A
Mini Project Report on

Code Converter Using NLP

Submitted in partial
fulfillment of the requirements for the degree of
BACHELOR OF ENGINEERING
IN
Computer Science & Engineering
Artificial Intelligence & Machine Learning

by

Sumit K Thakur (23106036)
Reva G Tol (23106083)
Mayankkant L Tiwari (23106037)
Tejas H Thorat (23106120)

Under the guidance of

Prof. Vijaya Bharathi

Department of Computer Science & Engineering
(Artificial Intelligence & Machine Learning)
A. P. Shah Institute of Technology
G. B. Road, Kasarvadavali, Thane (W)-400615
University Of Mumbai
2024-2025

A. P. SHAH INSTITUTE OF TECHNOLOGY

CERTIFICATE

This is to certify that the project entitled "**Code Converter Using NLP"** is a bonafide work of Sumit Thakur (23106036), Tejas Thorat (23106120), Reva Tol (23106083), Mayankkant Tiwari (23106037) submitted to the University of Mumbai in partial fulfillment of the requirement for the award of **Bachelor of Engineering** in **Computer Science & Engineering (Artificial Intelligence & Machine Learning).**

Prof. Vijaya Bharathi	Dr. Jaya Gupta
Mini Project Guide	Head of Department

A. P. SHAH INSTITUTE OF TECHNOLOGY

Project Report Approval

This Mini project report entitled "**Code Converter Using NLP**" by **Sumit Thakur, Reva G. Tol, Mayankkant Tiwari and Tejas Thorat** is approved for the degree of ***Bachelor of Engineering*** in ***Computer Science &Engineering***, (AIML) ***2024-25***.

External Examiner: ____________________

Internal Examiner: ___________________

Place: APSIT, Thane

Date:

Declaration

We declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Sumit Thakur	Reva Tol	Mayank Tiwari	Tejas Thorat
(23106036)	(23106083)	(23106037)	(23106120)

ABSTRACT

Manual code translation, particularly between programming languages with fundamentally different paradigms such as C and Python, remains a labor-intensive and error-prone task. C, being a low-level, statically typed language with explicit memory management, contrasts sharply with Python's high-level, dynamically typed syntax and abstracted memory handling. These differences introduce significant challenges in accurately converting logic, structures, and semantics from one language to another. Traditional rule-based tools often fall short when faced with non-trivial or custom code, leading to incorrect translations and increased developer burden. The **Code Converter** project addresses this problem through an AI-driven approach, leveraging the capabilities of the CodeT5 model—a transformer-based architecture pre-trained on source code and fine-tuned for the specific task of translating C programs into Python.

The model was trained on a carefully curated dataset comprising paired C and Python code snippets covering a range of foundational programming constructs, including loops, conditionals, functions, and recursion. This dataset was compiled using both AI-generated and manually written examples to ensure quality, correctness, and educational relevance. Fine-tuning was conducted using cloud-based resources, enabling efficient training without the need for extensive local hardware. The resulting model demonstrates strong performance in translating standard algorithmic logic and educational code samples with syntactic and semantic fidelity. To enhance accessibility and usability, a web interface was built using Flask, allowing users—ranging from students to professional developers—to submit C code and receive the corresponding Python translation in real-time.

Beyond its practical utility, the Code Converter also serves as a valuable platform for learning and experimentation in machine learning, natural language processing, and software engineering. It illustrates the adaptability of modern transformer models like CodeT5 for cross-language code understanding and generation. While the system currently handles basic to intermediate code structures effectively, future work will focus on expanding dataset diversity, improving support for complex C features such as pointer arithmetic, memory allocation, and advanced data structures, and extending the translation capability to other language pairs. This project thus represents a promising step toward more intelligent, reliable, and scalable solutions for code migration and programming education.

