CodeCraftAI : Code Generation for Natural Language Descriptions

by

Vishwa Patel (vup4)

at

Department of Statistics, Rutgers University

Fall 2023

Abstract

In the realm of programming, accessibility and efficiency are paramount. Our project embarks on the ambitious journey of developing a text-to-code generation system, with a fundamental objective – to seamlessly translate natural language descriptions into executable code snippets. The driving force behind this initiative is the desire to empower individuals with limited coding experience, providing them with a more intuitive avenue for expressing their ideas and realizing their computational concepts.

Introduction

At its core, our project engages in a sophisticated Natural Language Processing (NLP) task that revolves around sequence-to-sequence transformation. This transformative process involves the conversion of human-readable descriptions, akin to the way humans communicate in everyday language, into precise and executable code sequences. In essence, it is a form of translation, strategically bridging the gap between the nuanced understanding of natural language and the intricate synthesis of functional code.

By undertaking this task, our project not only delves into the technical intricacies of developing a text-to-code generation system but also contributes to the democratization of programming knowledge. The envisioned system aims to make coding more accessible to a broader audience, fostering a symbiotic relationship between natural language expression and the structured syntax of programming languages. Through this synthesis, our project aspires to create a tool that empowers individuals to articulate their ideas in a language they understand while seamlessly translating them into functional and executable code.

Dataset

In selecting the CodeSearchNet corpus—a compilation of 2 million (comment, code) pairs from GitHub (Husain et al., 2019)—our project harnesses rich diversity across programming languages. Focusing specifically on Python, known for its readability, curating a subset of 412,178 records, spanning various complexity levels. This deliberate choice transforms our dataset into a robust training ground, ensuring our model's adaptability to diverse coding challenges.

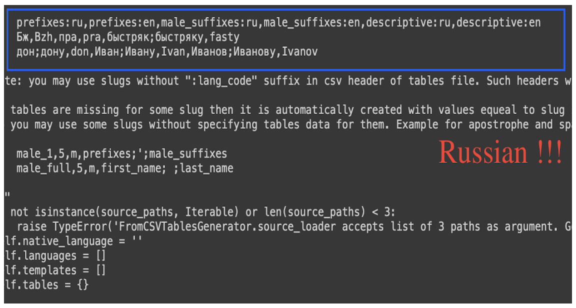
By centering on Python's relevance and inclusivity, CodeSearchNet becomes more than a dataset; it evolves into the crucible where natural language meets code synthesis, propelling us towards a more empowered programming landscape.

Preprocessing: Enhancing Dataset Quality

Non-English Descriptions:

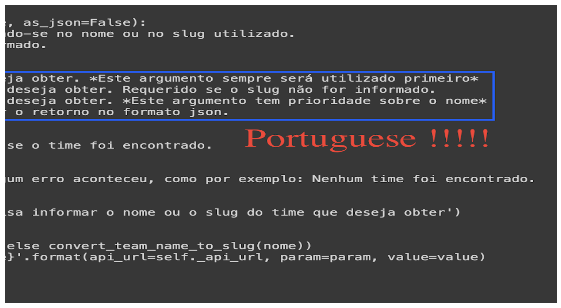
- Challenge: A significant issue surfaced with non-English code descriptions shown in *Figures 1.1 & 1.2*, including languages like Russian and Portuguese.

- Solution: To ensure a consistent focus on the English language, introducing the `removeNonEnglish()` function. This step was crucial for our text-to-code generation system's effectiveness, aligning with the models' training on English natural language descriptions.



```
 prefixes:ru,prefixes:en,male_suffixes:ru,male_suffixes:en,descriptive:ru,descriptive:en
 Бж,Bzh,пра,pra,быстряк;быстряку,fasty
 дон;дону,don,Иван;Ивану,Ivan,Иванов;Иванову,Ivanov

te: you may use slugs without ":lang_code" suffix in csv header of tables file. Such headers w

 tables are missing for some slug then it is automatically created with values equeal to slug
 you may use some slugs without specifying tables data for them. Example for apostrophe and sp

  male_1,5,m,prefixes;';male_suffixes
  male_full,5,m,first_name; ;last_name

"
 not isinstance(source_paths, Iterable) or len(source_paths) < 3:
  raise TypeError('FromCSVTablesGenerator.source_loader accepts list of 3 paths as argument. G
lf.native_language = ''
lf.languages = []
lf.templates = []
lf.tables = {}
```

