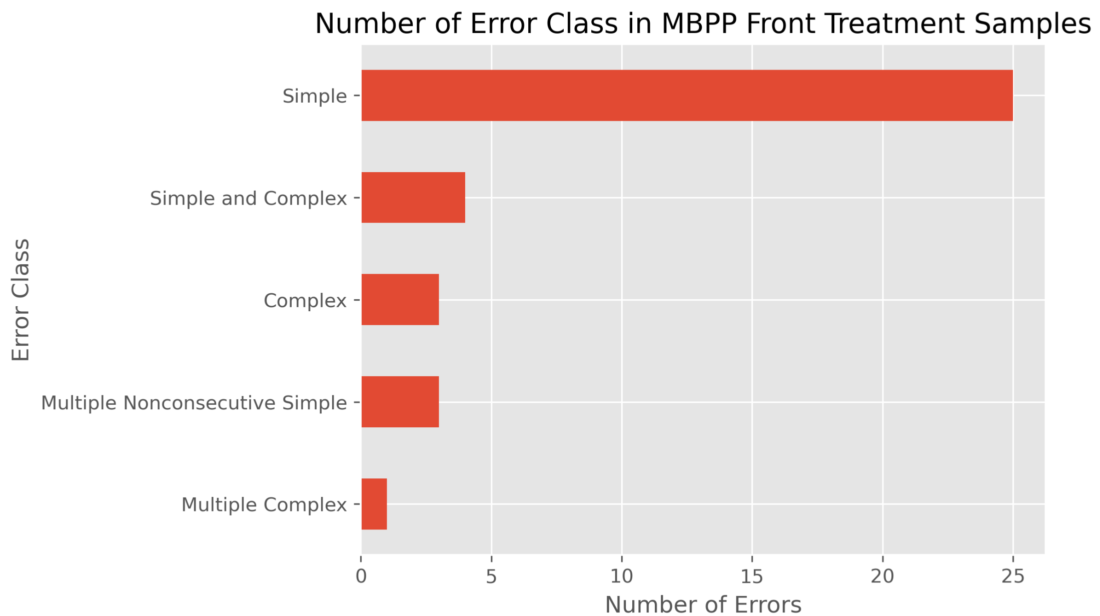


Figure 4.6 Error class count breakdown for the front prompt treatment test. This shows an overall increase in error numbers, but the errors are still overwhelmingly simple. (69.4%)

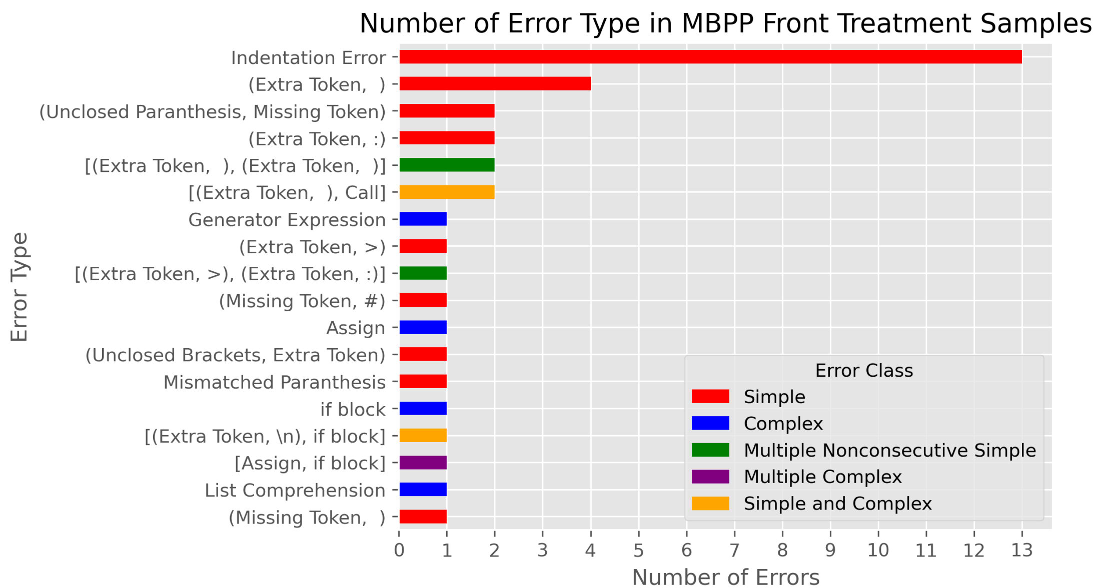


Figure 4.7 Error Type Count breakdown for the front prompt treatment test. This shows that the most common error type was Indentation error, and the most common complex error was 'Call'.

4.2.2 End Prompt Treatment

Out of the 500 samples tested with this treatment, there were 489 python samples. There were 28 raw errors, and 19 true errors. This means the prompt cause a 75%

and 58.3% increase respectively.