Keywords: Code Translation, CodeT5, C to Python, AI, Machine Learning, NLP

Index

CHAPTER 1
INTRODUCTION

1. INTRODUCTION

1.1 Introduction

In the evolving landscape of software development, code translation between programming languages has emerged as a critical necessity. Whether driven by the need for platform migration, cross-language integration, legacy code modernization, or educational exploration, developers and learners often find themselves needing to convert existing source code from one language to another. Among the most prominent and challenging language pairs is C to Python. C, known for its procedural and low-level capabilities, continues to dominate domains like embedded systems, operating systems, and performance-critical applications. Python, by contrast, has surged in popularity due to its readability, high-level abstractions, and ease of use, becoming a preferred language in education, data science, artificial intelligence, and web development. The fundamental differences in these languages—from their syntax and memory models to their execution paradigms—make manual translation not only time-consuming but also highly error-prone, particularly for those not deeply familiar with both languages. Manual code translation, while feasible for simple snippets, quickly becomes complex and risky when handling larger codebases, deeply nested logic, or language-specific features. Translating C code to Python is not a mere matter of substituting syntax; it requires careful reinterpretation of constructs. C offers explicit control over memory through pointers and static typing, while Python abstracts away such details using dynamic typing and high-level data structures. For example, translating a manually managed array in C to a Python list requires more than syntactic changes—it involves understanding how memory is allocated, accessed, and iterated in both languages. Similarly, translating a recursive function, pointer-based operation, or bit-level manipulation requires semantic awareness, not just syntax recognition. These challenges become bottlenecks in workflows where quick iteration is essential, such as in startup environments, prototyping, or classroom labs. In many cases, developers resort to writing the entire logic anew in Python rather than risking inconsistencies through manual translation, which increases development time and effort.

The issue becomes even more prominent in academic settings, where students are often introduced to programming via C but are expected to use Python in data science or machine learning coursework. The cognitive load of learning both languages and mentally translating logic between them can hinder comprehension. Students unfamiliar with memory management or low-level operations might struggle to convert a C program into Python accurately, leading to conceptual misunderstandings. Similarly, educators seeking to demonstrate language equivalencies must manually write dual examples, often constrained by time. Despite the frequency and importance of such translation tasks, existing tools fall short of delivering meaningful support. Most code converters available today are either web-based or part of IDE extensions, and rely on static rule-based mapping. These tools apply predefined transformations that fail in the face of custom user logic or nuanced control flows. They also lack adaptability—meaning they cannot generalize well across varied problem types, nor can they learn from user interactions or data over time. To address these gaps, the **Code Converter** project introduces a novel, AI-powered solution for intelligent C-to-Python code translation. At its core lies the **CodeT5 model**, a powerful transformer-based architecture developed by Salesforce Research and fine-tuned in this project to understand and translate source code across languages. Built upon the T5 (Text-to-Text Transfer Transformer) framework, CodeT5 extends NLP capabilities to programming tasks by treating code as a specialized form of structured text. The model is pre-trained on massive datasets from public code repositories like GitHub, covering multiple programming languages and coding patterns. By further fine-tuning this model on a curated dataset of C and Python code pairs, the Code Converter project empowers it to learn meaningful relationships between constructs such as loops, conditionals, arithmetic logic, and function calls. This machine learning-based approach significantly outperforms traditional rule-based methods by understanding not just the *syntax* but also the *semantics* and *intent* behind code. The dataset used to train the Code Converter was thoughtfully assembled to ensure both diversity and educational relevance. Code snippets were collected and aligned using a hybrid strategy.

GPT-based tools such as ChatGPT and DeepSeek were utilized to generate a foundational pool of paired code samples, covering a wide range of beginner to intermediate-level programs, including pattern printing, number theory problems, basic string manipulation, and recursion. To ensure correctness and remove bias, these examples were then manually reviewed, and additional hand-written examples were added to enrich the dataset. These custom examples focused on constructs typically taught in undergraduate programming courses: conditional branching, iterative loops (for, while), logical operators, function usage, and input/output operations. Each entry in the dataset was formatted as a structured JSON object, with clear delineation of source (C) and target (Python) code. This structure helped streamline preprocessing and ensured the model received consistent input-output pairs during training. The fine-tuning process was conducted on **Google Colab**, utilizing its free access to cloud-based GPUs such as the NVIDIA Tesla T4 and K80. This choice offered several benefits: it eliminated the need for expensive local hardware, reduced training time through GPU acceleration, and enabled easy integration with cloud storage via Google Drive. The software stack included Python 3 running within a Colab notebook environment, supported by key libraries such as transformers (for the CodeT5 architecture), torch (PyTorch framework), tokenizers, sklearn, json, and flask for deployment. Preprocessing routines ensured that code was appropriately tokenized, padded, and batched for efficient GPU utilization. Training logs and loss metrics were monitored to identify overfitting and tune hyperparameters accordingly. Despite being a lightweight deployment compared to enterprise-grade machine learning systems, the setup proved sufficient for achieving high-quality translations on common code types. Once the model was fine-tuned, it was integrated into a user-facing application built using Flask, a lightweight Python web framework. The application allows users to input C code through a simple web interface and view the corresponding Python translation in real-time.