Figure 1.1

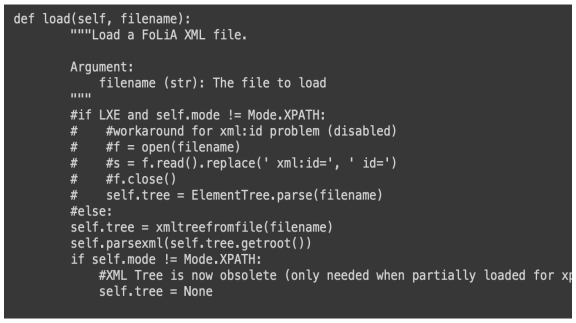


```
, as_json=False):
do-se no nome ou no slug utilizado.
mado.

ja obter. *Este argumento sempre será utilizado primeiro*
deseja obter. Requerido se o slug não for informado.
deseja obter. *Este argumento tem prioridade sobre o nome*
 o retorno no formato json.

se o time foi encontrado.

um erro aconteceu, como por exemplo: Nenhum time foi encontrado.

sa informar o nome ou o slug do time que deseja obter')

else convert_team_name_to_slug(nome))
}'.format(api_url=self._api_url, param=param, value=value)
```

Figure 1.2

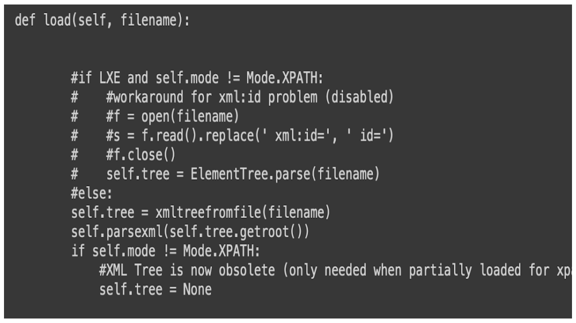
Redundancy in Code Descriptions:

- Challenge: The dataset contained redundancy due to code descriptions within code blocks as shown in *Figure 1.3*.
- Solution: Implementing the removeDocStrings() function addressed this challenge by removing redundant code descriptions shown in *Figure 1.4*. This streamlined the dataset, contributing to a more focused and efficient training process for our models.



```
def load(self, filename):
        """Load a FoLiA XML file.

        Argument:
            filename (str): The file to load
        """
        #if LXE and self.mode != Mode.XPATH:
        #    #workaround for xml:id problem (disabled)
        #    #f = open(filename)
        #    #s = f.read().replace(' xml:id=', ' id=')
        #    #f.close()
        #    self.tree = ElementTree.parse(filename)
        #else:
        self.tree = xmltreefromfile(filename)
        self.parsexml(self.tree.getroot())
        if self.mode != Mode.XPATH:
            #XML Tree is now obsolete (only needed when partially loaded for xp
            self.tree = None
```

Figure 1.3



```
def load(self, filename):


        #if LXE and self.mode != Mode.XPATH:
        #    #workaround for xml:id problem (disabled)
        #    #f = open(filename)
        #    #s = f.read().replace(' xml:id=', ' id=')
        #    #f.close()
        #    self.tree = ElementTree.parse(filename)
        #else:
        self.tree = xmltreefromfile(filename)
        self.parsexml(self.tree.getroot())
        if self.mode != Mode.XPATH:
            #XML Tree is now obsolete (only needed when partially loaded for xp
            self.tree = None
```

Figure 1.4

Tokenization

In quest for a seamless transition from natural language to executable code, the tokenization process emerges as a pivotal step. Tokenization is the art of disassembling raw text into distinct units—words or subwords—crucial for enhancing the interpretability of our models in the context of specific tasks. Explored various tokenization strategies to comprehend the optimal approach for code sequence representation.

Whitespace Tokenization:

- Although a basic approach, whitespace tokenization is fundamental for establishing a baseline understanding. However, it falls short in capturing the intricate structures inherent in coding, prompting the exploration of more sophisticated methods.

