

A
Major Project Report
on
**Natural Language to Code: Enhancing Source Code Generation through GPT
Integration**

submitted in partial fulfilment of the requirements for the award of the Degree of
Bachelor of Technology

By
Shaista Firdous
(20EG105442)
M. Sathvika Reddy
(20EG105428)
K. Shiva Sai
(19H61A05L6)



Under The Guidance
Of

Dr. K. Madhuri
Associate Professor,
Department of CSE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
ANURAG UNIVERSITY
VENKATAPUR (V), GHATKESAR (M), MEDCHAL (D), T.S 500088
(2023-24)

DECLARATION

We hereby declare that the Report entitled **Natural Language to Code: Enhancing Source Code Generation through GPT Integration** submitted for the award of Bachelor of technology Degree is our original work and the Report has not formed the basis for the award of any degree, diploma, associateship or fellowship of similar other titles. It has not been submitted to any other University or Institution for the award of any degree or diploma.

Place: *Anurag University, Hyderabad*

Shaista Firdous

Date:

(20EG105442)

M. Sathvika

(20EG105428)

K. Shiva Sai

(19H61A05L6)



CERTIFICATE

This is to certify that the report entitled **Natural Language to Code: Enhancing Source Code Generation through GPT Integration** being submitted by **Ms. Shaista Firdous** bearing the hall ticket number **20EG105442**, **Ms. M. Sathvika** bearing the hall ticket number **20EG105428**, **Mr. K. Shiva Sai** bearing the hall ticket number **19H61A05L6** in partial fulfilment for the award of B.Tech degree in Computer Science and Engineering to Anurag University is a record of bonafide work carried out by them under my guidance and supervision.

The results embodied in this report have not been submitted to any other University or Institute for the award of any degree or diploma.

Dr. K. Madhuri
Associate Professor
Department of CSE

Dr. G. Vishnu Murthy
Dean, CSE

External Examiner

ACKNOWLEDGEMENT

We would like to express our sincere thanks and deep sense of gratitude to project supervisor **Dr. K. Madhuri, Associate Professor, Department of CSE** for her constant encouragement and inspiring guidance without which this project could not have been completed. Her critical reviews and constructive comments improved our grasp of the subject and steered to the fruitful completion of the work. Her patience, guidance and encouragement made this project possible.

We would like express my special thanks to **Dr. V. Vijaya Kumar, Dean School of Engineering, Anurag University**, for his encouragement and timely support in our B. Tech program.

We would like to acknowledge our sincere gratitude for the support extended by **Dr. G. Vishnu Murthy, Dean, Dept. of CSE, Anurag University**. We also express my deep sense of gratitude to **Dr. VVSSS Balaram, Academic co-ordinator, Anurag University** and **Dr. Pallam Ravi, Project in-Charge**. Project Co-ordinator and Project review committee members, whose research expertise and commitment to the highest standards continuously motivated us during the crucial stage our project work.

Shaista Firdous
(20EG105442)

M. Sathvika Reddy
(20EG105428)

K. Shiva Sai
(19H61A05L6)

ABSTRACT

In today's fast-paced technological landscape, the demand for efficient software development processes has reached unprecedented levels. Traditional methods of manual coding often struggle to keep up with the rapid pace and complexity of modern projects. Recognizing these challenges, there has been a significant surge in interest in leveraging artificial intelligence (AI) and natural language processing (NLP) technologies to automate various aspects of the coding process. This project represents a pioneering exploration into the realm of AI-driven code generation, with the formidable GPT-3 Transformer model serving as its nucleus. By harnessing the capabilities of this advanced AI model and leveraging proprietary data, the project aims to decode the intricacies of programming languages and generate precise code snippets based on contextual cues. The essence of this innovative approach lies in its ability to significantly enhance software development performance. Developers are empowered with tools that enable them to swiftly prototype, debug, and craft code with unparalleled efficiency and accuracy. By automating repetitive and time-consuming coding tasks, developers can redirect their focus towards higher-level problem-solving and innovation. The implications of this research extend far beyond mere efficiency gains. By unlocking the potential of artificial intelligence in software engineering, we are not just optimizing existing processes; we are reshaping the very fabric of how software is conceived, developed, and deployed. This journey represents a transformative leap towards a future where the creative boundaries of business and technology converge, paving the way for unprecedented advancements in software engineering.

Keywords: GPT-3, transformer model, code generation, fine-tuning, natural language processing, machine learning.

TABLE OF CONTENTS

S. NO.	CONTENT	PAGE NO.
1.	Introduction	1
	1.1. Generative pre-trained transformers	3
	1.2. Language Translation Model	6
	1.3. Motivation	7
	1.4. Problem Definition	8
	1.5. Problem Illustration	9
	1.6. Objective of the Project	10
2.	Literature Document	11
3.	Source Code generation throughout GPT Integration	16
	3.1. Fine-tuning the model	17
	3.2. Generating source codes using fine-tuned model	21
	3.3. Generating Algorithm for the user's query	22
	3.4. Creating a language translation model	23
4.	Implementation	28
	4.1. Functionality	28
	4.2. Attributes	32
	4.3. Experimental Screenshot	34
5.	Experimental Setup	36
	5.1. Obtain OpenAI API key	39
	5.2. Setup Jupyter Notebook	36
	5.3. Setup Streamlit	37
	5.4. Libraries used	38
	5.5. Parameters	40
6.	Discussion of Results	41
7.	Conclusion	48
8.	References	49

LIST OF FIGURES

Figure No.	Figure Name	Page No.
1.1	Natural Language Processing	2
1.2	Transformer Model	4
1.3	Translation Model	7
1.4	Problem Illustration	9
2.1	Pre-trained Large Language Model	12
2.2	Code Generation Model	14
3.1	Training GPT-3 transformer model	17
3.2	Stages of fine-tuning a model	18
3.3	Training data	19
3.4	GPT-3 transformer model	21
3.5	Language Translation Model	27
4.1	Fine-tuning the model	30
4.2	Language Translation Model	31
4.3	Training the language translation model	34
4.4	User Interface	35
6.1	Precision Score Comparison	42
6.2	Accuracy Score Comparison	43
6.3	Rouge Score Comparison	45
6.4	BLEU Score Comparison	47

LIST OF TABLES

Table No.	Table Name	Page No.
6.1	Average precision scores of different models	42
6.2	Accuracy scores of different models	43
6.3	Rouge Scores of different models	45
6.4	BLEU Scores of different models	46

LIST OF ABBREVIATIONS

Abbreviations	Full Form
GPT	Generative Pre-trained Transformer
NLP	Natural Language Processing
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
API	Application Program Interface
NLI	Natural Language Interface
XML	Extensible Markup Language
LLM	Large Language Model

1. INTRODUCTION

Source code serves as the bedrock upon which computer programs are built, providing a set of human-readable instructions that define the behaviour and functionality of software applications. These instructions adhere to specific syntax, rules, and conventions dictated by the programming language being used. Typically authored by skilled programmers or developers, source code undergoes rigorous scrutiny to ensure precision, adherence to specifications, and compatibility with the target platform.

Despite the critical role source code plays in software development, the process of writing and understanding it can be complex and time-consuming, particularly for those less experienced with programming languages. This is where Natural Language Interface (NLI) steps in, offering an alternative input method that allows users to interact with computers using spoken or written human language, such as English, rather than through conventional interfaces like graphical user interfaces (GUIs) or command-line interfaces (CLIs).

By leveraging NLI technology, the proposed system aims to bridge the gap between human language and source code generation, enabling professionals to express their programming requirements in natural language rather than grappling with the intricacies of programming syntax. This approach holds immense potential for enhancing productivity and efficiency, as it streamlines the process of translating high-level concepts and requirements into executable code.

Furthermore, by abstracting away the complexities of programming languages, NLI empowers individuals with varying levels of technical expertise to engage in software development tasks. Even those with limited programming knowledge can articulate their intentions and requirements in plain language, allowing the system to generate the corresponding source code automatically.

Moreover, the use of NLI in source code generation facilitates rapid prototyping and iteration, as developers can quickly experiment with different functionalities and features without being encumbered by the need to manually write and debug code. This iterative approach fosters creativity and innovation, enabling teams to explore diverse solutions and refine their designs more efficiently.

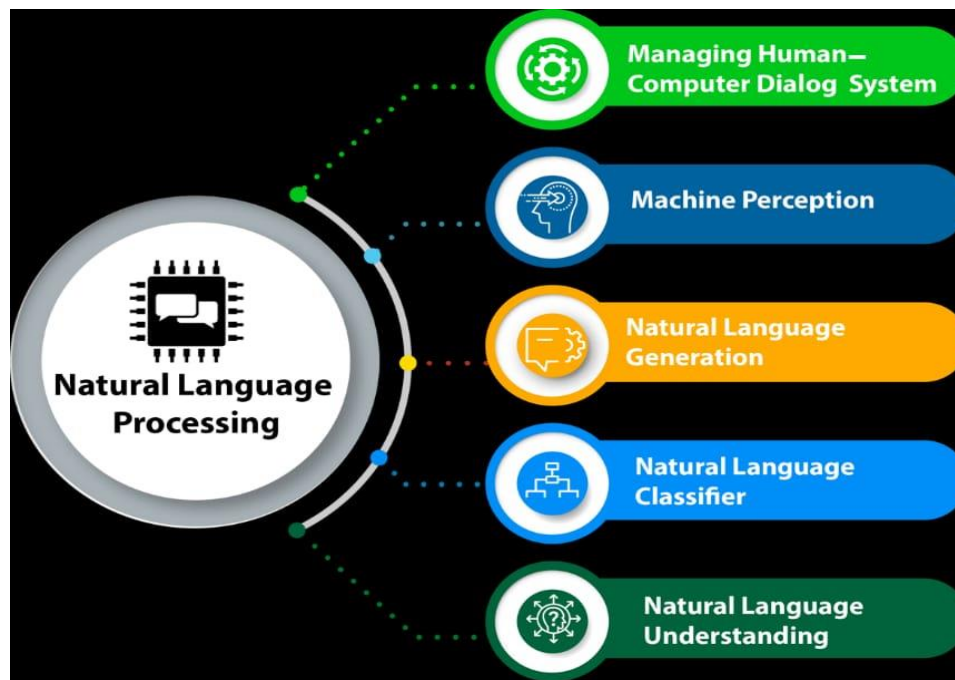


Figure 1.1 Natural Language Processing

As writing source-code is becoming more widespread and complicated, it is becoming an essential to automate writing simpler source-code by AIs to save time in writing, learning and understanding source-code. With Natural Language Processing (NLP) getting better and simple with each year and demand for writing new and complex source code is increasing every year , it was inevitable for these two fields to join. There are many natural language interfaces that connect NLP with databases but not many programs that connect NLP with computer languages. Creating an Natural Language Interface that helps programmers to create source code will reduce the time it takes for them to write source code as they will refer to complex documentation less frequently and will reduce the bar to enter the world of programming language.

- NLP stands for Natural Language Processing. It's a field of artificial intelligence that focuses on the interaction between computers and humans using natural language. NLP enables machines to understand, interpret, and generate human language, facilitating communication between computers and people.
- Incorporating GPT into source code generation elevates project efficiency by enabling intuitive natural language commands for code creation and refinement.

In this project, we aim to improve source code generation by integrating GPT into the process. By leveraging natural language, our goal is to enhance the efficiency and

accuracy of code creation, fostering a more intuitive development experience. This integration seeks to bridge the gap between human-like language understanding and code synthesis, ultimately streamlining the software development workflow.

1.1 Generative pre-trained transformers (GPT):

Generative pre-trained transformers (GPT) are a type of large language model (LLM) and a prominent framework for generative artificial intelligence. They are artificial neural networks that are used in natural language processing tasks. GPTs are based on the transformer architecture, pre-trained on large data sets of unlabeled text, and able to generate novel human-like content. As of 2023, most LLMs have these characteristics and are sometimes referred to broadly as GPTs.

The GPT (Generative Pre-trained Transformer) model stands as a pinnacle of natural language processing (NLP) and artificial intelligence (AI) research, representing a transformative advancement in machine learning. Built upon the Transformer architecture, GPT leverages self-attention mechanisms to capture intricate semantic relationships within text data, enabling it to generate coherent and contextually relevant outputs. Through pre-training on vast corpora of text data, GPT learns to understand and generate human-like language, exhibiting remarkable fluency and versatility across a wide range of tasks, from language translation and summarization to question answering and code generation. Its ability to grasp the nuances of language and generate coherent responses has revolutionized numerous applications, empowering developers, researchers, and businesses to harness the power of AI in unprecedented ways. As a cornerstone of modern NLP, GPT continues to push the boundaries of what is possible in natural language understanding and generation, heralding a new era of intelligent automation and human-computer interaction.

The first GPT was introduced in 2018 by OpenAI. OpenAI has released very influential GPT foundation models that have been sequentially numbered, to comprise its "GPT-*n*" series. Each of these was significantly more capable than the previous, due to increased size (number of trainable parameters) and training. The most recent of these, GPT-4 was released in March 2023. Such models have been the basis for their more task-specific GPT systems, including models fine-tuned for instruction following—which in turn power the ChatGPT chatbot service.

The term "GPT" is also used in the names and descriptions of such models developed by others. For example, other GPT foundation models include a series of models created by EleutherAI and seven models created by Cerebras in 2023. Also, companies in different industries have developed task-specific GPTs in their respective fields, such as Salesforce's "EinsteinGPT" (for CRM) and Bloomberg's "BloombergGPT" (for finance).