In conclusion, the **Code Converter** project is a robust application of machine learning in software development. It effectively demonstrates how AI can be used not just to automate but to intelligently assist in one of the more intellectually demanding tasks in programming—language translation. By leveraging the capabilities of the CodeT5 model and grounding it in a carefully curated dataset, the project achieves reliable, accurate translation of C programs into Python. The user-friendly interface ensures accessibility, while the educational orientation enhances its value in learning environments. As AI continues to influence every layer of the tech stack, from IDEs to deployment pipelines, projects like Code Converter offer a glimpse into the future of collaborative, AI-enhanced programming—one where human intent is amplified, not replaced, by machine intelligence.

CHAPTER 2
LITERATURE SURVEY

2. LITERATURE SURVEY

In the last few years, there has been a rapid and significant rise in the application of deep learning to software engineering, especially in tasks involving source code understanding, generation, and translation. This surge has been largely driven by the success of transformer-based architectures such as BERT (Bidirectional Encoder Representations from Transformers), GPT (Generative Pretrained Transformer), and T5 (Text-to-Text Transfer Transformer), which have revolutionized natural language processing (NLP) through their ability to learn complex language patterns and semantic relationships. These models, initially designed for human languages, have shown great potential when applied to programming languages, due to the structural and syntactic similarities between code and natural language. As a result, researchers have adapted these models to develop more sophisticated tools for source code tasks, including code summarization, classification, translation, generation, and documentation【2】.

The foundation of this progression began with GPT, introduced in 2018, which uses a unidirectional transformer decoder architecture that excels in generative tasks. GPT laid the groundwork for autoregressive models capable of producing coherent and contextually relevant output sequences. Following this, BERT was released in 2019 as a bidirectional encoder, which significantly improved the performance of tasks involving comprehension and contextual understanding by analyzing the entire sentence at once rather than one token at a time. Although both GPT and BERT demonstrated powerful capabilities, they were not originally designed to handle programming languages or code-specific tasks. As such, they struggled with modeling programming language semantics and failed to grasp nuances such as syntax rules, logical flow, or the functional roles of identifiers like variables and functions.

To address this gap, CodeBERT was introduced in 2020 as one of the first major attempts to adapt BERT specifically for software engineering tasks. Developed by Microsoft Research, CodeBERT extended BERT's architecture and was pretrained on the CodeSearchNet dataset, which includes millions of code snippets and associated natural language descriptions from several programming languages. CodeBERT demonstrated strong performance on code-related tasks such as code search, documentation generation, and classification. However, its encoder-only design limited its use for code generation tasks, as it lacked a decoder component necessary for generating new sequences of tokens. This limitation prompted the development of encoder-decoder models that could handle both understanding and generation tasks more effectively.

PLBART, introduced in 2021, was one such model that built upon the BART architecture—a combination of BERT's encoder and GPT's decoder. PLBART supported a broader range of tasks, including code summarization, translation between programming languages, and code synthesis. It extended the pretraining process to encompass both programming and natural languages, which allowed it to better understand context and generate more meaningful code. However, a major drawback of PLBART was its limited ability to fully interpret the semantics of identifiers. In programming, identifiers such as variable and function names often carry important semantic information, and ignoring their significance can lead to incorrect or ambiguous code generation.