Character Tokenization:

- Tokenizing at the character level provides granularity in representation. Despite its merits, this method might struggle with contextual understanding, urging us to delve deeper for a more nuanced comprehension of code nuances.

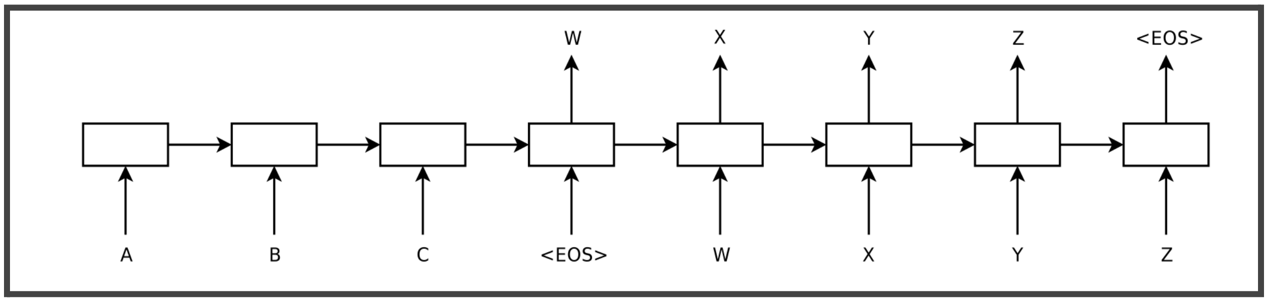
Abstract Syntax Tree (AST) Tokenization:

- AST tokenization is our chosen method due to its ability to preserve code structure, decode semantic nuances, handle complex expressions, and offer a granular representation. This meticulous approach aligns seamlessly with the intricacies of our text-to-code generation task.

Training Seq2Seq Model

In crafting our model architecture for transforming natural language docstrings into executable code snippets, as it is at base a text-to-text task , thus drew inspiration from the Sequence-to-Sequence (Seq2Seq) paradigm, as introduced by Sutskever, Vinyals, and Le in their pivotal work “Sequence to Sequence Learning with Neural Networks-2014”.

Model Architecture

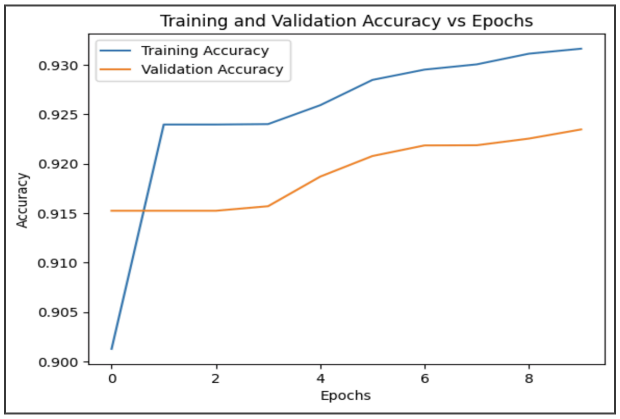


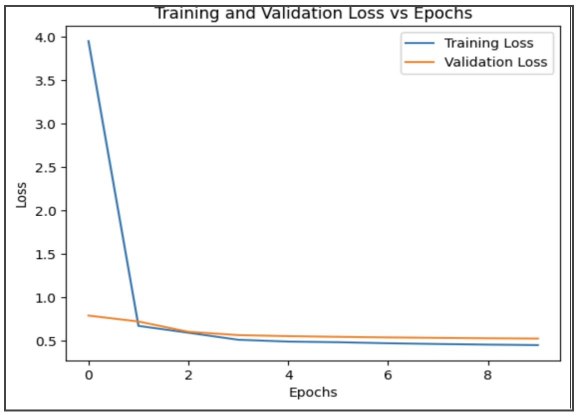
Seq2Seq Encoder and decoder

- Encoder:
  - Ingests natural language docstrings as input.
  - Applies an embedding layer to convert words into dense vectors.
  - Incorporates LSTM (Long Short-Term Memory) units to capture contextual information and generate encoded representations.
  - Produces the final encoder states.
- Decoder:
  - Takes code snippets as input during training.
  - Utilizes a distinct embedding layer for code tokens.
  - Employs LSTM units for sequence generation, leveraging the contextual information from the encoder states.
  - Generates sequences of code tokens through a dense layer with softmax activation.

