Natural Language Processing Open AI API

Internship Report

Bachelor of Technology
in
Information technology

by

Deepak Soni
(2021BITE051)

Under the Supervision of

Dr. Anil Kumar Singh

Department of Information technology

NATIONAL INSTITUTE OF TECHNOLOGY

SRINAGAR

March,2024

Acknowledgements

I would like to express my sincere gratitude to Dr.Anil Kumar Singh, my internship supervisor, for their guidance, encouragement, and valuable insights throughout the project. Special thanks to the entire team at Indian Institute of Technology, BHU for providing an enriching environment and fostering a culture of innovation.

DEEPAK SONI
(2021BITE051)

NOC

NATIONAL INSTITUTE OF TECHNOLOGY SRINAGAR
(An autonomous Institute of National Importance under the aegis of Ministry of Education, Govt. of India)
DEPARTMENT OF TRAINING & PLACEMENT
Tel/Fax: +91-9419226538,9419226574 Extn: 2130/31 Email: placements@nitsri.net
Hazratbal , Srinagar Jammu and Kashmir, 190006, INDIA

NO: NIT/T&P/INTP/2023-24/ITE/036
Date: 14-11-2023

TO WHOM IT MAY CONCERN

Subject: **Permission of Internship online/offline for student of NIT Srinagar.**

In – Plant/on–the –project internship/Practical Training is an important part of our engineering curriculum. This internship/training is regarded as a vital component of engineering education and is an indicator of extent of field experience, which is very essential for attaining excellence in the technical education. In this context, **Mr. /Ms. Deepak Soni,** Enrolment No: **2021BITE051** pursuing B. Tech in INFORMATION TECHNOLOGY DEPARTMENT (2021-2025) in this Institute has completed his/her 4th semester of the degree (pursuing 5th semester) and is interested in 45 days internship in your esteemed organization.

It will be highly appreciated if your organization provides him/her a chance to get an exposure to some project related to him/her branch of engineering online/offline that is being carried out by your organization during winter vacation from 20th December 2023 to 15th February 2024.

We fervently hope that you will accede to our request and allow him/her to pursue him/her internship in your esteemed organization. The student has been advised to abide by the rules and regulation of your organization. Also, the student has to submit completion report and certificate in the training & placement department after completion of the internship, failing this his/her internship will be deemed incomplete.

Associate TPO (Internships)
Training and Placement
NIT Srinagar

Associate TPO (Internships)
Training & Placement Department
National Institute of Technology
Srinagar, J&K

Certificate

भारतीय प्रौद्योगिकी संस्थान काशी हिन्दू विश्वविद्यालय

IIT INDIAN INSTITUTE OF TECHNOLOGY BANARAS HINDU UNIVERSITY

(Serving the nation since 1919)

Department of Computer Science & Engineering

Indian Institute of Technology (Banaras Hindu University)

Phone : 0542-2307055-56, - 0542-6702764 Fax : 0542-2368428

Internship Completion Certificate

To whom it may concern

This is to certify that Mr. DEEPAK SONI bearing enrollment no. 2021BITE051 studying for the Bachelor of Technology in the Department of Information Technology at the National Institute of Technology Srinagar has completed Winter Internship under my guidance from 22/12/2023 to 30/01/2024.

During the course of the internship, he worked on the project titled "Programming Language conversion using OPENAI API ".

He was found sincere and hardworking during this tenure, and the outcome of his work is satisfactory. He has good moral character.

I wish him all the best in his future endeavors.

With Regards,

सह आचार्य/Associate Professor

संगणक विज्ञान एवं अभियान्त्रिकी विभाग/Department of Computer Sc. & Engg

भारतीय प्रौद्योगिकी संस्थान/Indian Institute of Technology

(बनारस हिन्दू यूनिवर्सिटी)/(Banaras Hindu University)

Dr. Anil Kumar Singh
Associate Professor
Department of Computer Science and Engineering

Abstract

This internship centered on the creation of an advanced tool aimed at seamlessly translating Java code into its corresponding Python representation. The primary focus was on enhancing efficiency in handling extensive Java codebases. The project employed a multi-faceted approach, encompassing code tokenization and harnessing the capabilities of the GPT-3.5 API to achieve high-quality code conversions. This report offers an exhaustive overview of the methodologies utilized, the challenges encountered throughout the development process, and the overarching impact of the resulting solution.

