# NLPy - From text to code

16:954:577 STATISTICAL SOFTWARE

*Team 5 – Devyani Mardia, Neeti Patel, Abhinav Saxena*

## 1. Introduction

This project explores the fascinating challenge of code generation - transforming natural language descriptions into executable code, a task that lies at the cutting edge of artificial intelligence and software development. It seeks to develop a system capable of understanding textual instructions and converting them into syntactically and semantically accurate programming language.

The approach involves a meticulous process of training and fine-tuning neural network models, starting with a simpler baseline model to set a performance benchmark and then advancing to more sophisticated, pre-trained models like *PyCodeGPT, GPT2, CodeGen 2, CodeT5-small,* and *CodeT5-base*. The project places a strong emphasis on evaluation, comparing the efficacy of different models in understanding the intricacies of human language and accurately translating them into code.

Through fine-tuning and strategic dataset enhancement, we observed significant improvements, with the model achieving a BLEU score of 37% on the dataset. This report describes the journey through the challenges and breakthroughs of this code generation task, offering insights into both the potential and the drawbacks.

---

## 2. Data Collection

### 2.1 Datasets

The data collection for this project was focused on assembling a parallel dataset comprising text descriptions and corresponding Python code. Multiple sources were used, including *Code Parrot*, *Semeru's Text-Code Galeras*, *Mostly Basic Python Programming (MBPP)*, and *PyTorrent*. These datasets were specifically chosen for their wide range of coding patterns and examples, which are essential for training a robust model capable of understanding and generating Python code.

### 2.2 Data Extraction

The data extraction phase involved pulling Python code snippets and their related text descriptions from the sources mentioned in section 2.1. This involved a systematic approach to identifying and retrieving datasets that were extensive.

### 2.3 Data Cleaning

In this phase, significant emphasis was placed on maintaining dataset integrity through language filtering and standardizing code styles to maintain uniformity. The consistency of the data was ensured by removing any records that were not in English, thereby streamlining the dataset to a homogeneous linguistic set that would be conducive to the model's training. Only text and code fields were retained, discarding all other irrelevant data to streamline the focus for model training. The standardization process involved normalizing various coding styles into a consistent format. For example, keeping the characters that indicate new line tag (\n or NEWLINE), tab (\t or INDENT), and carriage return (\r or DEDENT) the same in all data sets.

### 2.4 Data Compilation

Post-cleaning, individual JSONL files were created for each dataset. The final and pivotal step in the data preparation involved merging these individual files into a single, unified corpus, which was then used for model training and evaluation.

The final compiled dataset has 46,584 records with 2 fields – 'text' and 'code.'

### 2.5 Data Segmentation

The dataset was partitioned using the *train_test_split* function from the *sklearn* library, which randomly shuffled and split the data to ensure an unbiased allocation. Specifically, 70% was set aside for training. The remaining 30% was further divided, with 20% for validation and 10% for testing. The exact numbers of text-code pairs in each segment are detailed in Table 2.5.1 for reference.

| Data Segment | Number of Text-Code Pairs |
|---|---|
| Train | 46,584 |
| Validation | 9,317 |
| Test | 4,659 |

*Table 2.5.1 Data Distribution*

---

## 3. Model Evaluation and Metrics

For the evaluation of the generated code's quality, this project utilized the Sentence BLEU score as the primary metric. Sentence BLEU was chosen due to its precision-oriented nature, focusing on the presence and frequency of token sequences found in the model output as compared to a reference. In code generation, where syntactic and functional accuracy is important, the ability of Sentence BLEU to assess the correctness of sequences such as function calls and control structures made it an ideal fit.

While Sentence BLEU provides a quantitative measure that is straightforward to compute and compare, it is acknowledged that it does not fully capture semantic accuracy or the execution success of the generated code. Therefore, it was supplemented with qualitative assessments, including manual code reviews and execution tests, to provide a more comprehensive evaluation of the model's performance.

---

## 4. Baseline Model for text-to-code

The NLP task of text-to-code generation involves the use of a Sequence-to-Sequence model to transform text input sequences into code output sequences. This base model comprises two primary components: an Encoder and a Decoder. The Encoder's role is to encode the input sequence into a context vector, which the Decoder then utilizes to generate the output sequences.