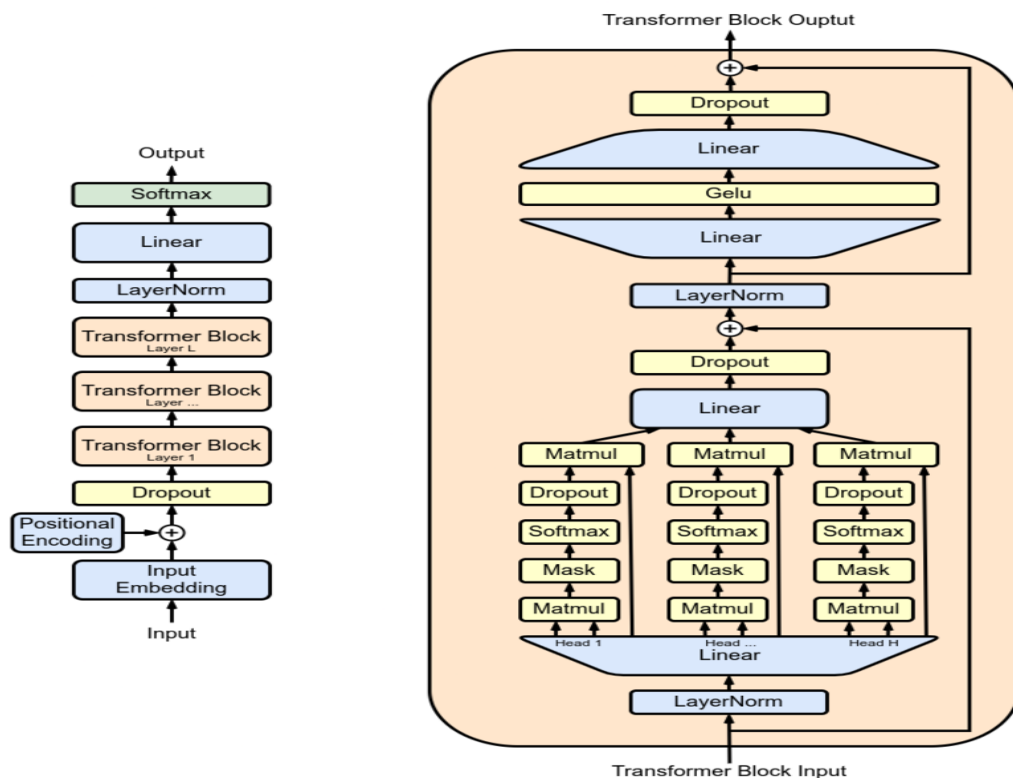


Figure 1.2. Transformer Model

Generative pre-training (GP) was a long-established concept in machine learning applications. It was originally used as a form of semi-supervised learning, as the model is trained first on an unlabeled dataset by learning to *generate* data points in the dataset, and then it is trained to classify a labelled dataset.

Modern transformer architecture was not available until 2017 when it was published by researchers at Google in a paper "Attention Is All You Need". That development led to the emergence of large language models such as BERT in 2018 which was a pre-trained transformer (PT) but not designed to be generative (BERT was an "encoder-only" model). Also around that time, in 2018, OpenAI published its article

entitled "Improving Language Understanding by Generative Pre-Training," in which it introduced the first generative pre-trained transformer (GPT) system ("GPT-1").

Prior to transformer-based architectures, the best-performing neural NLP (natural language processing) models commonly employed supervised learning from large amounts of manually-labeled data. The reliance on supervised learning limited their use on datasets that were not well-annotated, and also made it prohibitively expensive and time-consuming to train extremely large language models.

The semi-supervised approach OpenAI employed to make a large-scale generative system—and was first to do with a transformer model—involved two stages: an unsupervised generative "pretraining" stage to set initial parameters using a language modeling objective, and a supervised discriminative "fine-tuning" stage to adapt these parameters to a target task. Regarding more recent GPT foundation models, OpenAI published its first versions of GPT-3 in July 2020. There were three models, with 1B, 6.7B, 175B parameters, respectively named babbage, curie, and davinci (giving initials B, C, and D).

In July 2021, OpenAI published Codex, a task-specific GPT model targeted for programming applications. This was developed by fine-tuning a 12B parameter version of GPT-3 (different from previous GPT-3 models) using code from GitHub.

In March 2022, OpenAI published two versions of GPT-3 that were fine-tuned for instruction-following (instruction-tuned), named davinci-instruct-beta (175B) and text-davinci-001, and then started beta testing code-davinci-002. text-davinci-002 was instruction-tuned from code-davinci-002. Both text-davinci-003 and ChatGPT were released in November 2022, with both building upon text-davinci-002 via reinforcement learning from human feedback (RLHF). text-davinci-003 is trained for following instructions (like its predecessors), whereas ChatGPT is further trained for conversational interaction with a human user.

OpenAI's most recent GPT foundation model, GPT-4, was released on March 14, 2023. It can be accessed directly by users via a premium version of ChatGPT, and is available to developers for incorporation into other products and services via OpenAI's API. Other producers of GPT foundation models include EleutherAI (with a series of models starting in March 2021) and Cerebras (with seven models released in March 2023).

The GPT (Generative Pre-trained Transformer) model is capable of generating algorithms through its autoregressive language modeling mechanism. By conditioning on input prompts or queries, GPT can generate coherent and contextually relevant sequences of text, including algorithmic code snippets. During the generation process, GPT predicts the next token in the sequence based on the context provided by the preceding tokens, leveraging its multi-layered transformer architecture and learned representations of language. This allows GPT to produce algorithmic code that aligns with the syntax and semantics implied by the input prompt. Additionally, GPT can be fine-tuned on algorithmic data or trained in specific domains to improve the quality and relevance of the generated algorithms. Overall, GPT's ability to understand and generate text makes it a versatile tool for automating algorithmic tasks and assisting developers in code generation workflows.

1.2 Language Translation Model

The Helsinki NLP models, developed by the University of Helsinki's NLP research group, are renowned for their high performance in natural language processing tasks, including language translation. One prominent model within this family is "Helsinki-NLP/opus-mt-en-hi," tailored specifically for translating text between English (en) and Hindi (hi). These models typically rely on transformer architectures, which have revolutionized NLP due to their ability to capture long-range dependencies and contextual information effectively. Specifically, transformer models utilize self-attention mechanisms to weigh the importance of different words in a sentence when making predictions, allowing for more accurate translations.

For training, the Helsinki NLP models leverage large-scale parallel corpora containing pairs of English and Hindi sentences, ensuring the model learns the nuances of both languages and the mappings between them. The training process involves iteratively adjusting millions of parameters to minimize translation errors on the training data, often using techniques like gradient descent and backpropagation. What distinguishes the Helsinki NLP models is their emphasis on multilingualism and cross-lingual transfer learning. These models are often trained on data from multiple languages simultaneously, enabling them to learn universal language representations and transfer knowledge across language boundaries. This multilingual training approach enhances the models' ability to generalize and perform well even on low-resource languages.

Furthermore, the Helsinki NLP models benefit from continuous updates and improvements, incorporating the latest advancements in NLP research. They are also open-source, allowing researchers and developers worldwide to access, fine-tune, and deploy them for various applications.

The Helsinki NLP models, including "Helsinki-NLP/opus-mt-en-hi," represent a cutting-edge approach to language translation, leveraging transformer architectures, multilingual training, and open-source collaboration to achieve state-of-the-art performance in translating between English and Hindi and across a wide range of other languages.

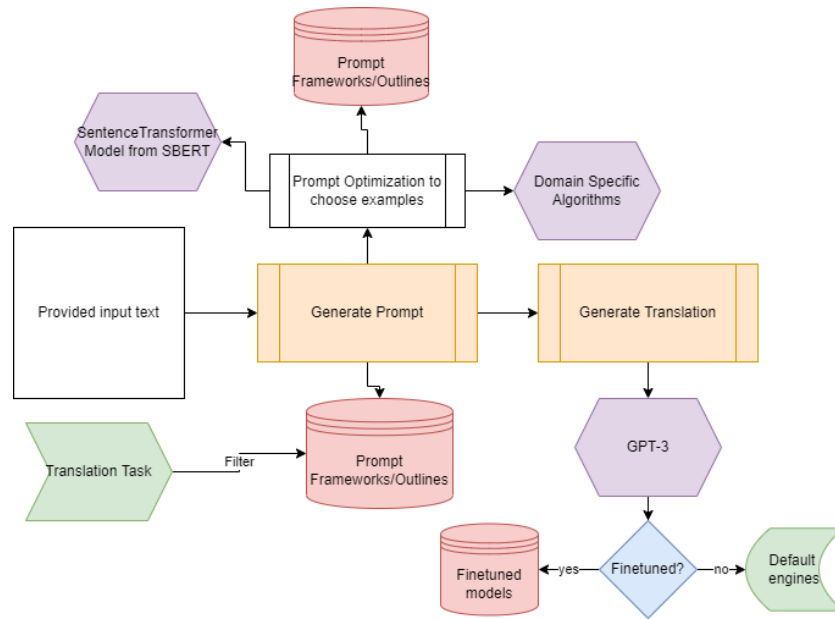


Figure 1.3. Translation Model

1.3 Motivation

The motivation to develop a Source Code Generator arises from the challenges experienced by the participants in the code generation process. And also from the drawbacks of the previous method. The drawbacks are:

XML Bottleneck: The process involves converting code into an XML format before translating it to the final programming language. This unnecessary step acts like a traffic jam, significantly slowing down the entire generation process, especially for complex code.

Looping Limbo: The existing system struggles with loops, which are essential building blocks in programming. Loops can be challenging to represent efficiently in XML, leading to difficulties in generating well-optimized code for them.

The provided solution to overcome the above challenges are:

Direct Code Generation: Unlike the XML-based approach, a trained GPT-3.5 transformer bypasses the entire XML conversion stage. Imagine your assistant understanding your instructions directly, without needing an extra step! This could significantly boost the speed of the entire code generation process.

Learning from Your Code: By training the GPT-3.5 transformer on your specific codebase, it becomes familiar with your coding style and the common patterns you use in your projects. This includes how you typically structure and implement loops. Think of it as your assistant learning your coding habits and preferences. With this newfound knowledge, the model can potentially generate code that incorporates loops accurately and efficiently, overcoming the limitations of the original approach.

1.4 Problem Definition

The challenge faced in the source code generation through intermediate XML step is the slow code generation caused by an intermediate XML step and its inability to handle loops effectively, you're considering a novel approach: training a GPT-3.5 transformer on your own dataset. This powerful AI model has the potential to revolutionize your workflow. Unlike the current method, the GPT-3.5 transformer could bypass the time-consuming XML conversion altogether, directly generating code in the target programming language. This would lead to a significant speed boost, especially for complex projects. But the benefits go beyond speed. By training the model on your specific codebase, it can learn the nuances of your coding style and common patterns, including how you typically structure and implement loops. This newfound knowledge empowers the transformer to generate accurate and efficient code that adheres to your project's conventions, particularly when dealing with loops – a current weak point in the existing process. However, there are hurdles to consider. Training a GPT-3.5 transformer requires substantial computational resources and expertise. Additionally, the success of this approach hinges on the quality and size of your dataset. It needs to contain a good amount of well-structured code examples, including various loop implementations, to effectively teach the model these patterns.

1.5 Problem Illustration

Input Statement	XML Pseudocode Generation	Code Generation
Provide a code to find factorial of a number	<pre> <program> <function name="calculate_factorial"> <parameters> <parameter type="int" name="n"/> </parameters> <body> <if condition="n == 0 or n == 1"> <return>1</return> </if> <else> <return>n * calculate_factorial(n-1)</return> </else> </body> </function> <assignment variable="result"> <call function="calculate_factorial"> <argument>5</argument> </call> </assignment> <print> <value>result</value> </print> </program> </pre>	<pre> def calculate_factorial(n): if n == 0 or n == 1: return 1 else: return n * calculate_factorial(n-1) # User-specified task result = calculate_factorial(5) print(result) </pre>

Figure 1.4. Problem Illustration

The proposal suggests leveraging GPT-3.5, a powerful language model, to address two key pain points in the current code generation process. Firstly, it highlights the potential to eliminate the XML conversion step entirely, streamlining the process by enabling direct code generation in the target programming language. By bypassing this intermediate step, GPT-3.5 can significantly enhance the speed and efficiency of the code generation process, particularly for complex projects where the XML conversion adds unnecessary overhead.

Secondly, the proposal emphasizes the capability of GPT-3.5 to learn and adapt to the specific coding style and patterns used in the given codebase. By training the model on the existing codebase, it can familiarize itself with the unique structures, conventions, and implementation practices, including how loops are typically structured and utilized. This tailored training enables GPT-3.5 to generate code snippets that align more closely with the coding style of the project, thereby enhancing the accuracy and effectiveness of the generated code, especially in handling loops and other common programming constructs. Overall, by harnessing the capabilities of GPT-3.5, the proposed approach offers a promising solution to improve both the speed and quality of code generation processes, addressing the existing pain points and enhancing the overall efficiency of software development workflows.

1.6 Objective of The Project

The project aims to harness the capabilities of advanced machine learning models, specifically GPT-3.5 for source code generation and a sequence-to-sequence model for language translation, to address key challenges in software development and language processing tasks. In the realm of source code generation, the objective is to fine-tune the GPT-3.5 model to understand user queries effectively and generate Python code snippets that accurately fulfill the specified requirements. This involves training the model on conversational data structured similarly to interactions in the Chat Completions API, enabling it to comprehend user intents and produce code that efficiently addresses their needs. By automating the generation of code snippets, the project seeks to streamline software development workflows, reducing manual effort and accelerating the pace of development. In the domain of language translation, the project aims to train a sequence-to-sequence model to accurately translate English text into Hindi. Through preprocessing and training on parallel English-Hindi datasets, the model learns to understand the nuances of both languages and produce translations that preserve meaning and context. This facilitates effective communication and localization in multilingual environments, empowering users to seamlessly interact with content in their preferred language.

2. LITERATURE REVIEW

Heidorn [1] proposed a natural language converging system that takes a description of a program to generate its corresponding source code. The program belongs to a certain domain where programs answer specific questions. The system first checks the completeness of the description of the program. If the first stated program description is sufficient, the program will produce the corresponding problem solving program with the answer. If the program description is insufficient or ambiguous, the system will ask questions to resolve the ambiguity. The program takes in both description of the program and the input values which mostly consist of number literals to be computed. The system continuously asks users questions until a sufficient amount of ambiguity is resolved to answer the question from the program description. The generated source code is in the form of GPSS (Gordon's Programmable Simulation system or General Purpose Simulation system). It is used to show process flow oriented simulation such as stimulating workflows in factories.