To address the shortcomings of previous models, Wang et al. proposed CodeT5 in 2021—a transformer-based model specifically designed for programming languages【3】【6】. Built on the T5 framework, CodeT5 introduced an identifier-aware encoder-decoder architecture that incorporated multiple novel pretraining objectives, including Masked Span Prediction (MSP), Identifier Tagging (IT), Masked Identifier Prediction (MIP), and Bimodal Dual Generation. These objectives were tailored to help the model better understand the syntactic structure of code, identify key components, and generate high-quality output. CodeT5 was trained on over 8 million code samples from GitHub and the CodeSearchNet dataset, and achieved state-of-the-art results on various CodeXGLUE benchmarks across tasks like code summarization, refinement, and clone detection. It unified various software engineering tasks under a single architecture, allowing for flexible adaptation and fine-tuning.

Despite its impressive performance, CodeT5 was primarily focused on raw code generation and lacked the ability to produce human-readable and well-documented code. This presented a critical limitation, especially in real-world development scenarios where explainability and maintainability of code are essential. Recognizing this need, researchers at the American University in Cairo introduced a novel downstream task called ConcodePlus in 2023. ConcodePlus aims to generate Python code with inline comments, given a natural language prompt. This task is particularly relevant for educational tools, documentation support

systems, and explainable AI applications, where the clarity and readability of code are just as important as its correctness.
To support this task, the researchers developed a specialized dataset named PyPlus. Derived from the Python subset of CodeSearchNet, PyPlus contains over 250,000 aligned pairs of natural language prompts and Python code with inline documentation. The dataset was carefully curated to ensure both syntactic correctness and the inclusion of meaningful comments, thereby promoting the generation of developer-friendly code【3】. For model training, the researchers used CodeT5-small—a lightweight variant of CodeT5 with approximately 60 million parameters. The model was fine-tuned on the PyPlus dataset using a modest computing setup (a local machine with 16GB RAM), demonstrating the feasibility of training effective models with limited hardware resources.
The performance of the ConcodePlus model was evaluated using a combination of traditional NLP and code-specific metrics, including BLEU-4, CodeBLEU, perplexity, and exact match scores. The results showed a strong balance between code functionality and the relevance of generated comments. The inclusion of inline documentation allowed the model to produce outputs that were not only syntactically correct but also explainable, which is crucial for educational use cases and collaborative development. This integration of natural language understanding with code generation represents a significant advancement in the direction of human-centered AI tools.
However, several challenges continue to hinder progress in this domain. One of the most pressing issues is the lack of large-scale, high-quality annotated datasets tailored for documentation-aware code generation. Although datasets like PyPlus are a step in the right direction, the field still lacks the diversity and richness needed to train robust models capable of generalizing across programming paradigms and domains. Furthermore, the computational limitations faced by many research groups prevent them from fine-tuning larger models like CodeT5-base or CodeT5-large, which could offer even better performance【3】【6】. Additionally, the evaluation metrics commonly used in NLP, such as BLEU, often fail to capture the semantic accuracy or documentation quality of generated code. This points to the need for more sophisticated, code-aware evaluation frameworks that can assess both the correctness and explainability of the outputs.
In conclusion, the evolution of transformer models from BERT and GPT to CodeBERT, PLBART, and CodeT5 has significantly advanced the state-of-the-art in automated code generation【2】【5】【6】. The recent adaptation of CodeT5 for the ConcodePlus task represents a pivotal moment in this journey, highlighting a growing emphasis on human-readable, well-documented code. These developments not only improve the practical utility of code generation models but also open up new possibilities in educational technology, explainable AI, and intelligent development environments. As the field continues to mature, future research will need to focus on expanding datasets, improving evaluation methods, and optimizing models for better performance under real-world constraints. The integration of deep learning with software engineering thus holds great promise for transforming how developers write, understand, and interact with code.

CHAPTER 3

PROBLEM STATEMENT

3. Problem Statement

Translating code from C to Python is a common yet challenging task in both academic and professional settings. Manual conversion is often error-prone, with issues ranging from syntax mismatches (like semicolon usage or indentation) to deeper logical inconsistencies due to fundamental differences in language structure and behavior. It is also time-consuming, especially for large codebases, requiring a solid understanding of both languages to maintain accuracy and functionality. For beginners, the gap between procedural C and high-level Python creates a steep learning curve, as they struggle with varying syntax, memory management, and language conventions. Furthermore, existing code conversion tools are typically rule-based and lack contextual understanding, resulting in rigid and inaccurate translations. These challenges highlight the need for an intelligent, adaptive system that can reliably bridge the gap between C and Python.