### 4.1 Model Architecture

In the initial step, input words are converted into tokens using the 'bert-base-cased' tokenizer, resulting in input IDs. The baseline model is structured with three layers: the Embedding layer, an LSTM layer, and a Linear layer. Image 4.1.1 indicates the architecture of the baseline model.

Embedding Layer: Given the word-level token processing, an embedding layer with a

dimensionality of 256 is employed. This allows each token to be represented as a vector of this size.

LSTM (Encoder) Layer: The baseline model incorporates an LSTM layer featuring 512 hidden units and is composed of 2 layers. The model's performance has the potential to be enhanced through the addition of more LSTM layers.

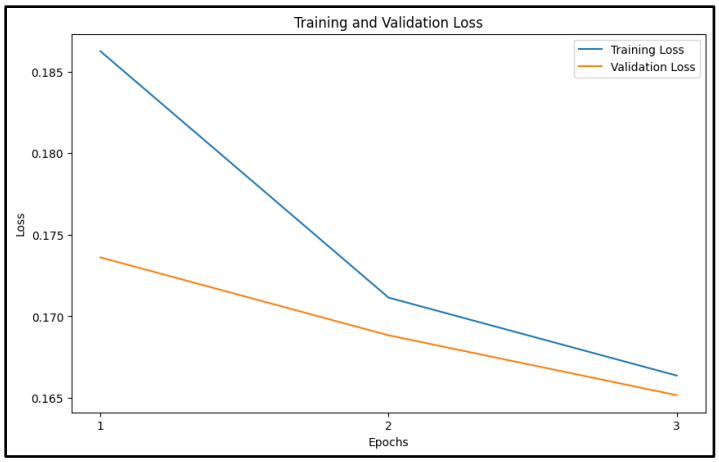
Linear (Decoder) Layer: This fully connected linear layer is designed to map the output from the LSTM/encoder to the vocabulary size, effectively determining the next token IDs.

```
LSTMTextToCode(
  (embedding): Embedding(30522, 256)
  (lstm): LSTM(256, 512, num_layers=2, batch_first=True)
  (fc): Linear(in_features=512, out_features=30522, bias=True)
)
```

*Image 4.1.1 Baseline Model Architecture*

### 4.2 Model Training

A comprehensive pipeline was established for setting up, training, validating, and evaluating the above model. Training is conducted by iterating through the training data in batches using the DataLoader, with set hyperparameters: epochs at 3, an Adam optimizer, a learning rate of 5e-5, a batch size of 32, a maximum output length of 128, and the Cross-Entropy loss function. Training loss converged from 0.18 to 0.16, while the validation loss converged from 0.17 to 0.16. See Graph 4.2.1 for the graph of training and validation loss.




*Graph 4.2.1 Training and Validation Loss for the Baseline Model*

### 4.3 Model Evaluation

For evaluation purposes, the model's generated outputs (or output sequences) undergo a process where the argmax is taken to select the token with the highest probability. These tokens are then decoded using the "tokenizer.decode" method to produce the final results. The model achieved a Sentence BLEU score of 0.26%.

---

## 5. Choosing the Optimal Pre-trained Model

### 5.1 Initial Options

For this project, the primary options for the pre-trained model were *GPT-2, PyCodeGPT, CodeGen 2-1B, CodeT5-small,* and *CodeT5-base.* These pre-trained models were trained on a dataset of 4000 rows. For the results of this experiment, refer to section 5.2.

*GPT-2*, while powerful in text generation, lacked specialization in code generation. It generated the same string which was in the text input. *PyCodeGPT* had improved scores but also generated the same

string as the input text. *CodeGen 2* with one billion parameters was a failed attempt due to the limited computational power. *CodeT5-small*, tailored for programming languages, was promising and this model finally generated code that, though not perfect, was better than the other models.

*CodeT5-small* was surpassed by *CodeT5-base* in BLEU without exceeding our computational power. Its base size model was the right mix of complexity and efficiency, crucial for better translation of text to Python code, making it the most suitable option for the project's specific needs.