Mandal and Naskar [2] proposed the system that generates a program solving mathematical word problems from natural language descriptions. The approach extracts relevant information from the natural language description and store them into an object oriented template. The mapped template is used to perform mathematical operations to solve the problems. For information extraction, they used natural language semantic role labelling and made a template called the OIA triplet (owner item attribute) and stored the template with the extracted information.

Knoll [3] proposed a programming language that read structured line by line natural language (in English, German, and Arabic) and produces the corresponding program in Java. The basic features of Pegasus consists of reading natural language, generating source code, and expressing natural language. When reading natural language it extracts keywords that have logical meaning such as if or then. With these keywords, it can extract the location of each statement and command clause. These pieces of information are then stored in a storage which, the author's defined as the brain in the form of an idea notation. The idea notation is a data structure defined to keep the syntax in the semantics of the natural language description. For the generation of source code Pegasus uses the meaning library which is the database defined to match an idea notation and the corresponding Java comments can also Express the program in natural language.

Begel and Graham [4] proposed a voice programming interface that receives spelling word (speech), natural language, or paraphrasing text, to produce the corresponding Java program. This approach has especially dealt with ambiguity in the context of speech recognition such as homophones. However, ambiguities still resides in the point of written natural language.

Curran [5] proposed a natural language interface for programming. With unrestricted Syntax, they used wide coverage syntactic and semantic methods to extract information from the natural language description. It uses a combinatorial grammar parser to get the syntax of natural language description. Although the input may take in unrestricted forms of description, The translation is done in a line by line fashion for evaluation the leader study of how people gave programming instructions in natural language. They found that most people preferred using programming terms than those of simple natural language.

Desai [6] proposed a program synthesizing framework that takes in a natural language description of a program and a training data set to generate the corresponding source code. The training data sets consists of text code pairs of a domain specific language in the corresponding description. With the input dataset, the system learns the language and generates a code snippet that matches the description of the program. The system Learns any language from the input data sets enabling the system to be language agnostic. If the user can provide any Programmable code and text pairs the system can generate the corresponding program of code.

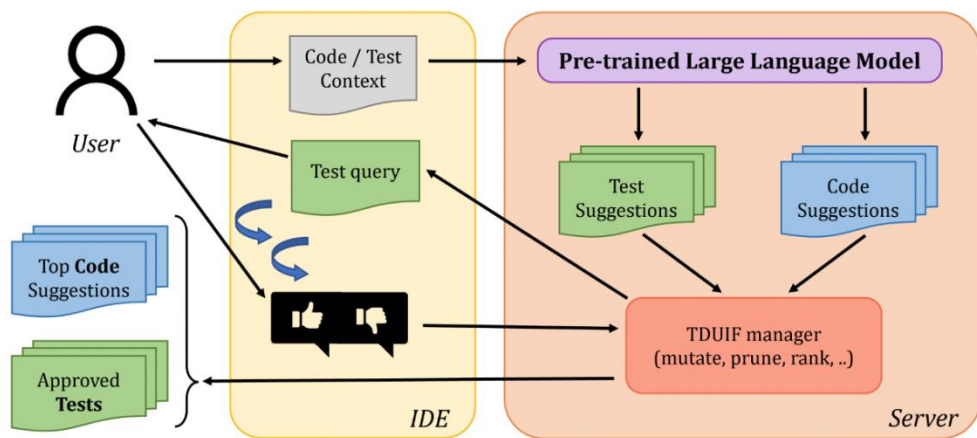


Figure 1: Workflow for test-driven user-intent formalization (TDUIF).

Figure 2.1 Pre-trained Large Language Model

Yin and Neubig [7] proposed a method to parse natural language descriptions to generate python code snippets. They modeled neural architecture that uses a probabilistic grammar model to explicitly capture the syntax of the programming languages prior knowledge. they also found that this approach is effective in scalability when generating Complex programs. they stated that this approach out performed many code generations approaches that use semantic parsing.

Zhong [8] proposed a model that generates a structured query from a natural language using reinforcement learning. The natural language description, in the form of questions is translated to corresponding SQL queries. The model uses the mixed objective of reward weights and cross entropy loss to train executions of the database and land policies to generate conditions of SQL. This approach leverages the structure of SQL to limit the range of generated queries to simplify the generation problem.

Gu [9] proposed a deep neural network and a code search tool that takes in a natural language description of a snippet for retrieving the corresponding code snippet. The deep neural network CODE (code description embedding neural network) is trained to capture the semantic similarities of natural language description and the code snippet. The two different modalities are trained and embedded into unified vectors. When a code snippet and description are semantically similar, the embedded vectors will be close to each other. With this model they implemented a deep learning based code search tool DeepCSand evaluated their approach.

Somasundaram and Swaminathan [10] proposed a compiler that parses the natural language description of a program to generate intermediate representation to help the compiler to convert them into the target language with minimal effort. They aim to reduce ambiguity by parsing through the natural language descriptions of a problem statement and generate the corresponding object oriented program. The key component of their approach consists of a syntactic analyzer, symbol table, lexical analyzer, semantic analyzer, intermediate code generator and code generator.

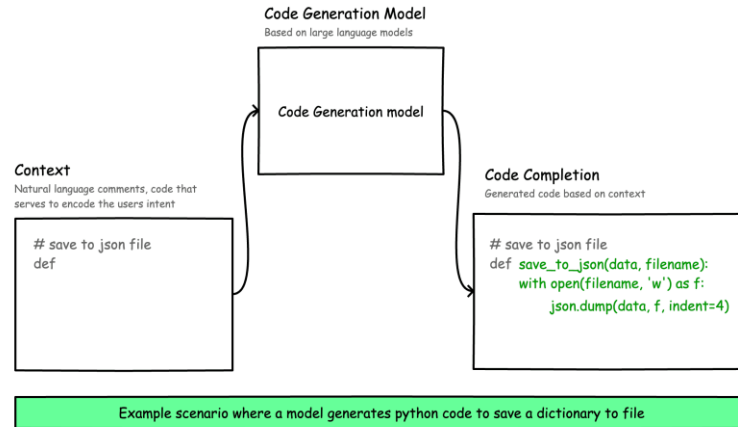


Figure 2.2 Code Generation Model

Manshadi [11] proposed a system that generates source code from a natural language description. The system receives unconstrained instructions and one or more input and output examples. This technique is used to decompose a problem into simpler problems that can be individually solved to reduce the complexity of problem-solving. They also use the method of probabilistic programming by example (PPbE) to reduce the number of possible solutions.

Soeken [12] proposed an assisted flow for behavior driven development (BDD) where the user provides an acceptance test composed of natural language dialogue with the computer about code pieces. The system extracts code information using NLP techniques from the dialogue to produce skeleton code.

Ahmed Sadik, Antonello Ceravola, Frank Joublin, Jibesh Patra [13] demonstrated the potential of large language models (LLMs) such as ChatGPT to help improve software performance. Author LL.M. It can make people useful, but it cannot replace their skills. This article explores the field of LL.M. It can perform different tasks such as code generation, data generation, and error detection. They demonstrate the potential of the LL.M. Update software development by addressing advances in areas that show deep learning's past limitations have been overcome.

Man Fai Wong, Shangxin Guo, Ching Nam Hang, Siu Wai Ho, Chee Wei Tan [14] investigated how large language models (LLMs) learned in large numbers (large numbers) can be replaced by intelligent assisted programming. Using the idea that numbers are similar to natural languages, LLMs can perform tasks such as generating numbers, executing them, and looking for errors. This article explores how this differs

from previous LLM studies and includes a review of the extensive literature, evaluations, and recent developments in AI-assisted programming for Transformer-based LLMs. He also talked about future challenges and opportunities.

Jiho Shin, Jaechang Nam [15] is working on a new programming method: using non-traditional programming languages. This method must overcome limitations such as high tuition fees and limited teaching capacity. This article reviews existing research on automatic code generation from annotations. It divides this process into input and output, analyzes current trends and reports future trends. The authors recommend focusing on language modeling rules and searching for better representatives of programming techniques such as embedding techniques and pre-learning models (implemented in NLP).

Tapornoy Adhikari [16] studied how natural language processing (NLP) can be used to identify program code based on description. It highlights benefits such as less effort and increased productivity, but also acknowledges challenges such as ambiguous language and complex numbers. This article reviews current methods, evaluations, and future directions, including exploring advanced machine learning techniques and combining NLP with other methods. Overall, research shows that NLP has the potential to revolutionize software development.

Hendrig Sellik [17] research papers on how to use natural language processing (NLP) to generate code programs from descriptions. NLP techniques such as word embedding and deep learning models are used for tasks such as generating code comments, executing code written by developers, and even generating names for jobs and classes. Although many generations still face problems, NLP holds promise in developing software for better results.

Sethunya R Joseph, Hlomani Hlomani, Keletso Letsholo, Freeson Kaniwa, Kutlwano Sedimo [18] Research papers on natural language processing (NLP), a branch of computer science dedicated to the computer's understanding and manipulation of human speech. NLP draws on disciplines as diverse as linguistics and computer science to perform tasks such as machine translation and transcription. This article explores the history of the science of NLP, its increasing importance due to the rise of the Internet and the need for software that can process text in many languages.

3. SOURCE CODE GENERATION THROUGH GPT INTEGRATION

The proposed methodology showcases a noteworthy improvement in ROUGE score, underscoring its efficacy in generating source code that precisely aligns with user queries and fulfils their requirements with exceptional accuracy. This enhancement is pivotal in addressing the perennial challenge of bridging the gap between user intent and generated code, ensuring that the resulting solutions accurately capture the essence of the desired functionality. By leveraging advanced natural language processing techniques, the methodology effectively circumvents the language barrier that often hinders effective communication between users and software systems. Moreover, it enhances the understandability of the generated source code, making it more accessible and interpretable to users regardless of their level of programming expertise. This breakthrough not only streamlines the development process but also fosters collaboration and innovation by empowering a broader spectrum of stakeholders to participate in software engineering tasks. The demonstrated improvements underscore the transformative potential of the proposed methodology in revolutionizing the way source code is generated, understood, and utilized in real-world applications.

The GPT (Generative Pre-trained Transformer) model stands as a pinnacle of natural language processing (NLP) and artificial intelligence (AI) research, representing a transformative advancement in machine learning. Built upon the Transformer architecture, GPT leverages self-attention mechanisms to capture intricate semantic relationships within text data, enabling it to generate coherent and contextually relevant outputs. Through pre-training on vast corpora of text data, GPT learns to understand and generate human-like language, exhibiting remarkable fluency and versatility across a wide range of tasks, from language translation and summarization to question answering and code generation. Its ability to grasp the nuances of language and generate coherent responses has revolutionized numerous applications, empowering developers, researchers, and businesses to harness the power of AI in unprecedented ways.

As a cornerstone of modern NLP, GPT continues to push the boundaries of what is possible in natural language understanding and generation, heralding a new era intelligent automation and human-computer interaction.

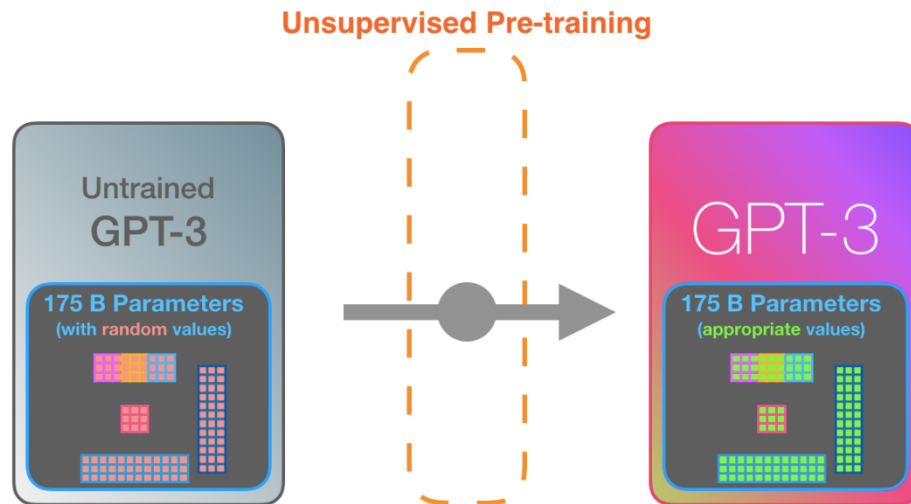


Figure 3.1 Training GPT-3 transformer model

3.1 Fine-tuning the model

Fine-tuning represents a critical technique in optimizing the performance of models accessible through APIs, offering a host of benefits that significantly enhance their capabilities. By fine-tuning models, developers can achieve superior results compared to simple prompting methods. This is because fine-tuning allows for training on a much larger volume of examples, going beyond the constraints imposed by prompts. With access to a broader range of training data, the model gains a deeper understanding of context, semantics, and language nuances, leading to more accurate and nuanced responses. Moreover, fine-tuning enables the creation of concise prompts, thereby achieving token savings and reducing computational overhead. This efficiency not only enhances the model's performance but also facilitates faster inference, lowering latency and improving overall responsiveness. Ultimately, fine-tuning empowers developers to unlock the full potential of AI models, delivering more robust, context-aware, and efficient solutions to meet a diverse range of application requirements. The fine-tuning process encompasses the following:

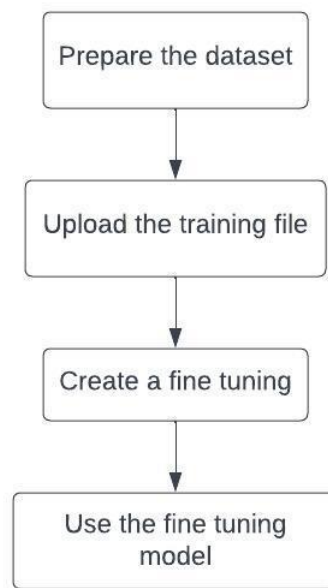


Figure 3.2 Stages of Fine-tuning a model

3.1.1 Prepare and upload training data:

Organizing the dataset into a conversational format, akin to the structure used by the Chat Completions API, involves structuring the data as a series of messages, each with a designated role and corresponding message content. This format facilitates the training of models for tasks such as dialogue generation or chatbot development. The dataset is typically converted into a JSON Lines file format, where each line represents a single JSON object containing the role and message content. This structured approach ensures that the model can effectively learn the dynamics of conversations and generate coherent responses. Once the training data is prepared in this format, the next step is to upload it using the Files API, which enables seamless integration of the data into fine-tuning jobs. With the dataset uploaded, the model can undergo the fine-tuning process, during which it learns to generate responses based on the conversational context provided in the training data. This iterative process culminates in a model that is proficient in generating natural and contextually relevant responses in a conversational setting, laying the groundwork for various applications in natural language understanding and generation. To achieve this, the dataset needs to be converted into a JSON Lines file format as shown in figure 3.1.1.

```
[{"messages": [{"role": "system", "content": " You are an AI assistant that generates the python codes based on the user input"}, {"role": "user", "content": "Write a NumPy program to repeat elements of an array. "}, {"role": "assistant", "content": "import numpy as np\nrx = np.repeat(3, 4)\nprint(rx)\nrx = np.array([[1,2],[3,4]])\nprint(np.repeat(x, 2))\n"}], [{"role": "system", "content": " You are an AI assistant that generates the python codes based on the user input"}, {"role": "user", "content": "Write a Python function to create and print a list where the values are square of numbers between 1 and 30 (both included). "}, {"role": "assistant", "content": "def printValues():\n\tl = list()\n\tfor i in range(1,31):\n\t\tl.append(i**2)\n\tprint(l)\n\t\t\nprintValues()\n"}], [{"role": "system", "content": " You are an AI assistant that generates the python codes based on the user input"}, {"role": "user", "content": "Write a Python program to remove duplicates from a list of lists. "}, {"role": "assistant", "content": "import itertools\nnum = [[10, 20], [40], [30, 56, 25], [10, 20], [33], [40]]\nprint("Original List", num)\nnum.sort()\nnew_num = list(num for num, _ in itertools.groupby(num))\nprint("New List", new_num)\n"}], [{"role": "system", "content": " You are an AI assistant that generates the python codes based on the user input"}, {"role": "user", "content": "Write a NumPy program to compute the x and y coordinates for points on a sine curve and plot the points using matplotlib. "}, {"role": "assistant", "content": "import numpy as np\nimport matplotlib.pyplot as plt\n# Compute the x and y coordinates for points on a sine curve\nrx = np.arange(0, 3 * np.pi, 0.2)\nry = np.sin(rx)\nprint("Plot the points using matplotlib")\nplt.plot(x, y)\nplt.show()\n"}], [{"role": "system", "content": " You are an AI assistant that generates the python codes based on the user input"}, {"role": "user", "content": "Write a Python program to alter a given SQLite table. "}, {"role": "assistant", "content": "import sqlite3\nfrom sqlite3 import Error\n\ndef"}]
```

Figure 3.3 Training Data

After preparing the training data, the next step is to upload it using the Files API. This allows the data to be utilized in fine-tuning jobs effectively. Once everything is done we can start the finetuning of the model.

3.1.2 Create a Fine Tune Model:

After uploading the file, the subsequent action is to initiate a fine-tuning job. This is accomplished by creating a fine-tuning job through the OpenAI SDK:

Step-1: Create an instance of the OpenAI client for interactions.

Step-2: Utilize the OpenAI client to create a fine-tuning job.

Step-3: Specify the training file ID that was returned when the training file was uploaded to the OpenAI API as its identifier (e.g., "file-abc123")

Step-4: Indicate the model to be used for fine-tuning as "gpt-3.5-turbo."

Upon commencing a fine-tuning job, the time for completion may fluctuate based on queue dynamics and the duration of the training process, which is contingent on factors like the model and dataset size. After the training concludes, users will be notified with a confirmation mail.

Once the job has been successfully completed, you'll observe that the fine-tuned model field is filled with the model's name when retrieving the job details.

The overall algorithm for fine tuning:

Step 1: Import pandas

Step 2: Upload the csv dataset

Step 3: Encode the dataset with 'cp1252'

Step 4: Create a new jsonl file named 'training.jsonl' and open it in writing mode

Step 5: Create a function `CREATE_DATASET` which takes a judgment and its corresponding summary as the input parameters.

Step 6: The function creates a message as shown in the above example and returns it.

Step 7: Iterate through the CSV file and send the judgment and summary one by one to `CREATE_DATASET`.

Step 8: While iterating write the returned message (every time in a new line) from `CREATE_DATASET` in training.jsonl.

Step 9: From 'openai' library import 'OpenAI' class

Step 10: Create an instance of the OpenAI class as 'client' with an 'api_key' parameter set as user API KEY , which will be used to communicate with the OpenAI API.

Step 11: Upload the training file using 'file.create()' method under 'OpenAI' and set its 'file' parameter as open 'training.jsonl' file in read binary mode and set the 'purpose' parameter as 'fine-tune'. A file id will be generated

Step 12: Fine tune the model using 'fine_tuning.jobs.create()' method under OpenAI class and set its parameters. Set 'training_file' as the file id generated after uploading the training file, set the 'model' as the 'gpt-3.5-turbo'. This will create a fine-tuning job id.

Step 13: Retrieve the fine-tuned model status using `fine_tuning.job.retrieve()` method under OpenAI. The model name will be generated.

The fine-tuning process for customizing AI models begins with the preparation and encoding of a dataset, followed by the creation of a JSONL file to store formatted data. Utilizing a function to process individual data points, messages are generated and written to the JSONL file. With the OpenAI client initialized, the dataset is uploaded, and a fine-tuning job is initiated by specifying the training file and model parameters. Once the fine-tuning process is underway, the status of the model is monitored until completion, enabling users to harness the customized model's enhanced capabilities for specific tasks, ultimately facilitating more accurate and contextually relevant natural language processing applications.

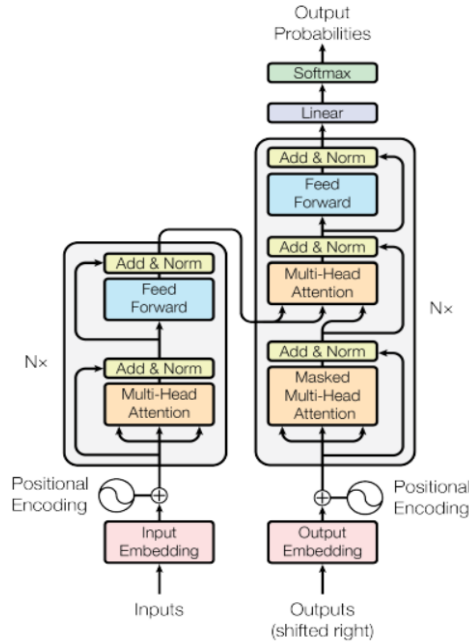


Figure 3.4 GPT Transformer Model

3.2 Generating source codes using fine-tuned model:

After completing the fine-tuning process, our refined model is primed for generating Python source code. Leveraging a fine-tuned GPT-3 model, our system is equipped to streamline Python code generation tasks with remarkable precision and efficiency. This entails enhancing the GPT-3 language model's comprehension of Python syntax and structure through specialized training. By supplying the model with well-defined prompts and concise descriptions of desired functionalities, we can harness its capabilities to produce Python code snippets tailored to automate tasks or implement specific algorithms. This bespoke development tool serves as a catalyst for accelerating our workflow, enabling our team to expedite the creation of Python scripts and applications while conserving valuable resources. Moreover, the integration of this tailored model into our development pipeline empowers us to tackle complex coding challenges with greater agility and confidence, thereby fostering innovation and driving continuous improvement within our organization.

The usage of the finetuned model goes something like this:

```
question = st.text_input("Type your question here...")
response = client.chat.completions.create(
    model= <Finetuned model name>,
```

```

        messages = [{"role": "user", "content": question}],
    )
    Client = OpenAI(api_key=YOUR_OPENAI_API_KEY)
    generated_code = response.choices[0].message.content

```

Here ‘question’ is the variable which holds the users query Q_a that is the question and when the user gives the input to the system the response function comes to the action where it takes the Q_a as the input for itself and generates a code G_c for it once the response is generated it is passed on to the function ‘*generate_algorithm()*’ where the algorithm for the G_c is stored in *Generated_algorithm* G_a .

3.3 Generating Algorithm for the user’s query

Generative Pre-trained Transformers (GPT) generate algorithms through a process called autoregressive language modelling. Here's a detailed breakdown of how GPT generates algorithms:

Tokenization: GPT first tokenizes the input query or prompt into a sequence of tokens. These tokens represent individual words, punctuation marks, and other linguistic elements.

Embedding: Each token is then converted into a high-dimensional vector representation known as embeddings. These embeddings capture semantic and syntactic information about the token's meaning and context in the input sequence.

Autoregressive Generation: GPT employs an autoregressive generation process, where it predicts the next token in the sequence given the tokens generated so far. It iteratively generates tokens one at a time, conditioning on the previously generated tokens. This process is repeated until a predefined stopping criterion is met, such as reaching a maximum sequence length or generating a special end-of-sequence token.

Probability Distribution: At each step of the generation process, GPT calculates the probability distribution over the vocabulary of possible tokens for the next position. This distribution is generated by passing the embeddings of the current tokens through multiple layers of transformer blocks, which apply self-attention mechanisms to capture dependencies between tokens.

Sampling: GPT samples the next token from the probability distribution generated in the previous step. Sampling techniques may vary, but common methods include greedy

decoding, where the token with the highest probability is chosen at each step, or stochastic sampling, where tokens are probabilistically sampled based on their likelihood.

Repetition and Diversity: GPT incorporates mechanisms to prevent repetition and encourage diversity in the generated text. Techniques such as nucleus sampling or temperature scaling adjust the probability distribution to control the diversity of generated tokens and mitigate the tendency to repeat phrases.

Post-Processing: Once the desired sequence length is reached or a stopping criterion is met, GPT post-processes the generated tokens to form a coherent algorithm. This may involve tokenizing, syntax parsing, and formatting the tokens into a structured code snippet.

3.4 Creating a language translation model

The Helsinki model, an exemplar in the domain of language translation, stands as a testament to the continuous advancements in natural language processing. Developed with state-of-the-art techniques and methodologies, the Helsinki model exhibits unparalleled proficiency in accurately translating text across multiple languages. Its robust architecture, built upon cutting-edge transformer-based frameworks, enables it to comprehend and preserve the nuances of language, facilitating seamless communication across linguistic boundaries. By leveraging large-scale pre-training on diverse corpora and innovative fine-tuning strategies, the Helsinki model achieves remarkable fluency and fidelity in translation, delivering outputs that are not only linguistically accurate but also contextually coherent. Its versatility and adaptability empower users across diverse domains and industries to transcend linguistic barriers, facilitating global collaboration, cultural exchange, and information dissemination on an unprecedented scale. As a cornerstone in the realm of language translation, the Helsinki model epitomizes the transformative potential of artificial intelligence in fostering cross-cultural understanding and connectivity in an increasingly interconnected world.

3.4.1 Dataset Acquisition and Preprocessing:

Dataset: We utilize the "cfilt/iitb-english-hindi" dataset from the Hugging Face Datasets library. This dataset provides parallel English and Hindi sentences for training the translation model.

Preprocessing: A preprocessing function is defined to handle the following:

- a. Extract English and Hindi sentences from the dataset.
- b. Tokenize the sentences using a pre-trained tokenizer from the Helsinki-NLP/opus-mt-en-hi model checkpoint. This tokenizer converts text into numerical representations suitable for the machine translation model.
- c. Set a maximum length for both English and Hindi sentences to ensure efficient model training.

3.4.2 Model Architecture and Training:

Pre-trained Model: The heart of our translation model lies the `TFAutoModelForSeq2SeqLM` class from Transformers, a powerful framework renowned for its prowess in natural language processing tasks. This class serves as the foundation for a pre-trained sequence-to-sequence model architecture tailored specifically for machine translation endeavors. The selected pre-trained model checkpoint, "Helsinki-NLP/opus-mt-en-hi," stands as a pinnacle of linguistic understanding, having been meticulously trained on an extensive corpus of English-Hindi translations. Leveraging the vast amount of data encapsulated within this model checkpoint, our translation model possesses a deep understanding of the intricacies and nuances inherent in both languages. This comprehensive training equips the model with the capability to accurately translate text between English and Hindi, facilitating seamless communication and comprehension across linguistic boundaries. By harnessing the power of this pre-trained model, we can confidently provide users with high-quality translation services, enabling them to bridge language barriers and engage with content in their preferred language with ease and accuracy.

Training Configuration:

Each of the specified hyperparameters plays a critical role in shaping the training process and ultimately determining the performance of our translation model:

a. Batch Size: Setting the batch size to 16 dictates the number of sentences processed simultaneously during each iteration of training. By batching data, computational efficiency is improved, as parallel processing can be leveraged effectively. Additionally, smaller batch sizes may lead to more stochastic updates to model parameters, potentially enhancing generalization and preventing overfitting.

b. Learning Rate: The learning rate, set to $2e-5$ (0.00002), governs the magnitude of adjustments made to model parameters during each optimization step. A lower learning rate can help ensure smoother convergence and prevent overshooting minima, while a higher rate can accelerate convergence but risks instability or overshooting. Titrating the learning rate is crucial to strike a balance between rapid convergence and stable optimization.

c. Weight Decay: Introducing a weight decay of 0.01 aids in regularizing the model by penalizing large weights in the parameter space. This regularization technique mitigates the risk of overfitting by discouraging the model from overly relying on specific features or correlations present in the training data. By imposing a penalty on large weights, the model is encouraged to learn simpler, more generalizable patterns.

d. Epochs: Training the model for 15 epochs entails iterating through the entire training dataset 15 times. Each epoch provides the model with exposure to a diverse range of training instances, allowing it to refine its parameters and gradually improve performance. Training for multiple epochs enables the model to capture complex patterns and nuances present in the data, thereby enhancing its ability to generalize to unseen examples.