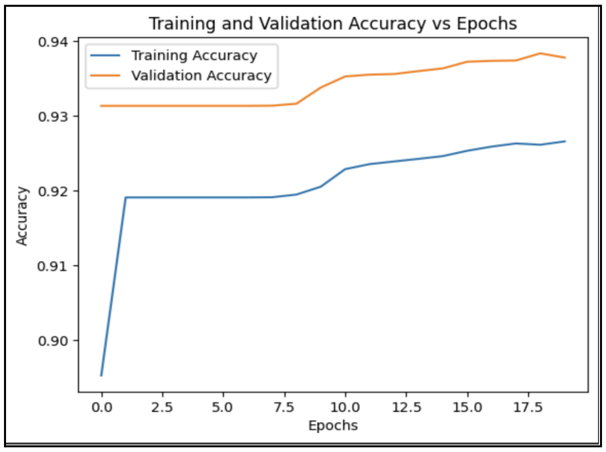
- Model Compilation:
    - Employs the Adam optimizer with a learning rate schedule to optimize the model via gradient-based optimization.
    - Used Sparse categorical cross entropy is a more memory-efficient loss function than categorical cross entropy when the number of classes (word_index) is large, as it avoids the need to create one-hot encoded vectors which become computationally heavy.
    - Mathematically, both loss functions compute the same cross-entropy loss. The only difference is in the input format of the target labels.

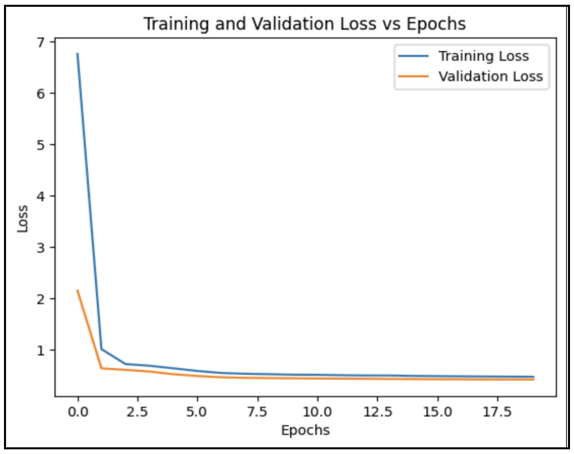
Batch Size : 8 ; Epochs : 10 training graphs





Batch Size : 16 ; Epochs : 20 training graphs





Addressing Documentation-Code Semantic Misalignment

As Documentation and code have different vocabularies and semantic meanings, the model has two different vocabularies. The words or tokens used in documentation might not perfectly align with the vocabulary of code. For example, programming keywords, identifiers, and syntax elements present in code may not be relevant or even exist in the natural language used in documentation.

Teacher Forcing and Data Preparation for Training

- Teacher Forcing Implementation:
  - Implemented the teacher forcing technique during training to stabilize and expedite the learning process. This involved feeding the actual or expected output from the training dataset as input to the decoder instead of using the predicted output from the previous time step.
- Decoder Data Preparation:
  - Reshaped the input data for a sequence-to-sequence model by shifting the target sequence one position to the right. It initializes a matrix with zeros and sets the first column to the index corresponding to the <start> token, indicating the beginning of the decoding process.

Challenges and Hurdles in Seq2Seq Model Implementation

- Tokenization Dilemmas:
  - Issue: Inconsistent or improper tokenization practices and a limited vocabulary size lead to the presence of out-of-vocabulary words, risking the loss of crucial information.
  - Impact: This challenge impeded the model's understanding of both natural language docstrings and code snippets, affecting the quality of generated sequences.
- Insufficient Training Data:
  - Issue: Training the Seq2Seq model on insufficient data hindered its ability to capture diverse patterns and variations present in the dataset, resulting in poor generalization and unexpected outputs on unseen data.