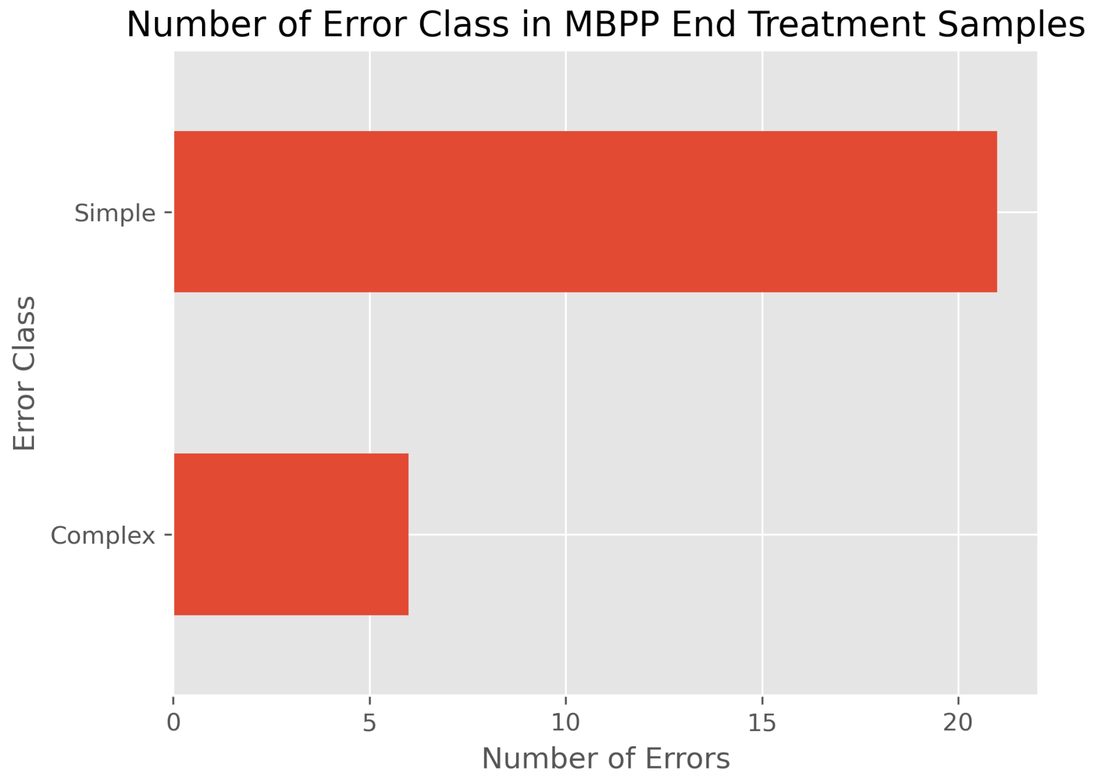


Figure 4.8 Error Class Count breakdown for the end prompt treatment. There are a lower number of errors that occured compared to the front treatment.

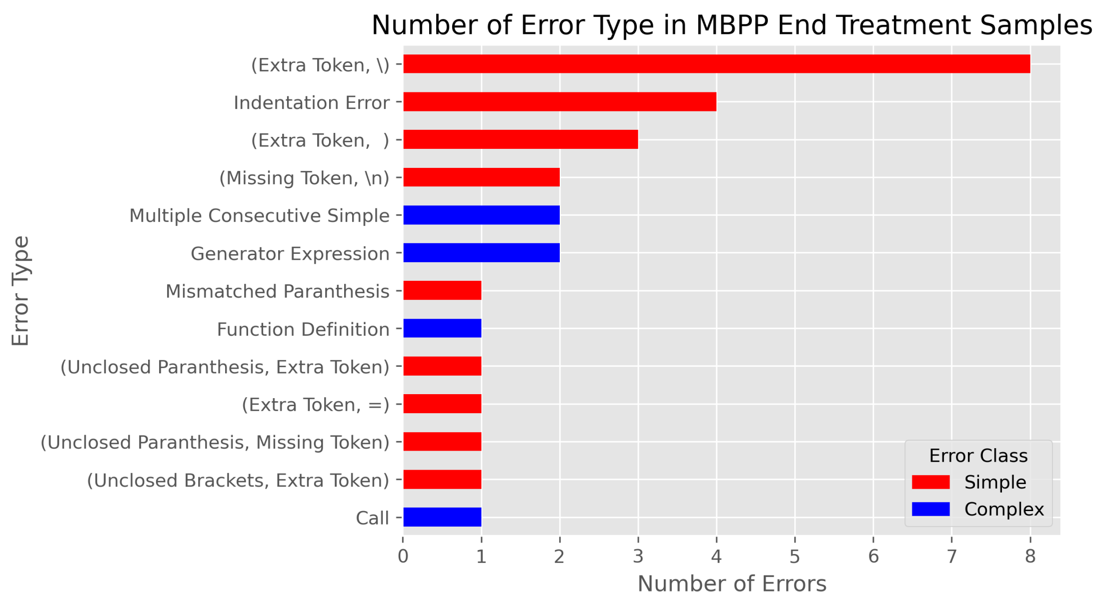


Figure 4.9 Error Type Count breakdown for the end prompt treatment. We find that the most common error is an extra ' \' token.

4.2.2.1 Statistical Significance

We computed a two-sample proportion z-test on the two prompt treatments. Keep in mind, the sample size we are comparing is the number of python code samples, rather than the MBPP size. The results of the two-sample proportion z-test comparing the error rates between the control and treatment groups are presented below:

Null Hypothesis: $p_1 = p_2$

Alternative Hypothesis: $p_1 \neq p_2$

- Control group:
  - Sample size: $n_1 = 495$
  - Number of errors: $x_1 = 14$
  - Error rate: $\hat{p}_1 = \frac{x_1}{n_1} = \frac{14}{500} = 0.028$
- Front Prompt:
  - Sample size: $n_2 = 484$
  - Number of errors: $x_2 = 29$
  - Error rate: $\hat{p}_2 = \frac{x_2}{n_2} = \frac{29}{484} = 0.0599$
- Two-sample proportion z-test:
  - Test statistic: $Z = \frac{(\hat{p}_1 - \hat{p}_2)}{\sqrt{\hat{p}(1-\hat{p})(\frac{1}{n_1}+\frac{1}{n_2})}}$
  - P-value: 0.0157

Based on the calculated p-value, we can conclude that the difference in error rates between the control and treatment groups is significant at the 0.05 level, indicating a statistically significant difference.