Training Loop: The `model.fit` function serves as a pivotal component in training our translation model on the meticulously prepared training dataset. Through a series of iterations, the model processes batches of data, each consisting of English sentences paired with their corresponding Hindi translations. During training, the model learns to map input English sentences to their corresponding output Hindi translations, continually refining its internal parameters to optimize translation accuracy. This iterative process involves backpropagation, wherein the model computes gradients and updates its internal parameters based on the error between predicted and actual translations. By iteratively adjusting these parameters, the model gradually improves its ability to accurately translate English sentences into Hindi, honing its linguistic understanding and proficiency. Through this training mechanism, our model evolves into a robust and adept translator, capable of delivering accurate and fluent translations across a diverse range of text inputs.

3.4.3 Inference and Model Saving:

a. Saving the Model: Upon completion of training, it is imperative to save the fine-tuned model to preserve its learned parameters and architecture for future use. This enables seamless integration into translation tasks without the need for re-training, saving valuable time and computational resources. By persisting the trained model's state, we ensure consistency and reproducibility across different translation instances. Additionally, saved models can be shared with collaborators or deployed into production environments for real-world applications, enhancing accessibility and scalability.

b. Translation Function: The `model.generate` function serves as the workhorse for translating new English sentences into Hindi. This process begins by tokenizing the input English text, breaking it down into its constituent tokens or sub-words, which are then fed into the trained model. Leveraging its learned parameters and architecture, the model generates the corresponding Hindi translation by decoding the input sequence and predicting the most probable output sequence. This translation function facilitates on-the-fly translation of arbitrary English text, providing a seamless and efficient means of converting English sentences into their Hindi counterparts.

c. Tokenizer Configuration: Throughout the translation process, the tokenizer plays a pivotal role in converting text between its human-readable form and the numerical representation understood by the model. During translation, the tokenizer is employed once again to decode the model's output, transforming the numerical representation of the generated Hindi text back into its original human-readable form. By adhering to the same tokenization scheme used during training, consistency is maintained between input and output representations, ensuring accurate and faithful translation results. This tokenizer configuration ensures the fidelity and intelligibility of the translated Hindi text, enabling users to seamlessly comprehend and interact with the output of the translation model.

The language translation process encompasses several essential steps to develop and deploy effective translation models. Beginning with data preparation, parallel text data in the source and target languages are collected and pre-processed. Model selection involves choosing a suitable pre-trained architecture, followed by fine-tuning on the prepared dataset to optimize translation accuracy. Hyperparameter tuning and

validation ensure the model's performance and generalizability, while saving the trained model enables reuse for future translation tasks. During translation, the model processes input text to generate translated output, which may undergo post-processing for enhanced fluency and correctness. By following these steps, practitioners can develop robust translation models facilitating effective communication across language barriers.

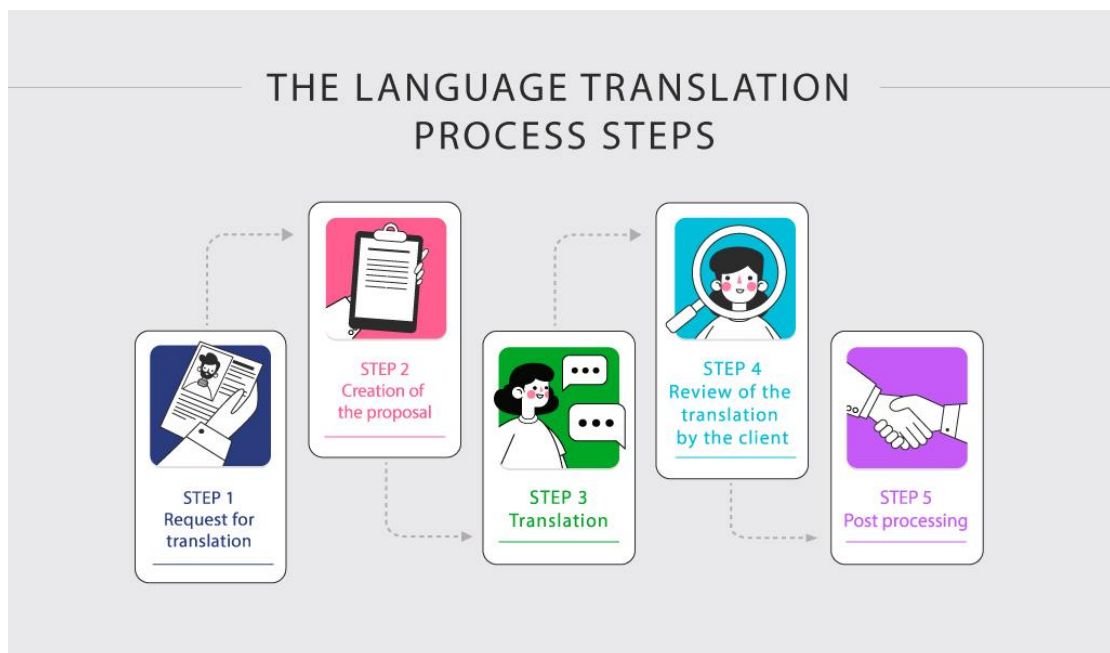


Figure 3.5 Language Translation Model

The figure [3.3] showcases a simplified view of a Transformer model in action for machine translation. Imagine an English sentence entering on the left. The encoder, like a detective, dissects the sentence, analysing each word's meaning and how it connects to others. This analysis is captured by those blue boxes with swirling arrows. Meanwhile, on the right, the decoder acts like a writer crafting the Hindi translation. It starts with a special token (like a "start of sentence" marker) and, using its own internal attention (represented by orange boxes with curvy arrows), builds the translation. But here's the clever part: the decoder also peeks back at the encoder's analysis (through the double-headed arrows) to ensure each translated word aligns with the original meaning in English. This interplay between analysing the source language and attentively generating the target language is what makes Transformers powerful for machine translation tasks.

4. IMPLEMENTATION

File training_dataset.json consists of the dataset which is used to fine-tune the GPT-3.5 transformer model. In Hugging face, we have trained our own language translation model that is created by us, which is used to translate English to Hindi language. Program file is code_gen.py which consists of the frontend interface source code written in streamlit and python acting as the bridge of communication between the user's query and the data retrieval using the semantic search.

Input: User query

Output: Source code generation to the user's query

4.1 FUNCTIONALITY

4.1.1 Preprocessing the data and Fine- tuning the model

The process starts with converting the .csv file dataset to .json file because we are taking GPT-3.5 transformer from the OpenAI, now we have finetuned the model with dataset. Fine-tuning in GPT-3.5 Transformer involves adapting the pre-trained language model to a specific task or domain by further training it on a specialized dataset. Here are the steps involved in the fine-tuning process:

Data Preparation: Gather and preprocess a dataset relevant to the target task or domain. This dataset should be formatted in a way that the model can understand, typically consisting of input-output pairs or prompts and corresponding responses.

Tokenizer Initialization: Initialize the tokenizer associated with the GPT-3.5 model. This tokenizer is responsible for converting text inputs into tokenized sequences that the model can process.

Model Initialization: Instantiate the pre-trained GPT-3.5 Transformer model. This model comes pre-trained on a vast corpus of text data and contains numerous layers and parameters that capture intricate language patterns.

Fine-tuning Setup: Configure the fine-tuning parameters, including the learning rate, batch size, and number of training epochs. These parameters determine how the model updates its internal parameters during fine-tuning.

Fine-tuning Loop:

a. Batching: Divide the fine-tuning dataset into batches of input-output pairs or prompts and responses. This batching process allows for efficient processing of large datasets.

b. Forward Pass: Feed each batch of input sequences through the model, generating predicted outputs or responses. During the forward pass, the model's parameters are fixed, and no updates are made.

c. Loss Calculation: Calculate the loss or error between the model's predicted outputs and the ground truth responses. This loss function quantifies how well the model is performing on the fine-tuning task.

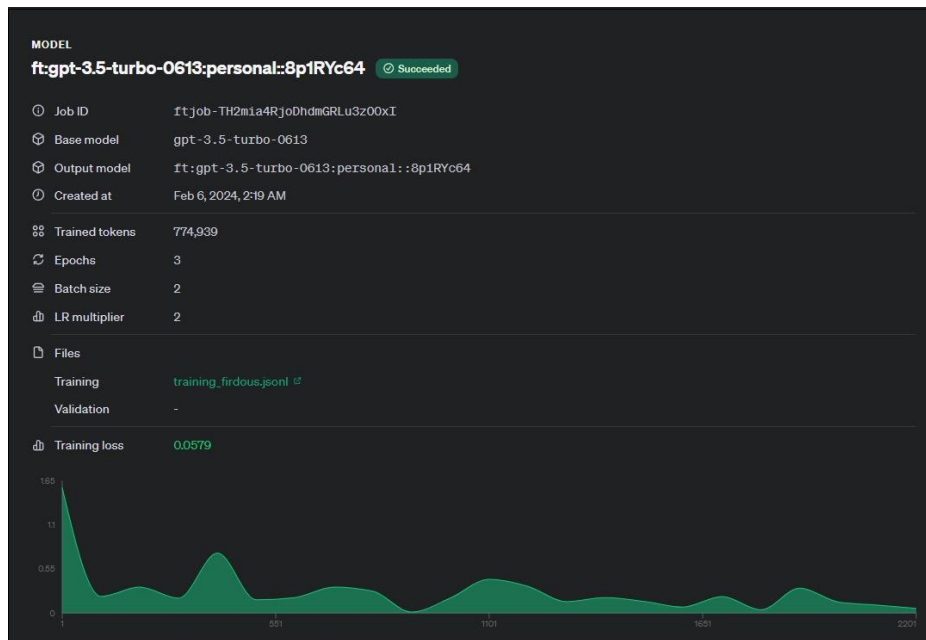
d. Backpropagation: Propagate the loss back through the model and update its parameters using gradient descent optimization. This backpropagation step adjusts the model's parameters to minimize the loss and improve performance on the fine-tuning task.

e. Parameter Update: Update the model's parameters based on the gradients computed during backpropagation, nudging them in the direction that reduces the loss.

f. Repeat: Iterate through multiple batches of data, repeating the forward pass, loss calculation, backpropagation, and parameter update steps until the entire fine-tuning dataset has been processed or a specified number of epochs have been completed.

Evaluation: Assess the fine-tuned model's performance on a separate validation dataset to ensure it generalizes well to unseen data. This evaluation step helps determine whether further fine-tuning or adjustments are necessary.

Model Saving: Once the fine-tuning process is complete and the model performs satisfactorily on the validation dataset, save the fine-tuned model's parameters for future use.



Time	Step	Training loss	Validation loss	Full validation loss
03:37:44	2201	0.0579	-	-
03:34:28	2101	0.0993	-	-
03:31:11	2001	0.1357	-	-
03:27:50	1901	0.3099	-	-
03:24:17	1801	0.0394	-	-
03:21:04	1701	0.2062	-	-
03:17:45	1601	0.0738	-	-
03:14:30	1501	0.1429	-	-
03:11:09	1401	0.1925	-	-
03:07:52	1301	0.1406	-	-
03:04:38	1201	0.3321	-	-
03:01:19	1101	0.4197	-	-
02:58:00	1001	0.1870	-	-
02:54:47	901	0.0115	-	-
02:51:30	801	0.2711	-	-
02:48:13	701	0.3225	-	-
02:45:00	601	0.1926	-	-
02:41:46	501	0.1650	-	-
02:38:31	401	0.7461	-	-
02:35:16	301	0.1839	-	-
02:31:59	201	0.3224	-	-
02:28:44	101	0.2058	-	-
02:25:27	1	1.5639	-	-

Figure 4.1 Fine-tuning the model

4.1.2 Language Translation Model

Data Collection: We need a large dataset of text that is parallel between English and Hindi. This means each sentence in English should have its corresponding translation in Hindi.

Data Preprocessing: The raw data needs cleaning and formatting. This might involve removing noise, correcting errors, and tokenizing the text (breaking it down into smaller units).

Model Selection: We wouldn't typically use a model designed for text classification (like the one in the image) for translation. Instead, we have chose a pre-trained model specifically designed for translation tasks, like T5 from Hugging Face.

Model Fine-tuning: Since we have an English-Hindi dataset, we have fine-tuned the pre-trained model on this specific data. This helps the model learn the nuances of translating between these two languages.

Training: The model is trained using the prepared data. This involves feeding the English sentences and their corresponding Hindi translations into the model and adjusting the model's internal parameters to improve its translation accuracy.

Evaluation: Once trained, we have evaluated the model's performance on a separate dataset to see how well it translates unseen English sentences to Hindi.

Deployment: After the model performs well, we deployed it for real-world use, allowing it to translate English text to Hindi.

Now, our model is ready through pipeline we have used it in our code_gen.py file.