    - Impact: The model exhibits signs of overfitting, memorizing specific training examples rather than learning the underlying patterns, leading to reduced performance on new and unseen data.
- Inherent Complexity of text2code Task:
    - Issue: The text-to-code task is inherently complex, posing challenges for the model to effectively learn intricate patterns and dependencies between natural language descriptions and corresponding code structures.
    - Impact: The model's capacity to generalize across different complexities and coding styles was compromised, affecting its overall performance.
- Absence of Attention Mechanism:
    - Issue: Lack of attention mechanisms in the Seq2Seq model resulted in difficulties capturing long-range dependencies between words in both docstrings and code blocks.
    - Impact: The model's ability to understand and incorporate contextual information from distant parts of the input sequences was limited, affecting the accuracy of generated code.

Addressing these challenges was critical for enhancing the robustness and performance of Seq2Seq models in the context of translating natural language docstrings into executable code blocks.

Addressing Challenges: Fine Tuning T5 Small

Model Overview

To surmount the challenges encountered in Seq2Seq model implementation, transitioning to a state-of-the-art solution — fine tuning the T5 Small model. T5, developed by Google Research, is a versatile Text-to-Text framework renowned for its unified approach to various Natural Language Processing (NLP) tasks.

Fine Tuning Specifications

- Model Choice:
    - T5 Small: Specifically selected for its computational efficiency, T5 Small strikes a balance between performance and resource requirements. With 60 million parameters, it offers a streamlined yet potent architecture.

- Adaptability:
    - Fine-Tunable: T5 Small is fine-tunable for specific tasks with limited data. This adaptability enables us to tailor the pre-trained model to the nuances of our text-to-code generation challenge.

- Architecture Details:
    - Parameters: T5 Small boasts 60 million parameters, contributing to its ability to understand complex relationships in sequences.
    - Encoder and Decoder Layers: Comprising 6 layers each, the model employs 128 hidden dimensions and 8 attention heads, ensuring a comprehensive grasp of input contexts.

- Training Specifications:
    - Learning Rate: Set at 2e-5, the learning rate is optimized to guide effective model adaptation during fine-tuning.
    - Batch Size: A batch size of 8 is employed for both training and evaluation phases.
    - Weight Decay: At 0.01, weight decay is configured to regulate model parameters during training.
    - Training Epochs: The model undergoes three training epochs to refine its understanding of the task, with each iteration providing insights into its evolving capabilities.

- Data Input Variations:
    - Increasing Data Input: The fine tuning process is executed in multiple iterations, progressively scaling the training data from 1,500 to 5,000, and finally to 25,000 rows. This systematic approach allowed our project to assess the model's performance under varying data volumes.

Through the finetuning of the T5 Small model, aiming to leverage its pre-trained knowledge on diverse language tasks to overcome the limitations posed by insufficient data and complex Seq2Seq patterns. This strategic shift promises enhanced adaptability, improved attention mechanisms, and a more nuanced understanding of the relationships between natural language descriptions and executable code.

Results

Performance metrics:

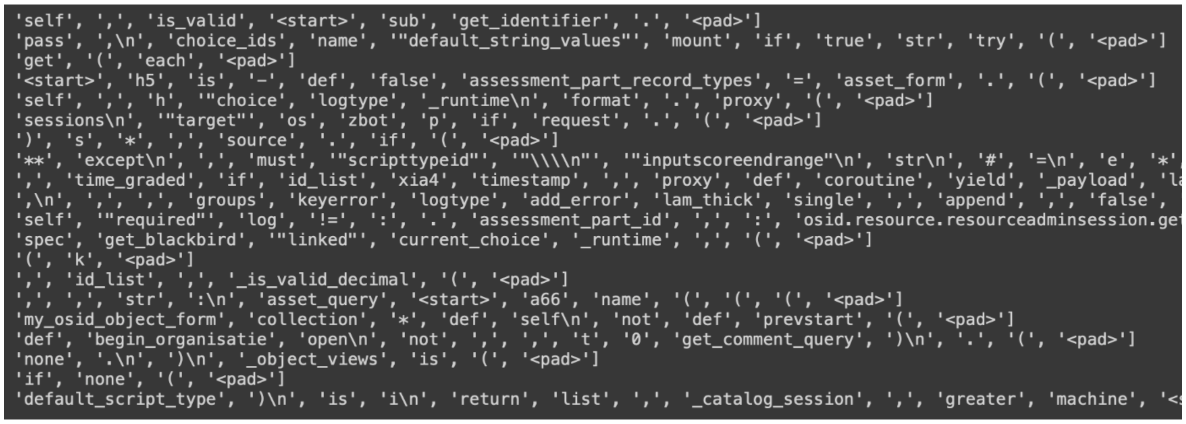
- BLEU Score (Precision)
    - Evaluates how closely the machine-generated code matches the reference (ideal) code.
    - Compares the n-gram (group of words) overlap between the generated code and a set of high-quality reference codes.
- Compilation and Execution
    - See if the generated code is compilable and executable