Null Hypothesis: $p_1 = p_3$

Alternative Hypothesis: $p_1 \neq p_3$

- End Prompt:
  - Sample size: $n_3 = 489$

  - Number of errors: $x_3 = 19$
  - Error rate: $\hat{p}_3 = \frac{x_3}{n_3} = \frac{19}{489} = 0.0389$

- Two-sample proportion z-test:
  - Test statistic: $Z = \frac{(\hat{p}_1 - \hat{p}_3)}{\sqrt{\hat{p}(1-\hat{p})(\frac{1}{n_1}+\frac{1}{n_3})}}$
  - P-value: 0.357

Based on the calculated p-value, we can conclude that the difference in error rates between the control and treatment groups is not significant at the 0.05 level, indicating no statistically significant difference.

4.2.3 Samples of Interest and Removed Samples

This section will discuss samples that were removed and the reason why, as well as any especially interesting samples.

4.3 CoNaLa

4.3.1 Rewritten

During our testing, we removed duplicate samples from the rewritten intent dataset. This caused our total sample count to drop to 2777 samples. Of which there were 2300 train samples, and 477 test samples.

4.3.1.1 Train

Of the 40 errors encountered in the 2300 training samples (1.74% error), 18 were Simple, 18 were Complex, 3 were Multiple Nonconsecutive Simple, and 7 were Multiple Complex as shown in Figure 4.12.

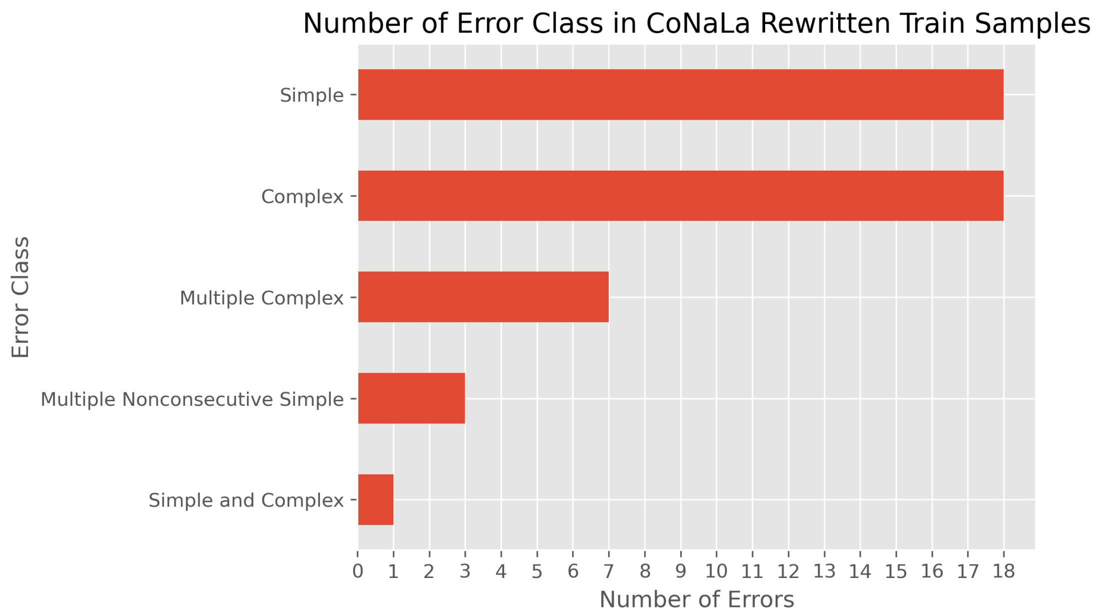


Figure 4.10 Error Class Count for the CoNaLa rewritten train samples. This is the only dataset where total complex errors were greater than total simple errors, including multiple error types.

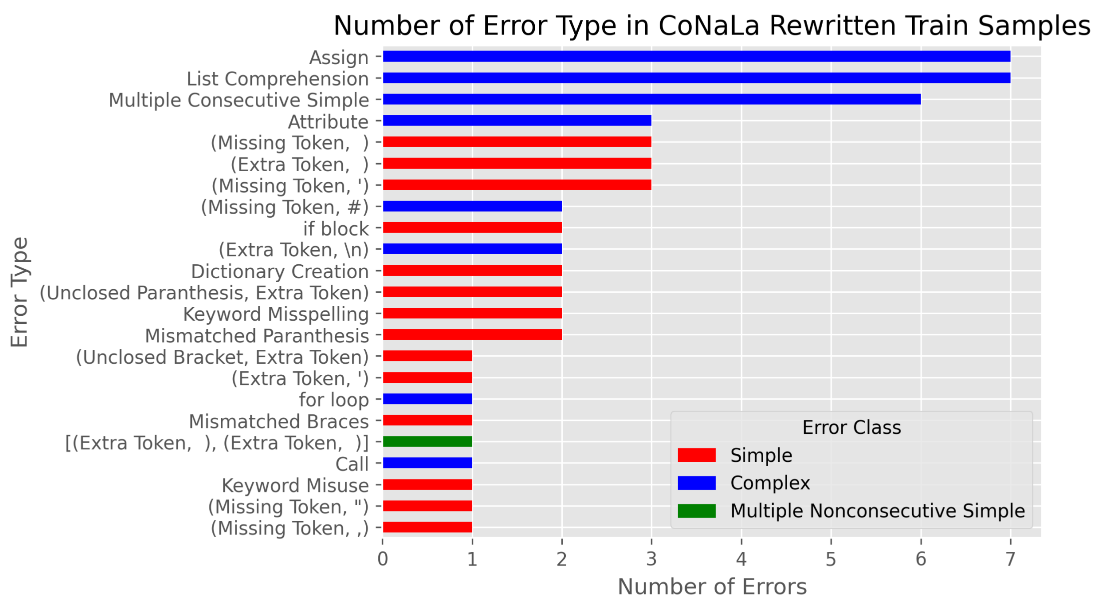


Figure 4.11 Error Type Count for the CoNaLa rewritten train samples. We found that the four most common error types were complex, this is unique to this dataset, and is the only dataset where a complex error was the most common. Color designates error class.