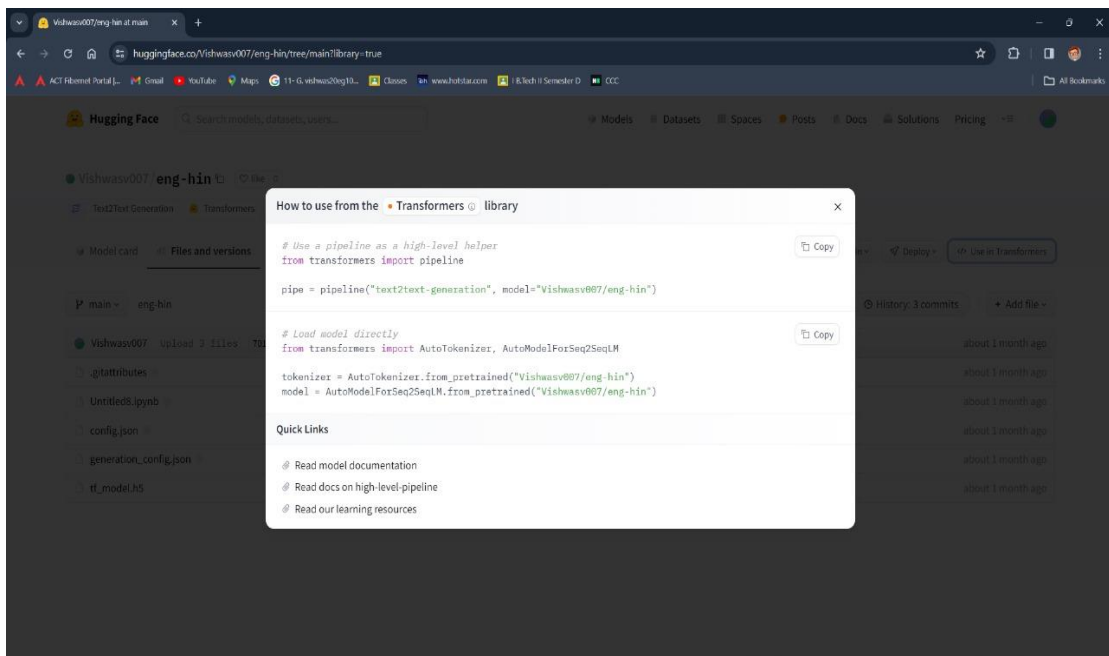


Figure 4.2 Language Translation Model

4.1.3 Code Generation for the user's query

After fine-tuning the GPT-3.5 model for code generation, the process of generating code for a given user's query involves the following steps:

User Query Input: Receive the user's query, which may consist of a natural language description of the desired functionality or a specific coding task.

Tokenization: Tokenize the user query using the same tokenizer that was used during fine-tuning. This converts the text input into a numerical representation that the model can understand.

Model Inference: Feed the tokenized input sequence into the fine-tuned GPT-3.5 model. During inference, the model generates a sequence of tokens representing the output, which includes the code snippet corresponding to the user's query.

Decoding: Decode the generated token sequence into human-readable code. This involves reversing the tokenization process and converting the numerical representation back into text.

Post-processing: Perform any necessary post-processing steps on the generated code to ensure correctness and readability. This may include formatting the code, adding comments, or applying best practices for code style and conventions.

Output Generation: Present the generated code snippet to the user as the result of their query. This could be displayed in an interactive interface, returned as an API response, or saved to a file for further use.

4.2 ATTRIBUTES

- I. **API_KEY:** It is a unique identifier or token that is used to authenticate and authorize access to OpenAI API.
- II. **Transformer:** A transformer is a type of deep learning model architecture used primarily in natural language processing (NLP) tasks. It employs a self-attention mechanism to weigh the significance of different words in a sequence, allowing it to capture long-range dependencies and relationships within the input data effectively. Transformers are characterized by their ability to process input data in parallel, enabling faster training and inference compared to traditional sequential models like recurrent neural networks (RNNs). They often utilize

multi-head attention mechanisms, positional encoding, and feedforward neural networks as components of their architecture. Transformers have become widely popular in NLP due to their ability to achieve state-of-the-art performance in various tasks, including machine translation, text generation, sentiment analysis, and more.

- III. **openai_api_key:** This attribute represents the API key required to authenticate and access the OpenAI API. The API key serves as a form of authentication and authorization to use OpenAI's services. It's typically obtained by registering with OpenAI and generating a unique key associated with the user's account. In this code, `API_KEY` is a placeholder for the actual API key value.
- IV. **model:** This attribute specifies the model to be used for generating responses. It determines the underlying architecture and parameters of the language model. Here, the model being used is 'ft:gpt-3.5-turbo-0613:personal::8p1RYc64', which appears to be a fine-tuned version of the GPT-3.5 model. The specifics of the fine-tuning process and the particular characteristics of this model are indicated by the string.
- V. **AutoTokenizerForSeq2SeqLM:** The "AutoTokenizerForSeq2SeqLM" class refers to an auto-generated tokenizer specialized for sequence-to-sequence language modelling tasks in the Hugging Face library. This class automatically selects the appropriate tokenizer based on the specified language model architecture, streamlining the tokenization process for tasks involving sequences of tokens as both input and output. It provides essential functionality for tokenizing input sequences, converting tokens to numerical representations, handling special tokens, and decoding output sequences from model predictions. Compatible with various sequence-to-sequence language model architectures such as T5, Marian, and BART, this tokenizer facilitates efficient and seamless integration into a wide range of natural language processing tasks, enabling rapid development and deployment of sequence-to-sequence models.

4.3 EXPERIMENTAL SCREENSHOTS

```
[ ] train_dataset = model.prepare_tf_dataset(  
    tokenized_datasets["test"],  
    batch_size=batch_size,  
    shuffle=True,  
    collate_fn=data_collator,  
)
```

```
[ ] validation_dataset = model.prepare_tf_dataset(  
    tokenized_datasets["validation"],  
    batch_size=batch_size,  
    shuffle=False,  
    collate_fn=data_collator,  
)
```

```
▶ generation_dataset = model.prepare_tf_dataset(  
    tokenized_datasets["validation"],  
    batch_size=8,  
    shuffle=False,  
    collate_fn=generation_data_collator,  
)
```

```
[ ] optimizer = AdamWeightDecay(learning_rate=learning_rate, weight_decay_rate=weight_decay)  
model.compile(optimizer=optimizer)
```

```
[ ] model.fit(train_dataset, validation_data=validation_dataset, epochs=15)
```

Epoch 1/15

156/156 [=====] - 85s 337ms/step - loss: 3.7607 - val_loss: 3.9476

Epoch 2/15

156/156 [=====] - 45s 290ms/step - loss: 3.3247 - val_loss: 3.8687

Epoch 3/15

```
[ ] 156/156 [=====] - 52s 336ms/step - loss: 1.7798 - val_loss: 3.8764
Epoch 11/15
156/156 [=====] - 47s 303ms/step - loss: 1.6497 - val_loss: 3.9049
Epoch 12/15
156/156 [=====] - 45s 290ms/step - loss: 1.5377 - val_loss: 3.9189
Epoch 13/15
156/156 [=====] - 47s 300ms/step - loss: 1.4285 - val_loss: 3.9395
Epoch 14/15
156/156 [=====] - 47s 298ms/step - loss: 1.3256 - val_loss: 3.9771
Epoch 15/15
156/156 [=====] - 47s 304ms/step - loss: 1.2320 - val_loss: 4.0013
<keras.src.callbacks.History at 0x7a318f736830>

[ ] model.save_pretrained("tf_model/")

[ ] tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
model = TFAutoModelForSeq2SeqLM.from_pretrained("tf_model/")

/usr/local/lib/python3.10/dist-packages/transformers/models/arian/tokenization_arian.py:197: UserWarning:
  warnings.warn("Recommended: pip install sacremoses.")
All model checkpoint layers were used when initializing TFMarianMTModel.

All the layers of TFMarianMTModel were initialized from the model checkpoint at C:/Users/Vishwas/Desktop/pe
If your task is similar to the task the model of the checkpoint was trained on, you can already use TFMari

▶ input_text = "My name vishwas i am student"

tokenized = tokenizer([input_text], return_tensors='np')
out = model.generate(**tokenized, max_length=128)
with tokenizer.as_target_tokenizer():
    print(tokenizer.decode(out[0], skip_special_tokens=True))

👤 मेरा नाम वेशय था- कि मैं विद्यार्थी हूँ
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_utils_base.py:3856: UserWarning: `as_targ
  warnings.warn(
```

Figure 4.3 Training the language translation model

The above image is about training the language translation model, we have taken the train, test, validation datasets and trained the model and then tested it. We have trained the dataset and fitted the model with 15 epochs, an epoch refers to one complete pass through the entire training dataset during the training phase of a model. During each epoch, the model is exposed to every sample in the dataset exactly once, allowing it to update its parameters based on the training data to improve its performance on the task at hand. After that the model is saved into our system.

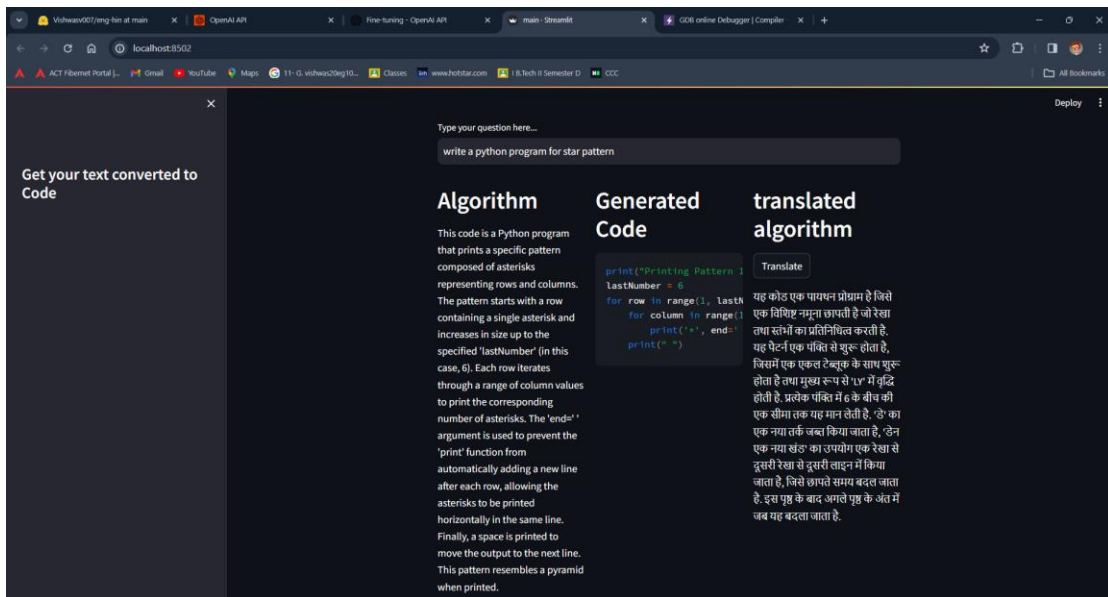


Figure 4.4 User Interface

The interface acts as a bridge between a human's understanding of an algorithm and the code written in a specific programming language Python. Here's how it works:

Input: You provide a written description of the algorithm or the task you want the code to perform. This description should be clear and concise, outlining the steps involved.

Processing: The interface uses its internal mechanisms to translate your written description into the corresponding code. This might involve techniques like natural language processing and pattern recognition.

Output: The interface displays the generated code in the designated output area. Ideally, this code should be functional and achieve the desired outcome based on your description, then provide the algorithm for the code generated by the user input. Now that English algorithm is translated to Hindi.

5. EXPERIMENTAL RESULTS

Used OpenAI API Key, Jupyter Notebook, streamlit to develop this Indian Legal text summarizer. API key will be used to access LLM and fine-tune LLM, Jupyter Notebook will be used to create summarizer with map-reduce paradigm, and streamlit will be used to create interface for end user. Also used the AWS console for the deployment of the app on to the internet where we created a ec2 instance and made it hosted into a public io address.

5.1. Obtain OpenAI API Key

To obtain an OpenAI API key, you need to follow these steps:

1. **Create an OpenAI Account:** If you haven't already, sign up for an account on the OpenAI website.
2. **Navigate to API Settings:** Log in to your OpenAI account and go to the API settings page. This is where you'll find your API key and manage your API usage.
3. **Generate an API Key:** If you haven't generated an API key yet, you'll need to create one. Click on the button or link to generate a new API key.
4. **Copy the API Key:** Once the API key is generated, you'll see it displayed on the screen. Copy this key to your clipboard.
5. **Store the API Key Securely:** It's essential to handle API keys securely. Avoid hardcoding API keys directly into your code, especially if you're sharing your code publicly. Instead, you can use environment variables or configuration files to store and access the API key.

5.2. Setup Jupyter Notebook:

To install and set up Jupyter Notebook, you can follow these steps:

1. **Install Python:** First, you need to have Python installed on your system. You can download and install Python from the official Python website: <https://www.python.org/>. Make sure to check the option to add Python to your system PATH during installation.

2. Install Jupyter Notebook: Once Python is installed, you can install Jupyter Notebook using pip, which is the Python package manager. Open a terminal or command prompt and run the following command:

- `pip install jupyter`
- **Launch Jupyter Notebook:** After the installation is complete, you can launch Jupyter Notebook by running the following command in your terminal or command prompt:
- `jupyter notebook`
- **Accessing Jupyter Notebook:** Once you run the command, your default web browser should open, and you'll be directed to the Jupyter Notebook dashboard. If it doesn't open automatically, you can manually open your web browser and go to `http://localhost:8888/`. Here, you'll see a file browser where you can navigate your filesystem and create or open Jupyter Notebook files.
- **Creating a New Notebook:** To create a new notebook, click on the "New" button in the top right corner and select "Python 3" (or any other available kernel you want to use).
- **Using Jupyter Notebook:** You can now start using Jupyter Notebook. Each notebook consists of cells where you can write and execute Python code, Markdown for documentation, and more. You can execute a cell by pressing Shift+Enter or by clicking the "Run" button in the toolbar.
- **Saving and Closing:** Make sure to save your work regularly by clicking the "Save" button or using the keyboard shortcut Ctrl+S. To close Jupyter Notebook, you can simply close the browser tab or stop the Jupyter Notebook server by pressing Ctrl+C in the terminal or command prompt where it's running.

5.3. Setup Streamlit

To install and set up Streamlit, a popular Python library for creating interactive web applications, follow these steps:

1. Install Python: Ensure you have Python installed on your system. You can download and install Python from the official Python website:

<https://www.python.org/>. Make sure to check the option to add Python to your system PATH during installation.