Performance

Model	**Dataset Size**	**BLEU SCORE**	**Compilation & Execution**	**Makes a good start**
Seq2Seq	1.5k rows	0.26	NO	NO
	2.5k rows	0.32	NO	NO
	5k rows	0.30	NO	NO
T5(fine tuned)	1.5k rows	0.002	NO	NO
	5k rows	0.069	NO	YES
	25k rows	0.091	NO	YES

Performance comparison table

Seq2Seq outputs:



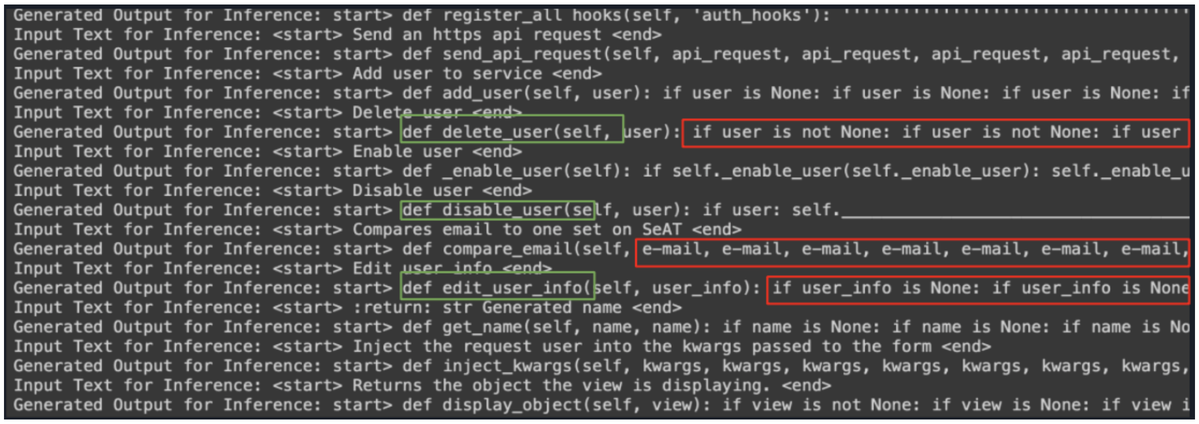
```
'self', ',', 'is_valid', '<start>', 'sub', 'get_identifier', '.', '<pad>']
'pass', ',\n', 'choice_ids', 'name', '"default_string_values"', 'mount', 'if', 'true', 'str', 'try', '(', '<pad>']
'get', '(', 'each', '<pad>']
'<start>', 'h5', 'is', '-', 'def', 'false', 'assessment_part_record_types', '=', 'asset_form', '.', '(', '<pad>']
'self', ',', 'h', '"choice', 'logtype', '_runtime\n', 'format', '.', 'proxy', '(', '<pad>']
'sessions\n', '"target"', 'os', 'zbot', 'p', 'if', 'request', '.', '(', '<pad>']
')', 's', '*', ',', 'source', '.', 'if', '(', '<pad>']
'**', 'except\n', ',', 'must', '"scripttypeid"', '"\\\\n"', '"inputscoreendrange"\n', 'str\n', '#', '=\n', 'e', '*'
',', 'time_graded', 'if', 'id_list', 'xia4', 'timestamp', ',', 'proxy', 'def', 'coroutine', 'yield', '_payload', 'la
',\n', ',', ',', 'groups', 'keyerror', 'logtype', 'add_error', 'lam_thick', 'single', ',', 'append', ',', 'false',
'self', '"required"', 'log', '!=', ':', '.', 'assessment_part_id', ',', ':', 'osid.resource.resourceadminsession.get
'spec', 'get_blackbird', '"linked"', 'current_choice', '_runtime', ',', '(', '<pad>']
'(', 'k', '<pad>']
',', 'id_list', ',', '_is_valid_decimal', '(', '<pad>']
',', ',', 'str', ':\n', 'asset_query', '<start>', 'a66', 'name', '(', '(', '(', '<pad>']
'my_osid_object_form', 'collection', '*', 'def', 'self\n', 'not', 'def', 'prevstart', '(', '<pad>']
'def', 'begin_organisatie', 'open\n', 'not', ',', ',', 't', '0', 'get_comment_query', ')\n', '.', '(', '<pad>']
'none', '.\n', ')\n', '_object_views', 'is', '(', '<pad>']
'if', 'none', '(', '<pad>']
'default_script_type', ')\n', 'is', 'i\n', 'return', 'list', ',', '_catalog_session', ',', 'greater', 'machine', '<
```