4.3.1.2 Test

Of the 11 errors in the test set, 8 were Simple and 5 were Complex.

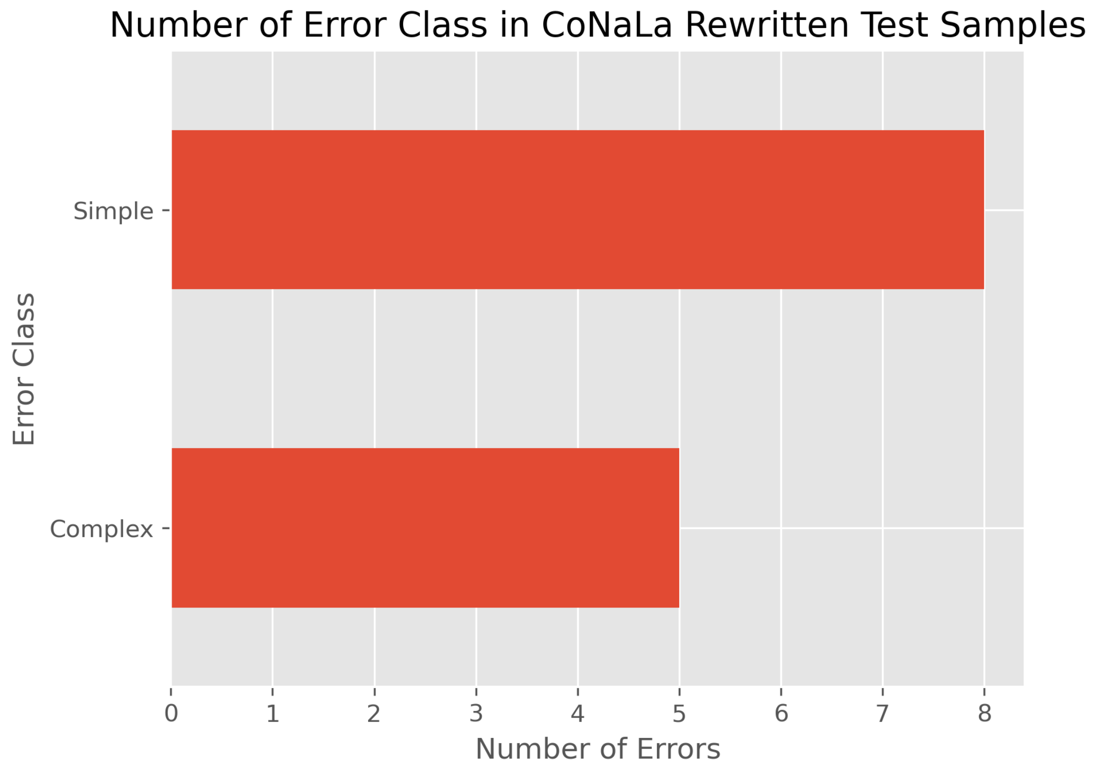


Figure 4.12 Error Class Count breakdown for the CoNaLa rewritten test set. This follows the typical pattern of the most common error being simple.

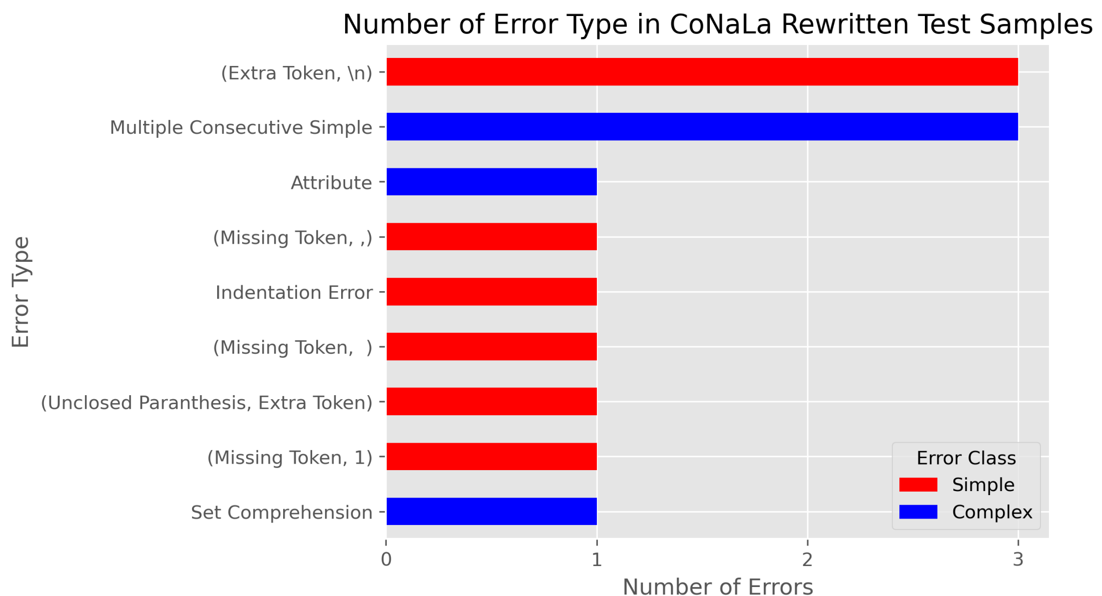


Figure 4.13 Error Type Count breakdown for the CoNaLa rewritten test set. Simple and Complex errors were tied for the most common error types. Color designates error class.