2. Install Streamlit: You can install Streamlit using pip, the Python package manager. Open a terminal or command prompt and run the following command:

- `pip install streamlit`
- **Create a Streamlit Application:** Once Streamlit is installed, you can create a new Python script (`.py` file) to define your Streamlit application. For example, create a file named `app.py`.
- **Write Streamlit Code:** Write your Streamlit application code in `app.py`. Streamlit provides a simple and intuitive API for creating web applications directly from Python scripts.
- **Run Streamlit Application:** In your terminal or command prompt, navigate to the directory containing `app.py` and run the following command:
 - `streamlit run app.py`
- **Access Your Streamlit App:** Once the Streamlit server starts, it will provide a local URL (usually `http://localhost:8501`) where you can access your Streamlit application in your web browser.
- **Develop Your Application:** You can continue to develop your Streamlit application by modifying `app.py`. Streamlit provides numerous components and features for building interactive data-driven applications, including widgets, charts, layouts, and more. Refer to the Streamlit documentation for detailed information: <https://docs.streamlit.io/>
- **Deploy Your Application:** Once you're satisfied with your Streamlit application, you can deploy it to various platforms, including Streamlit Sharing, Heroku, AWS, or any other hosting provider.

5.4. Libraries Used:

5.4.1. OpenAI

OpenAI Installed enables access to OpenAI's large language models, such as GPT (Generative Pre-trained Transformer), for natural language processing tasks. This involves installing the necessary libraries and dependencies, obtaining an API key for

authentication, integrating OpenAI's functionality into applications or workflows, and utilizing the models for tasks like text generation, summarization, translation, and more. Optional customization allows for fine-tuning or adapting models to specific requirements, empowering developers and researchers with state-of-the-art language processing capabilities.

5.4.2 GPT transformer

The GPT (Generative Pre-trained Transformer) model is a state-of-the-art architecture in natural language processing (NLP), renowned for its ability to generate coherent and contextually relevant text. Built upon the transformer architecture, GPT leverages self-attention mechanisms to capture dependencies between words in a sequence, enabling it to process input data in parallel and effectively model long-range relationships. GPT models are pre-trained on vast amounts of text data using unsupervised learning, allowing them to learn rich representations of language and generalize well to a wide range of NLP tasks. By fine-tuning on task-specific datasets, GPT models can be adapted to perform various tasks such as text generation, language translation, question answering, and more, making them versatile tools for natural language understanding and generation tasks.

5.4.3 AutoTokenizerForSeq2SeqLM

The "AutoTokenizerForSeq2SeqLM" class refers to an auto-generated tokenizer specialized for sequence-to-sequence language modeling tasks in the Hugging Face library. This class automatically selects the appropriate tokenizer based on the specified language model architecture, streamlining the tokenization process for tasks involving sequences of tokens as both input and output. It provides essential functionality for tokenizing input sequences, converting tokens to numerical representations, handling special tokens, and decoding output sequences from model predictions. Compatible with various sequence-to-sequence language model architectures such as T5, Marian, and BART, this tokenizer facilitates efficient and seamless integration into a wide range of natural language processing tasks, enabling rapid development and deployment of sequence-to-sequence models.

5.5 PARAMETERS

The definition of ROUGE-N or ROUGE-L Metrics in terms of precision, recall, and F1 Scores parameters is elucidated as follows.

Precision denotes the fraction of the code content that is pertinent to the original document. It is computed by dividing the count of relevant sentences in the code by the total number of sentences in the code, as illustrated in Equation (1).

$$Precision_{ROUGE-L} = \frac{\text{Common } n - \text{grams in generated code and reference code}}{\text{Number of } n - \text{grams in generated code}} \quad (1)$$

Recall serves as a metric for assessing the efficacy of a summarization algorithm, indicating the proportion of crucial information in the original text that is captured in the algorithm-generated code. The equation for recall is presented in Equation (2).

$$Recall_{ROUGE-L} = \frac{\text{Common } n - \text{grams in generated code and reference code}}{\text{Number of } n - \text{grams in reference code}} \quad (2)$$

The **F1 score** represents the harmonic mean of precision and recall, amalgamating both measures into a single score that achieves a balance between precision and recall, as demonstrated in Equation (3).

$$F1\ score_{ROUGE-L} = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (3)$$

Here, 'n' denotes the length of the n-gram under consideration (e.g., n = 1, 2, 3). These formulas are applicable for assessing the quality of a code generated by an algorithm by juxtaposing it with a reference code.

6. DISCUSSION OF RESULTS

For assessing the effectiveness of system-generated codes, we employed the ROUGE metric, an acronym for Recall-Oriented Understudy for Gisting Evaluation. In this context, "Gisting" refers to the extraction of the primary point from the text [31]. ROUGE serves as an evaluation matrix that compares the generated code with a reference code. A higher ROUGE score signifies a superior alignment between the code and the reference code. ROUGE comprises five variants, including ROUGE-N, ROUGE-S, ROUGE-L, ROUGE-W, and ROUGE-SU [32]. In this study, we specifically employed ROUGE score metrics to analyse various code generation models, where N denotes the length of the ngram, encompassing ROUGE-1 (unigram), ROUGE-2 (bigram), ROUGE-3 (trigram), and so forth. The definition of ROUGE-N or ROUGE-L Metrics in terms of precision, recall, and F1 Scores parameters is elucidated as follows.

Precision in code generation refers to the accuracy of generated code snippets compared to a reference set of correct code. High precision indicates that the generated code closely matches the reference code and contains minimal errors or discrepancies. Precision in code generation is crucial for ensuring that the generated code meets quality standards and performs correctly in practical applications.

$$\text{Precision} = \frac{\text{Number of overlapping tokens between generated and reference code}}{\text{Total number of tokens in generated code}}$$

Recall measures the proportion of relevant or correct tokens present in the reference code that are successfully captured by the generated code. High recall indicates that the generated code successfully captures most of the relevant tokens or features present in the reference code. Recall in code generation is essential for ensuring that the generated code covers all critical aspects and functionalities required by the application.

$$\text{Recall} = \frac{\text{Number of overlapping tokens between generated and reference code}}{\text{Total number of tokens in reference code}}$$

The F1 score represents the harmonic mean of precision and recall, amalgamating both measures into a single score that achieves a balance between precision and recall, as demonstrated in Equation.

$$\text{F1 score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

TABLE 6.1: AVERAGE PRECISION SCORES OF DIFFERENT MODELS

Models	Precision
MarianCG	0.57
RoBERTaMarian	0.40
BERTMarian	0.47
ELECTRAMarian	0.34
LUKEMarian	0.53
Proposed	0.7

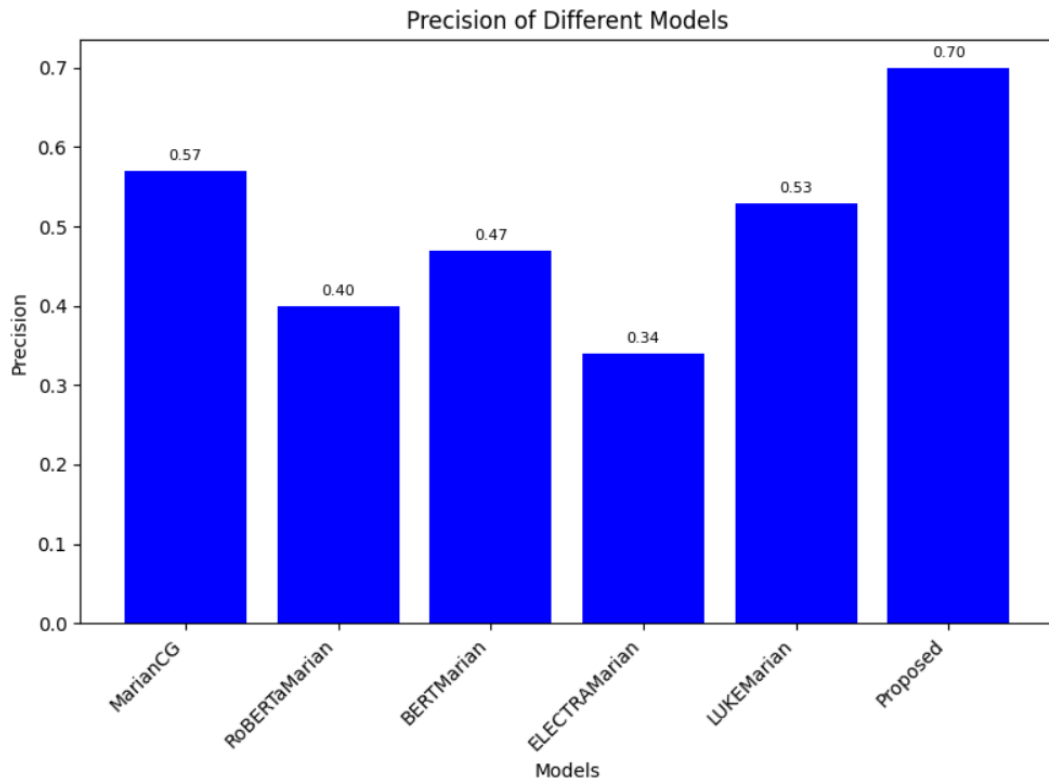
**Figure 6.1 Precision Score Comparison**

Table 6.1 presents the average precision scores of various code generation models tested on CoNaLa dataset. Our observation indicates that the proposed method excels, achieving a precision score of 0.7 surpassing other methods.

The proposed approach demonstrates the highest precision value among the evaluated methods for code generation. This signifies its exceptional ability to generate

accurate code snippets that closely match the reference set, showcasing its potential for enhancing code quality and reliability in various applications.

TABLE 6.2: ACCURACY SCORES OF DIFFERENT MODELS

Models	Accuracy (%)
MarianCG	10.2
RoBERTaMarian	13.8
BERTMarian	12.4
ELECTRAMarian	10.0
LUKEMarian	7.6
Proposed	15.23

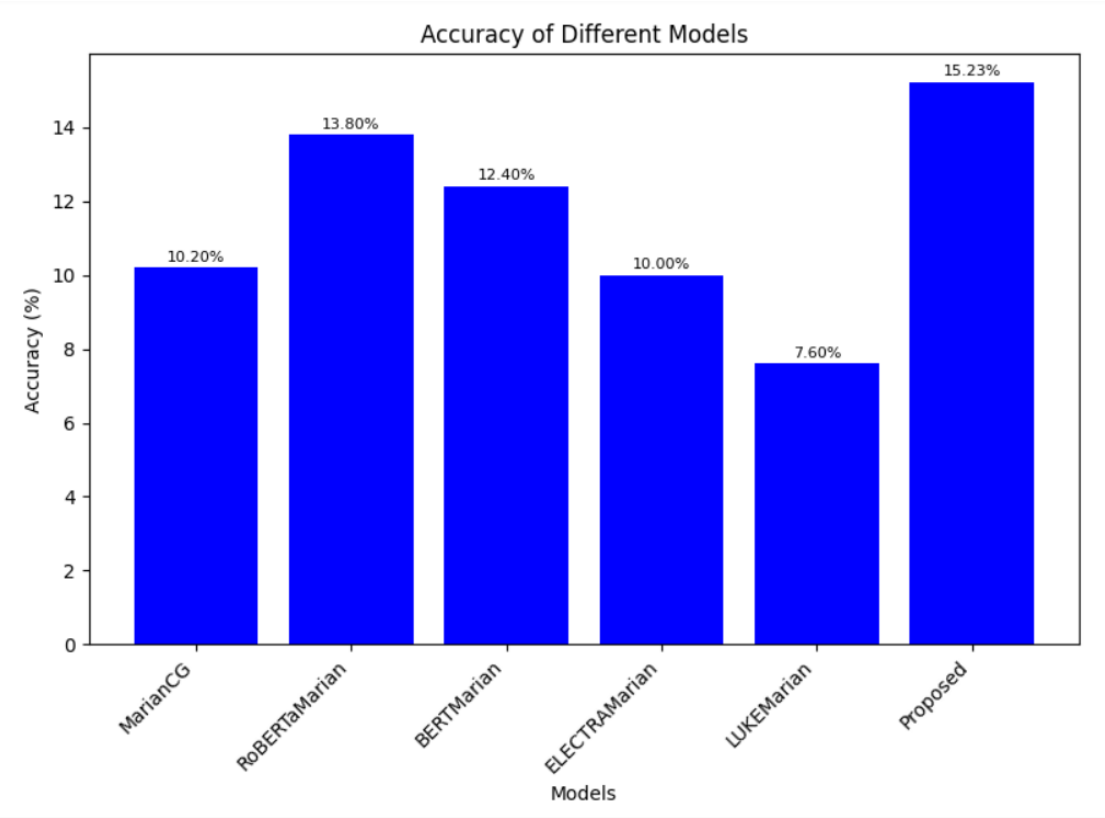


Figure 6.2 Accuracy Score Comparison

Table 6.2 displays the Accuracy score for various code generation models. When we examine the table it is evident that the proposed model has the highest accuracy stating that the code from the generated codes match nearly with the reference codes.

For code generation tasks, a variant of ROUGE could be devised to evaluate the similarity between the generated code and reference (human-written) code. Here's a possible adaptation of ROUGE for code generation:

ROUGE-N for Code: Similar to the original ROUGE-N, this variant would measure the overlap of n-grams (contiguous sequences of n tokens) between the generated code and reference code. It would assess how well the generated code captures the essential components and structure present in the reference code.

ROUGE-L for Code: This variant would compute the longest common subsequence (LCS) between the generated code and reference code. It would focus on identifying the longest matching sequence of tokens, reflecting the similarity in structure and functionality between the two.

ROUGE-W for Code: Similar to ROUGE-W, this variant would weigh the LCS by the length of the matching sequence, emphasizing longer matches. It would highlight the importance of capturing larger segments of code accurately in the generated output.

These adapted ROUGE metrics for code generation could provide a quantitative measure of how well the generated code aligns with the reference code in terms of structure, functionality, and overall similarity. However, it's essential to recognize that code generation tasks have unique characteristics, and evaluating code quality often requires a combination of metrics tailored to specific requirements and constraints of the task. Therefore, while adapted ROUGE metrics can offer insights into code generation quality, they may need to be complemented with other evaluation approaches to provide a comprehensive assessment.