CHAPTER 4

EXPERIMENTAL SETUP

4.Experimental Setup

A high-quality dataset is essential for the success of any machine learning model, especially in the context of source code translation, where both syntactic structure and semantic integrity must be preserved across different programming languages. For this project, a custom dataset was developed consisting of aligned C and Python code pairs. The goal was to create a balanced, accurate, and educationally relevant dataset capable of training the CodeT5 model effectively on the C-to-Python translation task. The dataset was constructed using a hybrid approach that combined AI-assisted generation with manual curation to ensure both scale and quality.

To accelerate the data collection process, GPT-based code generation tools such as ChatGPT and DeepSeek were employed. These tools enabled the automatic generation of aligned C and Python code snippets across a wide range of programming scenarios, including control structures, loops, conditionals, array manipulations, string handling, and mathematical functions. This AI-assisted method allowed for rapid dataset expansion while maintaining structural and logical consistency between code pairs. In parallel, manually curated code samples were written and thoroughly reviewed by team members to address scenarios commonly encountered in academic contexts, particularly in introductory programming courses. These manually crafted samples included examples like for and while loops, prime number checks, number reversal, pattern printing, user input/output processing, and basic function calls—ensuring the dataset addressed both syntactic complexity and logical alignment.

Each code pair in the dataset was organized in a clean and standardized JSON format to facilitate easy preprocessing and model ingestion. The JSON structure included two fields: "source" representing the original C code and "target" containing the corresponding translated Python code. This format made it simple to tokenize and batch the dataset for training while maintaining clarity and modularity.

The resulting dataset offered several key advantages. First, it had strong educational relevance, focusing on examples that are familiar to learners and therefore useful in both academic and tutorial settings. Second, it ensured logical alignment between the source and target code, improving the quality of the learning signal during model training. Third, by including a diverse set of programming constructs—such as loops, recursion, and conditionals—it encouraged generalization beyond one-to-one syntax mappings. Finally, although the dataset was relatively small compared to large-scale corpora, its examples were thoroughly validated to be free of syntactic and logical errors, making it a high-quality resource for fine-tuning.

Additional efforts were made to balance simple and complex examples to ensure that the model learns translation patterns effectively across difficulty levels. In future expansions, the dataset may be further enriched with community-sourced code or scraped from open educational resources, provided copyright and ethical guidelines are followed. Overall, this hybrid dataset construction strategy successfully blended scalability, accuracy, and domain relevance, ultimately contributing to improved model robustness and performance in translating real-world C code into Python.

CHAPTER 5

PROPOSED SYSTEM & IMPLEMENTATION

5. Proposed system & Implementation

The system architecture of the C-to-Python translation application is designed to offer a seamless user experience, combining the power of a fine-tuned transformer-based model with a lightweight and accessible user interface. The core goal of the architecture is to facilitate the translation of C code into its Python equivalent through a simple, browser-based platform that hides the complexity of machine learning operations from the end-user. The application is structured around a three-stage pipeline: input, processing, and output, all orchestrated within a Flask web framework.