Generated outputs of seq2seq model

Interpretation:

Good BLEU score , but the generated code doesn't make sense because of the problems mentioned in the Challenges and hurdles section of Seq2Seq Implementation.

T5 small outputs:



```
Generated Output for Inference: start> def register_all hooks(self, 'auth_hooks'): ''''''''''''''''''''''''''''''''
Input Text for Inference: <start> Send an https api request <end>
Generated Output for Inference: start> def send_api_request(self, api_request, api_request, api_request, api_request,
Input Text for Inference: <start> Add user to service <end>
Generated Output for Inference: start> def add_user(self, user): if user is None: if user is None: if user is None: if
Input Text for Inference: <start> Delete user <end>
Generated Output for Inference: start> def delete_user(self, user): if user is not None: if user is not None: if user
Input Text for Inference: <start> Enable user <end>
Generated Output for Inference: start> def _enable_user(self): if self._enable_user(self._enable_user): self._enable_u
Input Text for Inference: <start> Disable user <end>
Generated Output for Inference: start> def disable_user(self, user): if user: self.________________________________
Input Text for Inference: <start> Compares email to one set on SeAT <end>
Generated Output for Inference: start> def compare_email(self, e-mail, e-mail, e-mail, e-mail, e-mail, e-mail, e-mail,
Input Text for Inference: <start> Edit user info <end>
Generated Output for Inference: start> def edit_user_info(self, user_info): if user_info is None: if user_info is None
Input Text for Inference: <start> :return: str Generated name <end>
Generated Output for Inference: start> def get_name(self, name, name): if name is None: if name is None: if name is No
Input Text for Inference: <start> Inject the request user into the kwargs passed to the form <end>
Generated Output for Inference: start> def inject_kwargs(self, kwargs, kwargs, kwargs, kwargs, kwargs, kwargs, kwargs,
Input Text for Inference: <start> Returns the object the view is displaying. <end>
Generated Output for Inference: start> def display_object(self, view): if view is not None: if view is None: if view i
```

Generated outputs of T5 fine tuned model

Interpretation:

Lower score however more relevant code tokens as it succeeds in correctly defining the function with appropriate name, but still not executable as code tokens start repeating after that.

Conclusion: Unveiling Insights and Future Paths

In the journey to bridge the gap between natural language docstrings and executable code blocks, our projects' exploration into Seq2Seq and T5 Small models has yielded valuable insights, shaping the way forward for text-to-code generation. The nuanced evaluation of both models has illuminated critical aspects that demand our attention and innovative solutions.

Seq2Seq Model Reflections:

Seq2Seq model, marked by a commendable BLEU score, showcases the limitations inherent in simplistic approaches. Despite achieving a high precision metric, the generated code often lacks coherence and logical structure. The challenges outlined in the hurdles section, including tokenization inconsistencies and the absence of an attention mechanism, unveil the intricacies of code generation.