4.3.2 Intent

We only tested the test set of the CoNaLa intent prompt. There were 500 original samples. In our results, we had 475 python samples, which contained 19 raw errors and 12 true errors. (2.4% Error Rate) There were 8 Simple, 4 Complex, and 1 Multiple Complex as shown in Figure 4.14.

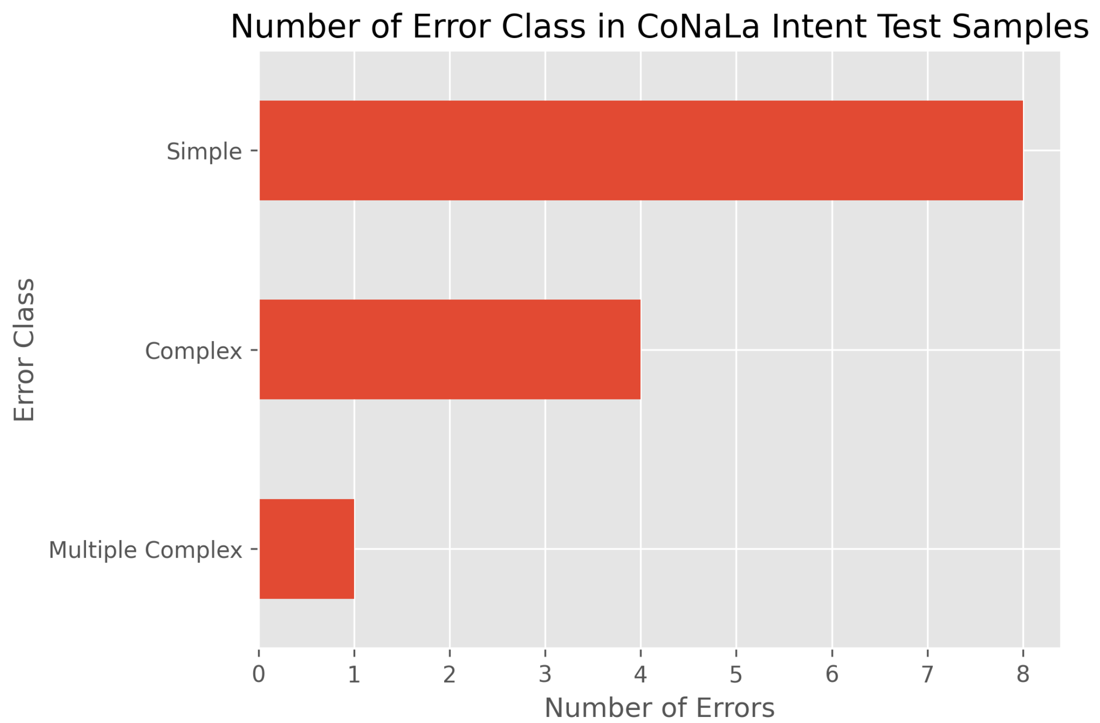


Figure 4.14 Error Class Count breakdown for the CoNaLa intent test set. This follows the typical pattern of the most common error being simple.

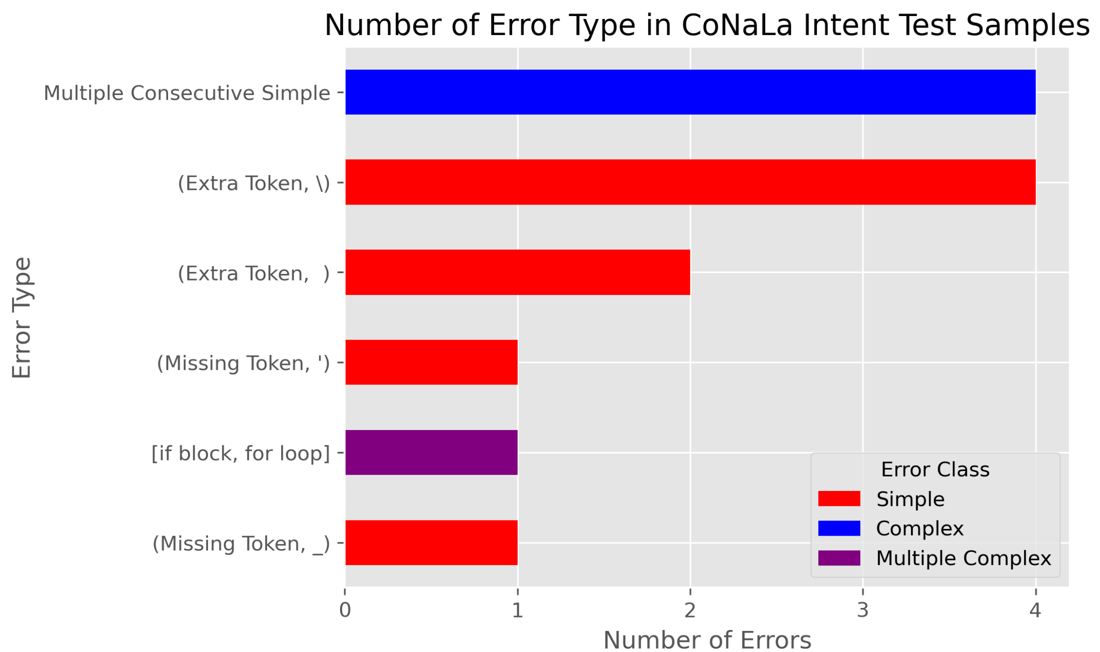


Figure 4.15 Error Type Count breakdown for the CoNaLa intent test set. Simple and Complex errors were tied for the most common error types. Color designates error class.