### 5.2 Comparative Analysis

Table 5.2.1 shows a comparison of evaluation metrics for the pre-trained model options when trained on about 4,000 records.

| Model | Sentence BLEU |
|---|---|
| GPT2 | 0.043% |
| PyCode GPT | 0.039% |
| CodeT5- small | 0.15% |
| **CodeT5-base** | **0.26%** |

*Table 5.2.1 Comparison of pre-trained models*

---

## 6. CodeT5 – base

### 6.1 Model Architecture

In this project, the architecture of choice for the text-to-code conversion task was CodeT5 base, a state-of-the-art model built upon the encoder-decoder framework of the T5 model (Raffel et al., 2020). CodeT5 is specifically pre-trained on a corpus of unlabeled source code, aiming to derive sophisticated, generic representations for both programming language (PL) and natural language (NL).

This model goes beyond the standard T5 by integrating identifier-aware pre-training tasks. These tasks are designed to harness the information provided by the token types common in PL, such as identifiers which are typically assigned by developers, allowing for a more nuanced understanding and generation of code.

Additionally, CodeT5 introduces a bimodal dual learning objective that facilitates a bidirectional conversion between NL and PL. This feature is particularly crucial in aligning the two modalities, enabling the model to not only translate natural language instructions into executable code but also to perform the reverse.

It is a Unified Encoder-Decoder Model with:

- Vocabulary Size: 32,100 tokens
- Embedding Dimensions: 768
- Encoder and Decoder Blocks: 12 T5 Blocks each with Self-Attention, Cross-Attention, and Feedforward Network
- Output Layer: Linear (LM Head) with 32,100 output features

Figure 6.1.1 demonstrates the model's ability to perform diverse coding tasks. For the purpose of this project, fine-tuning was carried out to refine the model's capabilities, aligning it with the specific requirements of generating Python code from natural

language descriptions. Refer to section 7 for fine-tuning methodology.

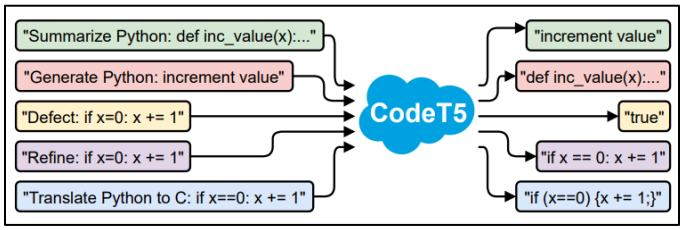



*Figure 6.1.1 Illustration of CodeT5 for code-related understanding and generation tasks*

### 6.2 Model performance on the dataset without training

The project initially evaluated the capabilities of the pre-trained CodeT5-base model by testing it on a dataset of over 40,000 records. This assessment was conducted without any fine-tuning to check the raw performance of the model on the task of generating Python code from natural language descriptions.

The results from this preliminary evaluation indicated a BLEU score of 0.03%, reflecting the model's baseline ability to understand and produce code sequences. Although modest, this score provided a benchmark for the potential enhancements that could be achieved through subsequent fine-tuning and optimization tailored to our dataset and tasks.

---

## 7. Model Fine-Tuning

### 7.1 Data Partitioning

The dataset was scaled incrementally to assess the impact of data volume on model performance. Initially, the model was fine-tuned on 4,000 records, followed by a larger set of about 10,000 records, and finally on the complete dataset comprising over 40,000 records. At each stage, the data was partitioned in a 70-20-10 ratio for training, validation, and testing to maintain a consistent evaluation framework across all phases of model development.

### 7.2 Epochs

The model was fine-tuned over a range of 4 to 6 epochs. This specific range was selected to find a balance that would allow the model adequate exposure to the data to learn effectively, while also avoiding the model's over-adjustment to the training set, which could lead to poor generalization on unseen data.

It was observed that after completing the 4 to 6 epochs, signs of overfitting began to show in some models. This was indicated by an increase in validation loss, which denotes that the model's performance was starting to degrade on the validation set, even though it may have continued to improve on the training set.