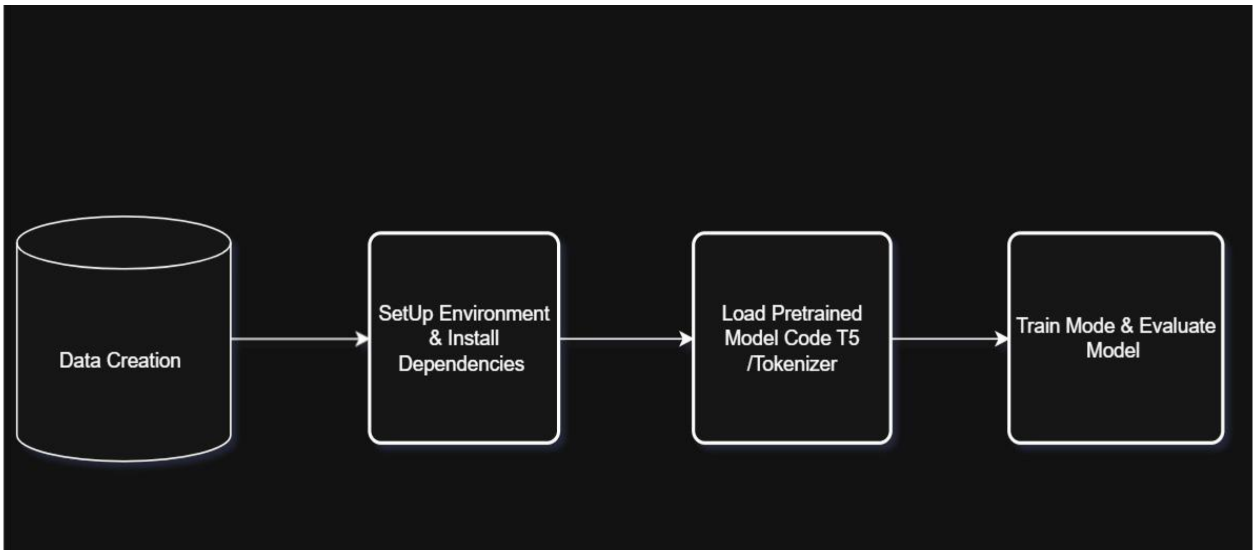
Data Creation
SetUp Environment & Install Dependencies
Load Pretrained Model Code T5 /Tokenizer
Train Mode & Evaluate Model

Fig 5.1
Overall System Architecture

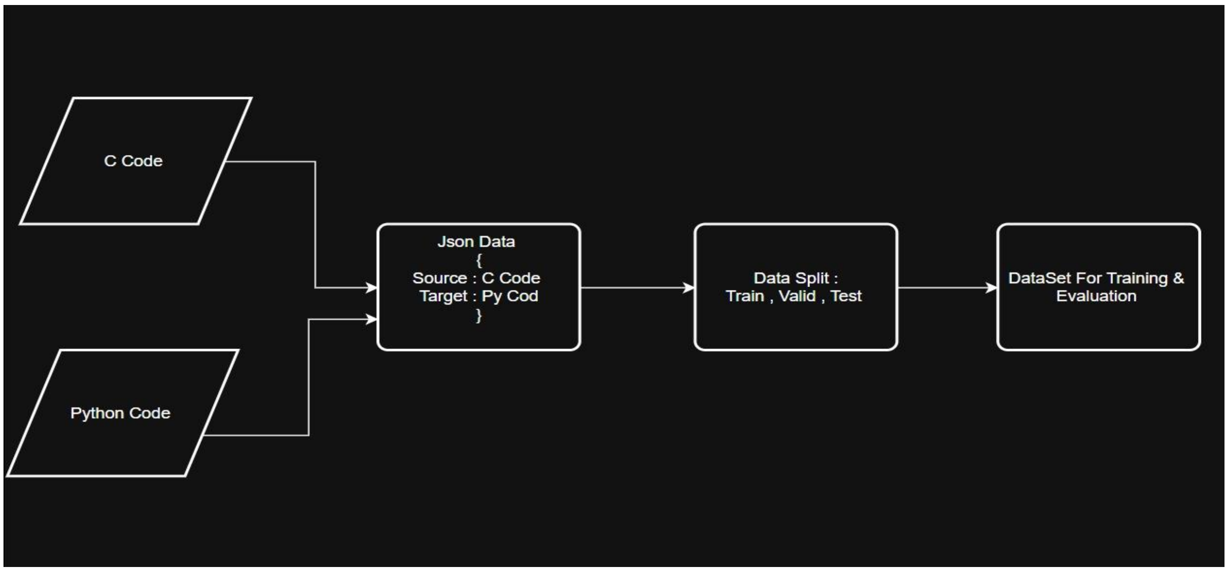
C Code
Python Code
Json Data { Source : C Code Target : Py Cod }
Data Split : Train , Valid , Test
DataSet For Training & Evaluation

Fig 5.2
Dataset Creation

The C-to-Python code translation system is powered by a transformer-based architecture, specifically **HuggingFace's CodeT5-base** model, which has been widely recognized for its effectiveness in code understanding and generation tasks. This model forms the computational backbone of the application, enabling it to accurately translate programming logic across different languages. The CodeT5-base variant was chosen for its balance between performance and scalability, especially when fine-tuned with a carefully curated dataset.