4.4 HumanEval

The HumanEval dataset was to cover the task of code completion. All of the code snippets provided were python code. There were 2 raw errors, and 1 true error. This was a complex error, breaking the relation type of list comprehension.

This is the code containing the error:

```
def even_odd_count(num):
    '''Given an integer. return a tuple that has the number of even
    and odd digits respectively.

     Example:
         even_odd_count(-12) ==> (1, 1)
         even_odd_count(123) ==> (1, 2)
    '''

    # Convert the number to a list of digits using list comprehension
    digits = [abs(num % 10) for num in abs(num) // 10 * 10 downTo 1]

    # Initialize counter variables
```

```
13      even_count = 0
14      odd_count = 0
15
16      # Iterate through the list of digits and increment the respective
        counter
17      for digit in digits:
18          if digit % 2 == 0:
19              even_count += 1
20          else:
21              odd_count += 1
22
23      # Return a tuple with the number of even and odd digits
24      return (even_count, odd_count)
25
26  python
27  print(even_odd_count(-12))  # Output: (1, 1)
28  print(even_odd_count(123))  # Output: (1, 2)
29
```

Listing 4.1: HumanEval error

The error occurs at line 10, with the call to downTo. This is not supported in python, and seems to be closer to other programming languages like Kotlin.

Chapter 5

Discussion

5.1 Main Findings

We found that most syntax errors generated by an LLM are caused by a single token (60.53%). Only 25% of errors broke a syntactical relationship. [10] The most common syntactical relationship broken by an LLM was the assign relationship, closely followed by list comprehension.

We also found that adding "Generate code only." to the front of the prompt increased the error rate by 107%, a statistically significant difference, while adding the same string to the end only produced a 58.3% increase which is not significant. Perhaps this is because adding the treatment to the front of the prompt causes the model to focus more on the code aspect, rather than solving the problem first.

5.1.1 RQ1: Common Error Types

As seen in 4.2, the most common error class was simple. The most common error type 4.3, was Indentation Error, followed by an extra space, then multiple consecutive simple. The most common complex error was multiple consecutive simple. The most common relationship violated was assign, followed by list comprehension.

5.1.2 RQ2: Difficultly of Software Engineering Tasks

We found that the model struggled the most with natural language to code tasks, and excelled at code completion. The highest error rate occurred in the MBPP front prompt treatment, with an error rate of 5.99%. The lowest error rate occurred in the HumanEval dataset, with only 1 error; an error rate of 0.61%.

5.1.3 RQ3: Prompt Changes

We found that adding "Generate code only." cause a 100% increase of true errors, from 14 to 32 on the MBPP dataset, while adding the same string to the end only caused a 58.3% increase.

5.1.4 RQ4: Task Differences

While there was only one error in the HumanEval test, it was a complex error, list comprehension specifically. As for the natural language to code task, we saw an abundance of simple errors, except for in the CoNaLa intent train dataset. This dataset had an equal amount of complex and simple errors.

5.2 Research Implications

Several previous studies have examined LLM syntax understanding by having the LLM examine the code. [10], [42] We have expanded on their work by exploring the way this internal AST understanding affects code generation. We have helped begin to open the black box of LLM internal code representation.

This exploratory study has shown that LLMs do seem to struggle with syntactical relationships in code generation to an extent. We provide a basis for future research to build upon in their experiments on code syntax understanding.

We have also shown that the same string, added in different places, can have a significant difference in quality of prompt generation. This has very significant prompt engineering potential. We suggest increasing prompt variance, and attempting to identify exactly why adding the string to the front increases errors significantly.

5.3 Study Limitations

There are several limitations on our experiments. While there are papers that include references to syntax error classification [43], [44], we were unable to find any previous literature exploring the classification of syntax errors to study LLM understanding. Therefore, the syntax error taxonomy is very experimental, and may have flaws. It was iterativelly updated as we conducted experiments, as many taxonomies are, which also could have introduced inconsistencies or contradictions.

There are two main extensions that can be made to this study. Expansion of the number and size of models tested. This is important because at the moment we cannot draw a conclusion on if the rate of syntax errors is caused by our model, or LLM architecture in general. Right now our results can only be applied to the mistral-7b-instruct model. As well as covering more machine learning for software engineering tasks. We do not cover code translation, or code correction. There are several questions that remain. How can models effectively correct syntax errors if they lack syntax understanding?

Of the software engineering tasks we did cover, the number of samples tested was extremely imbalanced. We tested 4,777 samples of natural language to code tasks, but only 164 of code completion. Given this imbalance, we cannot draw accurate conclusions about code completion, it needs to be explored further. We also only included python code in our study, which has less syntactical relationships than other languages, [10] due to its dynamically-typed interpreted nature.

We also only used two versions of the prompt engineering, and while they showed interesting results, the extent to which prompt engineering affects syntax errors in the generated code has yet to be fully explored.

5.4 Future Research

There are various directions future researcher can take. The limitations of our study mean that code completion, and other software engineering tasks, have yet to be explored in this way. We also only used one model due to hardware constraints, so the difference in model size is another avenue of exploration. Another potential subject of future work is the exploration of different coding languages. This could

also be included in the prompt engineering treatments. We propose a research question such as:

RQ: Does requesting a different programming language change the amount of syntax errors produced?

Given the lack of a literature background in syntax taxonomies used for this task, a reworked taxonomy could be used to repeat this study. Perhaps future work can explore the use of error classification methods used in syntax error correction and explanation tools. [43], [44]

As mentioned above, causal and empirical analysis of the way prompts affect syntax errors in generated code is another potential extension of this study.