### 7.3 Optimizer Selection

Adam optimizer was employed for its adeptness in handling large datasets and complex model architectures. Its adaptive learning rate capabilities facilitated efficient convergence during the fine-tuning phase.

### 7.4 Batch Size Determination

While experimenting with the batch size, it was observed that a batch size greater than 8 led to memory constraints. Consequently, the batch size was set to 8 to maximize resource efficiency and

avoid space issues, thereby ensuring smooth model training and optimization.

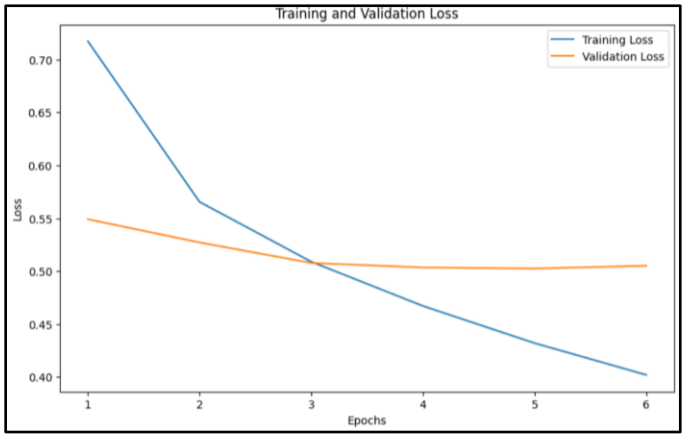
### 7.5 Training and Validation Loop

The training and validation loop played a crucial role in the effective fine-tuning of the CodeT5-base model, especially given its substantial size of 220 million parameters. To optimize and expedite training time on the GPU, strategic decisions were made during the training loop. The model underwent a total of 6 epochs, divided into two phases of 4 and 2 epochs each, showing convergence around the 5th and 6th epoch without signs of overfitting in training or validation losses. Graph 7.5.1 shows the training and validation loss for each epoch.

PyTorch's DataLoader class was utilized to create Train and Validation loaders for efficient batch processing, aiding in computational efficiency and stability in convergence. The process involved clearing gradients, calculating them through backpropagation, and updating weights accordingly, with an averaging of loss.

During the validation phase, no gradient calculations were performed as the model operated in evaluation mode. A pivotal factor in achieving satisfactory performance while maintaining quality was the implementation of Mixed Precision training. This technique employs both 32 and 16-bit floating-point types during training, enhancing speed on GPUs. Critical components like weight updates used 32-bit precision, while forward and backward passes were computed in 16-bit. Loss scaling was employed to mitigate the issue of gradient underflow associated with 16-bit computations.

Tools such as GradScaler (for loss scaling) and autocast (to enable mixed precision) were integral in successfully implementing Mixed Precision Training. This approach significantly reduced the training time of each epoch from 60 to 30 minutes. The optimal hyperparameter combination for this training involved parameters such as 6 epochs, a batch size of 8, the Adam optimizer, and a learning rate of 5e-5.




*Graph 7.5.1 Training and Validation Loss for the final fine-tuned model*

### 7.6 Performance Evaluation

Model performance was assessed using the sentence BLEU score. The initial BLEU score of 0.03% (from Section 6.2) marked the untrained model's baseline ability. Fine-tuning with 4,000 records improved the BLEU score to 0.26%, and further fine-tuning with about 9,000 records resulted in a BLEU score of 24%. The full dataset of over 40,000 records achieved a BLEU score of 37%, confirming the effectiveness of the fine-tuning process in enhancing the model's code generation accuracy. Refer to Table 7.6.1 for a comparative summary of fine-tuning. The values in the table clearly indicate the importance of having a good dataset from which the model can

learn. The quality as well as the quantity of data matters.

| Model | Number of Records | BLEU score |
|---|---|---|
| Baseline – Seq2Seq | 46,584 | 0.26% |
| Original CodeT5 – base without training | 46,584 | 0.03% |
| Code T5 – base fine-tuned | 3,911 | 0.26% |
| Code T5 – base fine-tuned | 9,263 | 24% |
| **Code T5 – base fine-tuned** | **46,584** | **37%** |