TABLE 6.3: ROUGE SCORES OF DIFFERENT MODELS

Models	Rouge Score
MarianCG	49.62
RoBERTaMarian	44.25
BERTMarian	43.94
ELECTRAMarian	42.42
LUKEMarian	39.84
Proposed	51.23

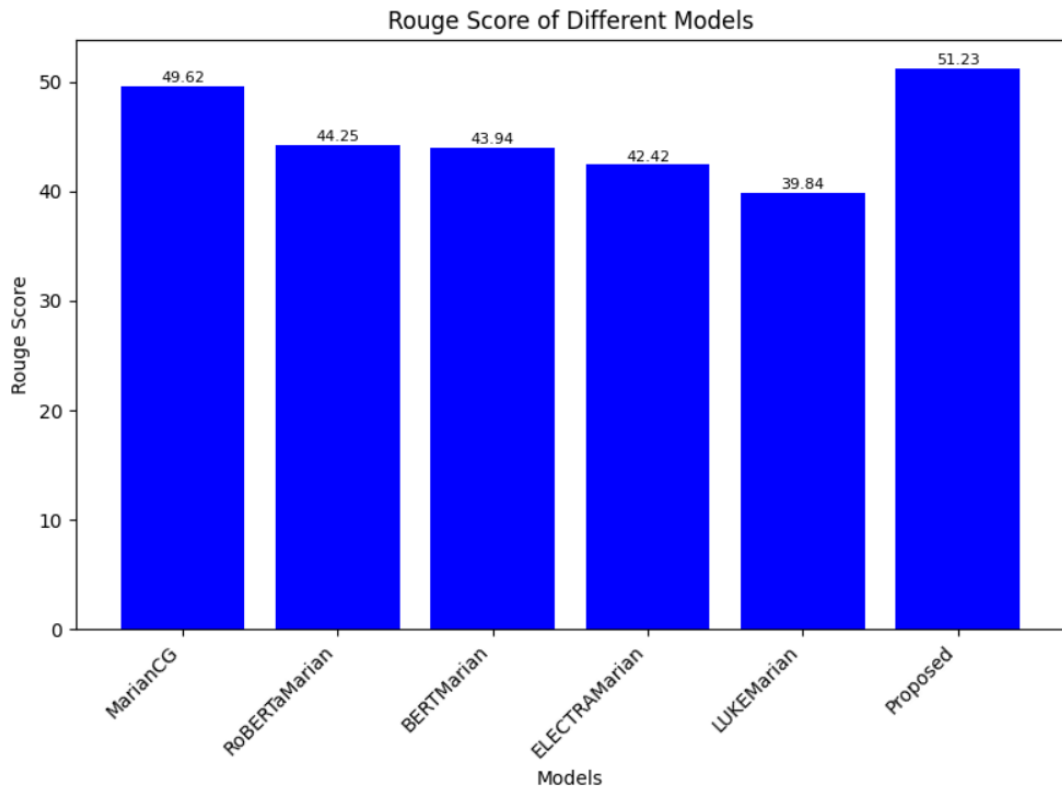
**Figure 6.3 Rouge Score Comparison**

Table 6.3 displays the Average Rouge scores of code generation models on CoNaLa dataset. According to the table, the proposed method achieves the highest ROUGE scores.

BLEU score measures the similarity between the generated code and the reference (human-generated) code based on n-gram overlap. It calculates the precision

of n-grams (contiguous sequences of n tokens) in the generated code compared to the reference code. BLEU score typically ranges from 0 to 1, where a higher score indicates better agreement between the generated and reference code.

The steps to calculate BLEU score for code generation are similar to those for text translation:

Tokenization: Both the generated and reference code are tokenized into individual tokens (e.g., words or subtokens).

N-gram Matching: Calculate the precision of n-grams (unigrams, bigrams, etc.) in the generated code compared to the reference code. Precision measures the proportion of correctly generated n-grams out of all n-grams in the generated code.

Brevity Penalty: Penalize the BLEU score if the generated code is shorter than the reference code to discourage overly concise outputs.

Cumulative BLEU Score: Calculate the cumulative BLEU score by averaging the precision scores across different n-gram sizes, typically up to a certain maximum (e.g., BLEU-4 considers precision up to 4-grams).

TABLE 6.4: BLUE SCORES OF DIFFERENT MODELS

Models	Bleu Score
MarianCG	34.42
RoBERTaMarian	35.73
BERTMarian	32.46
ELECTRAMarian	30.18
LUKEMarian	29.82
Proposed	37.23

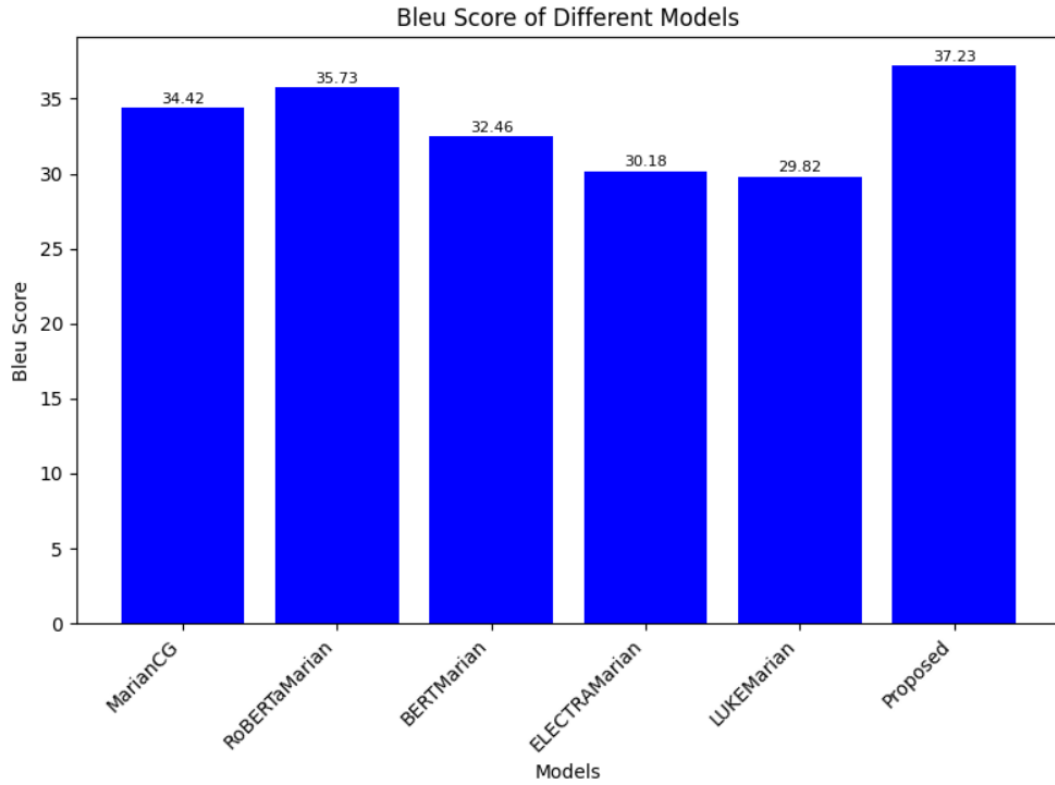


Figure 6.4 BLUE Score Comparison

Table 6.4 displays the BLUE scores of code generation models on CoNaLa dataset. According to the table, the proposed method achieves the highest BLUE scores.

When see the above results clearly tell that proposed method excels in its task when compared to the previous methods. the high BLEU score obtained through evaluation indicates the proposed method's proficiency in generating code snippets that closely resemble the reference set. This achievement underscores the model's efficacy in capturing relevant structures and functionalities, thereby enhancing its utility in code generation tasks across diverse domains.

7. CONCLUSION

The research has demonstrated that the proposed code generation model, when compared to existing models, exhibits superior accuracy and efficiency. Unlike previous models which faced limitations in accurately generating code and providing guarantees of legitimate responses, the proposed method overcomes these challenges by leveraging fine-tuning techniques on the GPT-3 transformer model. By fine-tuning GPT-3 with a large dataset specific to the task at hand, the code generation model excels in performance. Fine-tuning involves training the pre-trained GPT-3 model with additional data related to the specific domain or task. In this case, the large dataset used for fine-tuning likely includes a vast amount of code examples, programming language syntax, and possibly domain-specific knowledge relevant to the code generation task. By training on this dataset, the model learns to understand the intricacies of code structure, semantics, and logic, enabling it to generate code snippets that are not only accurate but also efficient in fulfilling the desired functionality. Moreover, by focusing the model's knowledge base on a specific dataset, the proposed method reduces the risk of generating erroneous or irrelevant code. This targeted approach ensures that the model's outputs are aligned with the requirements and constraints of the given task, thereby enhancing the overall quality of the generated code.

8. REFERENCES

- [1] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In 2022 IEEE Symposium on Security and Privacy, Pages 754-768. IEEE, 2022.
- [2] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In International Conference on Learning Representations, 2022
- [3] Singh, S.M., Meetei, I.S., Singh, T.D., Bandyopadhyay, S., 2021b. Multiple captions. embellished multilingual multi-modal neural machine translation. In: Proceedings of the First Workshop on Multimodal Machine Translation for Low Resource Languages. MMTLRL 2021, pp. 2-11
- [4] Ramesh, G., Doddapaneni, S., Bheemaraj, A., Jobanputra, M., AK, R., Sharma, A Sahoo, S., Diddee, H., Kakwani, D., Kumar, N., et al., 2022. Samanantar: The largest publicly available parallel corpora collection for 11 indic languages. Trans. Assoc Comput. Linguist, 10, 145-162 <http://dx.doi.org/10.1162/tacil-a-00452>.
- [5] Elazar Y, Kassner N, Ravfogel S, Ravichander A, Hovy E, Schütze H, Goldberg Y (2021) Erratum: measuring and improving consistency in pretrained language models. Trans Assoc Comput Linguist9:1407.
- [6] Raffel C, Shazeer NM, Roberts A, Lee K, Narang S, Matena M, Zhou Y, LiW, Liu PJ (2020) Exploring the limits of transfer learning with a unified text-to-text transformer. ArXiv arXiv:1910.10688
- [7] Gad W, Alokla A, Nazih W, Salem AB, Aref M (2021) DlbDeep learning-based transformer to generate pseudo-code from source code. CMC 70:3117–3123.
- [8] Alokla A, Gad W, Nazih W, Aref M, Salem AB (2022) Retrievalbased transformer pseudocode generation. Mathematics 10(4):604.
- [9] Kaur P, Kumar H, Kaushal S (2023) Technology-assisted language learning adaptive systems: a comprehensive review. Int J Cogn Comput Eng 4:301-313

- [10] Javidpanah M, Javadpour A, Rezaei S (2021) ROOA: CloudIDE framework for extension development. *Int J Cogn Comput Eng* 2:165–170.
- [11] Guizzo G, Zhang J, Sarro F, Treude C, Harman M (2023) Mutation analysis for evaluating code translation. *Empir Softw Eng* 29:19
- [12] Athiwaratkun B, Gouda SK, Wang Z, Li X, Tian Y, Tan M, Ahmad WU, Wang S, Sun Q, Shang M, Gonugondla SK, Ding H, Kumar V, Fulton N, Farahani A, Jain S, Giaquinto R, Qian H, Ramanathan MK, Nallapati R, Ray B, Bhatia P, Sengupta S, Roth D, Xiang B (2023) Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*
- [13] Dahal S, Maharana A, Bansal M (2021) Analysis of tree-structured architectures for code generation. In: *Findings of the association for computational linguistics: ACL-IJCNLP 2021*, pp 4382–4391
- [14] Qin P, Tan W, Guo J, Shen B, Tang Q (2021) Achieving semantic consistency for multilingual sentence representation using an explainable machine natural language parser (mparser). *Appl Sci* 24:11699
- [15] Tang Z, Shen X, Li C, Ge J, Huang L, Zhu Z, Luo B (2022) Asttrans: code summarization with efficient tree-structured attention. In: *2022 IEEE/ACM 44th international conference on software engineering (ICSE)*, pp 150–162
- [16] Shin R, Lin CH, Thomson S, Chen C, Roy S, Platanios EA, Pauls A, Klein D, Eisner J, Van Durme B (2021) Constrained language models yield few-shot semantic parsers. *arXiv preprint arXiv:2104.08768*
- [17] Lano K, Xue Q (2023) Code generation by example using symbolic machine learning. *SN Comput Sci* 4:1–23
- [18] Le THM, Chen H, Babar MA (2020) Deep learning for source code modeling and generation. *ACM Comput Surv (CSUR)* 53:1–38
- [19] Norouzi S, Tang K, Cao Y (2021) Code generation from natural language with less prior knowledge and more monolingual data. In: *Proceedings of the 59th Annual*

meeting of the association for computational linguistics and the 11th international joint conference on natural language processing (volume 2: short papers), pp 776–785.

[20] Orlanski G, Gittens A (2021) Reading stackoverflow encourages cheating: adding question text improves extractive code generation. arXiv preprint arXiv:2106.04447.

[21] Beau N, Crabbé B (2022) The impact of lexical and grammatical processing on generating code from natural language. arXiv preprint arXiv:2202.13972

[22] Wang Z, Cuenca G, Zhou S, Xu FF, Neubig G (2022) Mconala: a benchmark for code generation from multiple natural languages. arXiv preprint arXiv:2203.08388

[23] Kusupati U, Ailavarapu VRT (2022) Natural language to code using transformers. ArXiv arXiv:2202.00367

[24] Al-Hossami E, Shaikh S (2022) A survey on artificial intelligence for source code: a dialogue systems perspective. ArXiv arXiv:2202.04847

[25] Ni P, Okhrati R, Guan S, Chang VI (2022) Knowledge graph and deep learning-based text-to-graphQL model for intelligent medical consultation chatbot. Inf Syst Front 2022:1–20.

[26] Ross SI, Martinez F, Houde S, Muller M, Weisz JD (2023) The programmer’s assistant: Conversational interaction with a large language model for software development.