Overall, every aspect of our study can be expanded on in some way, such as increased sample sizes, different models, different datasets, more software engineering tasks, and different prompt engineering treatments.

Chapter 6

Conclusions

In this paper, we conducted an exploratory study on LLM syntax error understanding in various software engineering tasks. In particular, we examine LLMs ability to generate syntax error free python code on 4,941 prompt samples, over two software engineering tasks. Overall, the results of this study indicated that LLMs have decent code syntax understanding, but do struggle at times. We found that the most common errors have to do with white-space, a key part of python syntax. We believe our study provides a basis for future research into LLM understanding of various software engineering tasks.

References

[1] J. Austin, A. Odena, M. I. Nye, *et al.*, “Program synthesis with large language models”, *CoRR*, vol. abs/2108.07732, 2021. arXiv: 2108.07732. [Online]. Available: https://arxiv.org/abs/2108.07732.

[2] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, “Attention is all you need”, *CoRR*, vol. abs/1706.03762, 2017. arXiv: 1706.03762. [Online]. Available: http://arxiv.org/abs/1706.03762.

[3] S. Ren, D. Guo, S. Lu, *et al.*, “Codebleu: A method for automatic evaluation of code synthesis”, *CoRR*, vol. abs/2009.10297, 2020. arXiv: 2009.10297. [Online]. Available: https://arxiv.org/abs/2009.10297.

[4] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: A method for automatic evaluation of machine translation”, in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL ’02, Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. [Online]. Available: https://doi.org/10.3115/1073083.1073135.

[5] Z. Feng, D. Guo, D. Tang, *et al.*, “Codebert: A pre-trained model for programming and natural languages”, *CoRR*, vol. abs/2002.08155, 2020. arXiv: 2002.08155. [Online]. Available: https://arxiv.org/abs/2002.08155.

[6] B. Rozière, J. Gehring, F. Gloeckle, *et al.*, *Code llama: Open foundation models for code*, 2024. arXiv: 2308.12950 [cs.CL].

[7] C. B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, “Pymt5: Multi-mode translation of natural language and python code with transformers”, *CoRR*, vol. abs/2010.03150, 2020. arXiv: 2010.03150. [Online]. Available: https://arxiv.org/abs/2010.03150.

[8] A. Mastropaolo, S. Scalabrino, N. Cooper, *et al.*, “Studying the usage of text-to-text transfer transformer to support code-related tasks”, *CoRR*, vol. abs/2102.02017, 2021. arXiv: 2102.02017. [Online]. Available: https://arxiv.org/abs/2102.02017.

[9] D. N. Palacio, A. Velasco, N. Cooper, A. Rodriguez, K. Moran, and D. Poshyvanyk, “Toward a theory of causation for interpreting neural code models”, *IEEE Transactions on Software Engineering*, pp. 1–28, 2024, ISSN: 2326-3881. DOI: 10.1109/tse.2024.3379943. [Online]. Available: http://dx.doi.org/10.1109/TSE.2024.3379943.

[10] D. Shen, X. Chen, C. Wang, K. Sen, and D. Song, "Benchmarking language models for code syntax understanding", in *Findings of the Association for Computational Linguistics: EMNLP 2022*, Y. Goldberg, Z. Kozareva, and Y. Zhang, Eds., Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, December 2022, pp. 3071–3093. DOI: 10.18653/v1/2022.findings-emnlp.224. [Online]. Available: https://aclanthology.org/2022.findings-emnlp.224.

[11] A. Kurenkov, *A brief history of neural nets and deep learning*, Sep. 2020. [Online]. Available: https://www.skynettoday.com/overviews/neural-net-history.

[12] X. Yan and X. Su, *Linear Regression Analysis Theory and computing*. World Scientific, 2009.

[13] I. Data and AI, *Ai vs. machine learning vs. deep learning vs. neural networks: What's the difference?*, Jul. 2023. [Online]. Available: https://www.ibm.com/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks/.

[14] F. Rosenblatt, "The perceptron - a perceiving and recognizing automaton", Cornell Aeronautical Laboratory, Ithaca, New York, Tech. Rep. 85-460-1, January 1957.

[15] M. L. Minsky and S. Popeit, *Perceptrons*. MIT Pr.), 1969.

[16] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", *Nature*, vol. 323, pp. 533–536, 1986. [Online]. Available: https://api.semanticscholar.org/CorpusID:205001834.

[17] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks", in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, G. Gordon, D. Dunson, and M. Dudík, Eds., ser. Proceedings of Machine Learning Research, vol. 15, Fort Lauderdale, FL, USA: PMLR, November 2011, pp. 315–323. [Online]. Available: https://proceedings.mlr.press/v15/glorot11a.html.

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[19] M. A. Nielsen, January 2019. [Online]. Available: http://neuralnetworksanddeeplearnin com/chap2.html.

[20] S. Basodi, C. Ji, H. Zhang, and Y. Pan, "Gradient amplification: An efficient way to train deep neural networks", *Big Data Mining and Analytics*, vol. 3, no. 3, pp. 196–207, 2020. DOI: 10.26599/BDMA.2020.9020004.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation", in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA, USA: MIT Press, 1986, pp. 318–362, ISBN: 026268053X.

[24] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities", *Proceedings of the National Academy of Sciences of the United States of America*, vol. 79, no. 8, pp. 2554–2558, April 1982, ISSN: 0027-8424. [Online]. Available: `http://view.ncbi.nlm.nih.gov/pubmed/6953413]`.