*Table 7.6.1 Comparison of BLEU Scores*

### 7.7 Failed Experiments

In the course of this project, not all experiments yielded positive results. A notable trial involved implementing a weight decay of 0.1 as a regularization technique. This approach, intended to mitigate overfitting by penalizing large weights, resulted instead in a model that was significantly underfitted, unable to capture the complexities of the data. One of the unsuccessful fine-tuning attempts comprised adding a linear layer on top of the CodeT5-base model's architecture with the intention of doing specialized training on this layer for Python code generation. However, trying to train the model by freezing model layers, or gradual unfreezing of model layers to total unfreezing of model layers had only negative impacts on the model's performance as the weights of the new layer must have been unstable so the decision to not continue with this approach was reached.

An experiment with few-shot learning yielded performance comparable to the baseline model but limited by the availability of diverse, instruction-based/prompt-based examples, necessary for significant improvement in code generation tasks.

These failed experiments were valuable learning experiences, underscoring the delicate balance required in model tuning and the specificity of the dataset to the task of code generation.

---

## 8. Results

In our exploration of code generation using the model, we put it to the test with a few prompts. Image 8.1 is the NLPy model output for the prompt "Program to add two numbers" and Image 8.3 is the NLPy model output for the prompt "Create a function to check if a string is a palindrome." The resulting code, while a nice attempt by the model to meet the requirements, has some notable imperfections. The first code snippet has indentation and bracketing errors, hinting at the model's struggle with some of Python's syntactic nuances. The second example, a function to determine palindromes, although functionally on the right track, had structural errors, including an incorrect call to a non-existent function "check_palindrome."

Manual correction was required to rectify these issues. Post-correction, both code snippets were executed in the environment, shown in Image 8.2 and Image 8.4, and to our satisfaction, they worked as

intended. This not only demonstrated the model's potential in understanding and responding to programming tasks but also the need for human evaluation.

```
def add ( x, y ) :
        if x == 0 or y == 0 :
        return x
 return x + y
 x = 10
 y = 10
 print add ( x, y )
```

*Image 8.1 NLPy model output for the prompt "Program to add two numbers."*

```
def add ( x, y ) :
  if x == 0 or y == 0 :
    return x
  return x + y
x = 10
y = 10
print(add(x, y))
```

```
20
```

*Image 8.2 Executed code from image 8.1 with manual corrections*

```
def isPalindrome ( s ) :
        n = len ( s )
 for i in range ( n ) :
        if ( s [ i ]!= s [ n - i - 1 ] ) :
        return False
 return True
 def checkPalindrome ( s ) :
        n = len ( s )
 if ( checkPalindrome ( s ) ) :
        print ( " Yes " )
 else :
        print ( " No " )
```

*Image 8.3 NLPy model output for the prompt "Create a function that checks if a string is a palindrome."*

```
def isPalindrome(s):
  n = len(s)
  for i in range(n):
      if (s[i]!= s[n - i - 1]) :
        return False
  return True
def checkPalindrome(s):
  # n = len(s)
  if (isPalindrome(s)):
    print( " Yes " )
  else:
    print( " No " )
checkPalindrome("MOM")
```

```
Yes
```

*Image 8.4 Executed code from image 8.3 with manual corrections*

Image 8.5 indicates that while our model adeptly navigated through the complexities of creating functions for addition and palindrome checks, it stumbled upon the seemingly straightforward task of printing "Hello World." It highlights a quirk in the training process, where the model, busy mastering advanced coding concepts, may have oversight over the basics. This unexpected quirk underscores the need for a dataset that is not only diverse but also balanced across varying levels of coding complexity, ensuring that the model is equally adept at handling both fundamental and advanced programming tasks.

```
def print_hello_world():
     print hello world print hello world
 def print_hello_world():
     print hello world print hello world
 def print_hello_world_world():
     print hello world
     print hello world
    print hello world
    print hello world_world_world()
    print hello world_world_world()
    print hello world_world_world()
```

*Image 8.5 NLPy model output for the prompt "Print hello world"*

## 9. Limitations and Potential Enhancements

### 9.1 Computational Constraints

A key limitation faced by this project, as with many in this field, is computational constraints. The choice of hyperparameters, the size of the pre-trained model, and the volume of the dataset used for training were all influenced by the available computational resources. To address this, future iterations of the project could explore more efficient model architectures or seek access to more powerful computing resources.