The **model training process** involved fine-tuning the pre-trained CodeT5-base model using the **PyTorch** framework and **HuggingFace Transformers** library. The dataset, composed of semantically aligned C and Python code pairs, was tokenized using the appropriate tokenizer and passed through the model during training. This process enabled the model to learn patterns and structural transformations required to convert procedural C code into Python syntax effectively. The fine-tuning helped the model adapt to the specific nuances of programming constructs across the two languages, such as indentation-based blocks in Python versus braces in C.

On the frontend, a **Flask-based web interface** was developed to serve as the interaction point for users. The interface consists of a clean and responsive layout with a text field for users to input their C code. Upon submission, the backend handles the request, processes the input through the tokenizer and model, and returns the translated Python code, which is dynamically displayed on the page. This real-time response mechanism makes the tool highly interactive and easy to use.

5.3 Advantages

- Accuracy: Learns from real code structure; avoids rigid mappings.
- Educational: Helps understand syntax and structure differences.
- Time-Efficient: Automates the translation process.
- Scalable: Can be extended to support more languages (Java, JS, etc.)

5.4 Challenges Faced

- Lack of large diverse dataset.
- Model struggles with:
    - Pointers
    - Dynamic memory (malloc, free)
    - Recursion
- Addressed by:
    - Structuring dataset from simple to complex.
    - Improving token-level labeling.
    - Planning to integrate reinforcement learning.

CHAPTER 6

CONCLUSION

The **Code Converter** project successfully demonstrates the integration of artificial intelligence and natural language processing in automating programming language translation. By fine-tuning the CodeT5 transformer model on a custom dataset of C and Python code pairs, the system effectively performs accurate and context-aware translations of basic to intermediate C programs into Python. This not only reduces the manual effort traditionally required for such tasks but also minimizes common errors associated with syntax and logic mismatches. The use of a Flask-based web interface further enhances accessibility, allowing users to interact with the model through a simple and intuitive platform.

Beyond its practical utility, the project serves as a robust learning experience in the interdisciplinary fields of AI, machine learning, and software development. It highlights the real-world applicability of NLP models in domains beyond natural language, specifically in programming and code understanding. Developing the Code Converter involved critical stages such as dataset curation, model fine-tuning, and deployment, each of which provided insight into the challenges of training AI systems to reason about code structure and semantics. The educational value of the system also extends to its users, particularly students, who benefit from clear, side-by-side translations that reinforce their understanding of language syntax, logic design, and cross-paradigm programming principles.

Looking forward, there are significant opportunities for enhancing the system's capabilities and expanding its scope. Future improvements will focus on incorporating more diverse and complex datasets, enabling the model to handle advanced C constructs like pointers, structs, memory management, and file operations. Additionally, expanding support to include other programming languages such as Java, JavaScript, or Rust can transform the Code Converter into a versatile multilingual development assistant. Further refinement of evaluation metrics—both quantitative and qualitative—will also help in assessing translation fidelity and logical accuracy at scale. Ultimately, this project not only delivers a functional tool but also lays the groundwork for continued innovation at the intersection of AI and software development.

References

Research Papers

[1] Apte, Onkar & Agre, Shubham & Adepu, Rajendraprasad & Ganjapurkar, Mandar. (2021). C TO PYTHON PROGRAMMING LANGUAGE TRANSLATOR. 10.1729/Journal.26932.

[2] Baptiste Roziere* Facebook AI Research Paris-Dauphine University broz@fb.com Marie-Anne Lachaux⇤ Facebook AI Research malachaux@fb.com Lowik Chanussot Facebook AI Research lowik@fb.com Guillaume Lample Facebook AI Research glample@fb.com

[3] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, Xiaoguang Mao

Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 1509-1521, 2023

Links

[4] Onkar Apte (2021) – C to Python Language Translator
(PDF) C TO PYTHON PROGRAMMING LANGUAGE TRANSLATOR

[5] “Unsupervised Translation of Programming Languages” – Facebook Research
https://paperswithcode.com/paper/unsupervised-translation-of-program

[6] Salesforce CodeT5 Model
Salesforce/codet5-base · Hugging Face