[25] A. M. Turing, "Computing machinery and intelligence", English, *Mind*, New Series, vol. 59, no. 236, pp. 433–460, 1950, ISSN: 00264423. [Online]. Available: `http://www.jstor.org/stable/2251299`.

[26] F. Jelinek, "Continuous speech recognition by statistical methods", *Proceedings of the IEEE*, vol. 64, no. 4, pp. 532–556, 1976. DOI: `10.1109/PROC.1976.10159`.

[27] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation", *CoRR*, vol. abs/1406.1078, 2014. arXiv: `1406.1078`. [Online]. Available: `http://arxiv.org/abs/1406.1078`.

[28] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks", *CoRR*, vol. abs/1409.3215, 2014. arXiv: `1409.3215`. [Online]. Available: `http://arxiv.org/abs/1409.3215`.

[29] Jun. 2018. [Online]. Available: `http://jalammar.github.io/illustrated-transformer/`.

[30] R. Waldinger, R. Lee, and S. International, *PROW: A Step Toward Automatic Program Writing*. SRI International, 1969. [Online]. Available: `https://books.google.com/books?id=3BITSQAACAAJ`.

[31] D. E. Shaw, W. R. Swartout, and C. C. Green, "Inferring lisp programs from examples", in *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'75, Tblisi, USSR: Morgan Kaufmann Publishers Inc., 1975, pp. 260–267.

[32] A. Odena and C. Sutton, "Learning to represent programs with property signatures", *CoRR*, vol. abs/2002.09030, 2020. arXiv: `2002.09030`. [Online]. Available: `https://arxiv.org/abs/2002.09030`.

[33] J. W. Backus, R. J. Beeber, S. Best, *et al.*, "The fortran automatic coding system", in *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, ser. IRE-AIEE-ACM '57 (Western), Los Angeles, California: Association for Computing Machinery, 1957, pp. 188–198, ISBN: 9781450378611. DOI: `10.1145/1455567.1455599`. [Online]. Available: `https://doi.org/10.1145/1455567.1455599`.

[34] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk, "A systematic literature review on the use of deep learning in software engineering research", *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, March 2022, ISSN: 1049-331X. DOI: `10.1145/3485275`. [Online]. Available: `https://doi.org/10.1145/3485275`.

[35] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories", in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15, Florence, Italy: IEEE Press, 2015, pp. 334–345, ISBN: 9780769555942.

[36] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection", in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16, Singapore, Singapore: Association for Computing Machinery, 2016, pp. 87–98, ISBN: 9781450338455. DOI: 10.1145/2970276.2970326. [Online]. Available: https://doi.org/10.1145/2970276.2970326.

[37] D. N. Palacio, A. Velasco, D. Rodriguez-Cardenas, K. Moran, and D. Poshyvanyk, *Evaluating and explaining large language models for code using syntactic structures*, 2023. arXiv: 2308.03873 [cs.SE].

[38] D. Rodriguez-Cardenas, D. N. Palacio, D. Khati, H. Burke, and D. Poshyvanyk, *Benchmarking causal study to interpret large language models for source code*, 2023. arXiv: 2308.12415 [cs.SE].

[39] A. Q. Jiang, A. Sablayrolles, A. Mensch, *et al.*, *Mistral 7b*, 2023. arXiv: 2310.06825 [cs.CL].

[40] M. Chen, J. Tworek, H. Jun, *et al.*, "Evaluating large language models trained on code", *CoRR*, vol. abs/2107.03374, 2021. arXiv: 2107.03374. [Online]. Available: https://arxiv.org/abs/2107.03374.

[41] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow", in *International Conference on Mining Software Repositories*, ser. MSR, ACM, 2018, pp. 476–486. DOI: https://doi.org/10.1145/3196398.3196408.

[42] W. Ma, S. Liu, Z. Lin, *et al.*, *Lms: Understanding code syntax and semantics for code analysis*, 2024. arXiv: 2305.12138 [cs.SE].

[43] T. Ahmed, N. R. Ledesma, and P. Devanbu, "Synshine: Improved fixing of syntax errors", *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2169–2181, 2023. DOI: 10.1109/TSE.2022.3212635.

[44] K. Zhang, H. Chen, L. Li, and W. Wang, *Syntax error-free and generalizable tool use for llms via finite-state decoding*, 2023. arXiv: 2310.07075 [cs.CL].

[45] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis", *Commun. ACM*, vol. 14, no. 3, pp. 151–165, March 1971, ISSN: 0001-0782. DOI: 10.1145/362566.362568. [Online]. Available: https://doi.org/10.1145/362566.362568.

[46] H. Saadany and C. Orasan, "Bleu, meteor, bertscore: Evaluation of metrics performance in assessing critical translation errors in sentiment-oriented text", *CoRR*, vol. abs/2109.14250, 2021. arXiv: 2109.14250. [Online]. Available: https://arxiv.org/abs/2109.14250.

[47] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, *No need to lift a finger anymore? assessing the quality of code generation by chatgpt*, 2023. arXiv: 2308.04838 [cs.SE].

[48] A. Kelly, "A system for classifying and clarifying python syntax errors for educational purposes", 2018. DOI: http://hdl.handle.net/1721.1/119750. [Online]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/119750/1078691087-MIT.pdf?sequence=1&isAllowed=y.

[49] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding", in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds., Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. [Online]. Available: https://aclanthology.org/N19-1423.

[50] R. Rojas, *Neural networks: A systematic introduction*. Springer, 2000.

[51] A. Rush, "The annotated transformer", in *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, E. L. Park, M. Hagiwara, D. Milajevs, and L. Tan, Eds., Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 52–60. DOI: 10.18653/v1/W18-2509. [Online]. Available: https://aclanthology.org/W18-2509.