### 9.2 Model Generalization

The model encountered challenges in generalizing to coding tasks beyond its training scope. An approach like few-shot learning could be implemented to enhance the model's ability to learn from a smaller number of examples, potentially improving its adaptability to a wider range of coding tasks.

### 9.3 Data Diversity

While the dataset used was extensive, it did not cover all of Python coding styles and practices. This limitation is particularly evident in basic code constructs like addition (Image 8.1) and palindrome checks (Image 8.3), which are foundational for generating more complex code.

Retraining the model on certain good-quality data also provided better results and is a domain to be explored further as it provides a factor of recurrency and temporal importance due to its recency aspects.

### 9.4 Evaluation Metrics

The use of the BLEU score as the primary evaluation metric presents a limitation in assessing the functional accuracy of the generated code. BLEU scores primarily measure linguistic alignment rather than executable correctness. To obtain a more comprehensive evaluation, additional metrics such as manual human code reviews or automated code execution tests could be employed.

---

## 10. Research Paper Review

This project was greatly inspired by the paper: CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Transformer Model for Code Understanding and Generation. CodeT5 is built on T5 architecture, 220M tokens, trained specifically on working with Natural Language (NL) and Programming Language (PL). It was pre-trained on Identifier-aware Denoising Pre-training, Identifier Tagging (IT), Masked Identifier Prediction and Bimodal Dual Generation. Fine-tuning was done on tasks such as Task-specific Transfer Learning: Generation vs. Understanding Tasks and Multi-task Learning. It performs well on tasks such as code defect detection, code search, and code generation. CodeT5 does a great job at capturing semantic information from code due to the proposed identifier-aware pre-training (like function names and variables, which are crucial in code and carry a lot of meaning) and bimodal dual generation for code tasks. It gives state-of-the-art results on 14 tasks of the CodexGLUE subtasks.

CodeT5 was trained on a mix of programming language (PL) data with and without

natural language (NL) descriptions, gathering around 8.35 million instances. Various programming languages were included like Ruby, JavaScript, Go, Python, Java, PHP, C, and CSharp. To understand code better, they used identifiers (like function names) and tokenization methods to break down code into manageable parts.

---

## 11. Conclusion

In this project, we explored automating Python code generation from natural language descriptions. The experiments yielded promising results, especially after fine-tuning, but also highlighted challenges in model generalization and dataset limitations. The experience highlighted the importance of a diverse and comprehensive dataset and the pivotal role of computational resources in training large language models. This journey has advanced our technical understanding and helped us apply the knowledge we gained throughout the semester to a real-world problem.

---

## 12. Contributions

Abhinav was responsible for data collection and pre-processing. Devyani and Neeti created the baseline model, did the comparative analysis of pre-trained models, performed experiments for fine-tuning, and documented the project.

---

## 13. References


1. Code Parrot Dataset https://huggingface.co/datasets/codeparrot/xlcost-text-to-code/viewer/Python-program-level
2. Semuru's Text-Code Galeras Dataset https://huggingface.co/datasets/semeru/text-code-galeras-code-generation-from-docstring-3k-deduped?row=4
3. Mostly Basic Python Programming Dataset https://huggingface.co/datasets/mbpp
4. PyTorrent Dataset and Paper https://github.com/fla-sil/PyTorrent
5. CodeT5 Paper https://arxiv.org/abs/2109.00859v1
6. Mixed Precision Training https://pytorch.org/docs/stable/notes/amp_examples.html
7. CodeT5-base Model https://huggingface.co/Salesforce/codet5-base
8. https://www.nltk.org/_modules/nltk/translate/bleu_score.html
9. GPT2 Model https://huggingface.co/gpt2
10. PyCode GPT Model https://github.com/microsoft/PyCodeGPT
11. CodeGen2-1B Model https://huggingface.co/Salesforce/codegen2-1B
12. CodeT5-small Model https://huggingface.co/Salesforce/codet5-small