T5 Small's Achievements:

Focusing on T5 Small, the story unfolds differently. While the BLEU score might be comparatively lower, the model excels in generating more contextually relevant code tokens. This is particularly evident in its adeptness at defining functions with appropriate names. However, the repetitive nature of code tokens after a certain point hints at the nuances that persist. It's not just about the score; it's about the meaningful representation of code structures that T5 Small strives to achieve.

Insights:

Our findings echo a resounding truth – simplicity is not the key to success in the intricate realm of code generation. More nuanced and fine-tuned approaches are imperative, as evidenced by the enhanced performance of the T5 model with increased training data. The inadequacy of current metrics to holistically capture the essence of generated tokens highlights the complexity of evaluating the synergy between natural language and code synthesis.

Compilation and Execution Imperatives:

The journey doesn't conclude with a BLEU score; it extends to the vital realms of compilation and execution. Meaningful code requires not just precision but practical viability, a domain where both models face challenges. This poses a clarion call for future investigations into enhancing the practical applicability of generated code snippets.

The Power of Transfer Learning:

Amidst the challenges, the application of transfer learning, particularly in the fine-tuning of T5 Small, emerges as a beacon of promise. Harnessing the knowledge embedded in pre-trained models allows us to navigate the hurdles posed by insufficient data and complex task requirements. Transfer learning becomes the catalyst for model refinement, transforming potential pitfalls into opportunities for growth.

In conclusion, our journey illuminates a path forward – one that embraces the intricacies of code synthesis, recognizes the limitations of simplistic models, and champions the potential of transfer learning. Navigating this ever-evolving landscape, each insight becomes a stepping stone towards a more empowered and nuanced approach to text-to-code generation. The quest continues, fueled by the curiosity to unravel the complexities of code in the language of natural expression and the promise shown by the fine-tuning approach.

Acknowledgements: Individual Contributions

1. Team Member : Akshay Amarnath Mudda (am3364@rutgers.edu):

- Dataset Selection: Spearheaded the selection of the CodeSearchNet corpus, emphasizing its rich diversity and relevance to the Python programming language.

- Dataset Curation: Led the curation of a Python subset, focusing on readability and complexity levels, ensuring the dataset's suitability for training the text-to-code generation models.

- Preprocessing Implementation: Implemented the `removeNonEnglish()` & `removeDocStrings()` function to address the challenge of non-English descriptions and redundancy inturn enhancing the dataset's effectiveness for English-based training.

2. Team Member : Vishwa Upendra Patel (vup4@rutgers.edu):

- Tokenization Strategies: Explored and evaluated various tokenization strategies, including whitespace, character, and Abstract Syntax Tree (AST) tokenization, contributing to the understanding of optimal code sequence representation.

- Seq2Seq Model Training: Involved in crafting and training the Seq2Seq model, contributing to the development of the model architecture and addressing challenges related to tokenization and insufficient training data.

- Challenges Section: Played a key role in identifying and articulating the challenges faced during Seq2Seq model implementation, offering insights into tokenization dilemmas and the model's struggles with inherent complexities.

3. Team Member : Atharva Hajare (ah1377@rutgers.edu):

- Fine-Tuning T5 Small Model: Led the transition to fine-tuning the T5 Small model as a strategic response to challenges faced by the Seq2Seq model, contributing to the model overview and fine-tuning specifications.

- Results Section: Contributed to the performance metrics, compilation and execution analysis, and insights in the Results section, offering a comprehensive evaluation of both Seq2Seq and T5 Small model outputs.

- Conclusion Section: Played a pivotal role in synthesizing key insights and outlining the significance of transfer learning, shaping the concluding remarks on the future paths for text-to-code generation.

This collaborative effort showcases the diverse skill sets and contributions of each team member, culminating in a comprehensive and insightful exploration of text-to-code generation using Seq2Seq and T5 Small models.

References:

● Husain, M. I., Wu, H., Gazit, T., Allamanis, M., Brockschmidt, M., & Gaunt, A. (2019). CodeSearchNet: A Dataset and Benchmarks for Semantic Code Search. arXiv:1909.09436 [cs.LG].

● Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Advances in Neural Information Processing Systems (NeurIPS).arXiv:1409.3215 [cs.CL]

● Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, arXiv:1910.10683[cs.LG]