The methodology began with the systematic tokenization of Java code, facilitating a granular understanding of its structure and syntax. Leveraging the GPT-3.5 API, the tool then generated Python code that mirrored the functionality of the original Java source. The synergy between tokenization and AI-driven conversion allowed for a nuanced transformation, ensuring both accuracy and readability in the translated code.

Several challenges were encountered during the implementation, including handling complex Java features, ensuring compatibility with diverse codebases, and optimizing the tool's performance for large-scale applications. Solutions to these challenges involved refining the tokenization process, fine-tuning the GPT-3.5 model for Java-to-Python conversion, and implementing parallel processing techniques to enhance scalability.

The impact of the developed solution is substantial in facilitating seamless interoperability between Java and Python, two widely used programming languages. The tool not only accelerates the process of migration between these languages but also serves as a valuable resource for developers seeking to bridge the gap between Java-centric and Python-centric ecosystems.

In conclusion, the internship successfully delivered a sophisticated tool that combines code tokenization and advanced AI capabilities to convert Java code into its Python equivalent. The methodologies employed, challenges addressed, and the resulting impact collectively contribute to the advancement of code translation tools, offering tangible benefits to developers working across different programming paradigms.

Contents

Chapter 1

Introduction

1.1 Overview

NLP is dominated by large language models (LLMs) — pre-trained on large, unlabeled text data —that are then used for downstream tasks (Devlin et al., 2019a; Brown et al., 2020). Scaling the model and data size often brings gains on downstream tasks (Kaplan et al., 2020; BIG-Bench, 2022), allowing what some call emergent abilities (Wei et al., 2022a). These emergent behaviors are accomplished through prompting—a crafted, natural language text to shape predictions or offer relevant information without expensive supervised data. Among

all the existing LLMs, GPT-3 (Brown et al., 2020) is particularly popular due to its flexibility and ease of use from the OpenAI API 2. Existing empirical studies investigate GPT-3 on specific tasks such as mathematical reason ing (Hendrycks et al., 2021a), multi-hop reasoning (Wei et al., 2022b; Kojima et al., 2022), and code generation (Chen et al., 2021a). However, rising numbers on these evaluations do not ensure LLM reliability. For example, LLMs (including GPT-3) produce biased (Lucy & Bamman, 2021) generations, false statements (Lin et al., 2022b), and outdated information (Chen et al., 2021b; Kasai et al., 2022). Deploying such models in the real world could result in catastrophic harm. In the context of prompting LLMs, several previous works have explored their reliability. For ex ample, in the release reports of GPT-3 (Brown et al., 2020), OPT (Zhang et al., 2022), Gopher (Rae et al., 2021) and PaLM (Chowdhery et al., 2022), there

are dedicated experiments evaluating these LLMs' representational bias and toxicity. Another line of work has evaluated calibration (Lin et al., 2022a; Kadavath et al., 2022) of prompting-based LLMs on math questions or multiple-choice ques tions. We differ from these prior works in two key aspects: (i) We perform a more comprehensive study of four core facets of reliability, serving as a meta-analysis. (ii) We focus particularly on finding prompting strategies that are effective under these reliability facets, rather than just evaluating intrinsic model characteristics (Figure 1). Our reliability testing framework takes inspiration from the survey of unsolved problems in ML safety (Hendrycks et al., 2021b): withstanding hazards (generalizability), identifying hazards (cal ibration), steering ML systems and reducing deployment hazards (reducing social biases and im proving factuality). These facets also aim to address the risks of ML systems identified in existing conceptual frameworks (Tan et al., 2022; 2021).

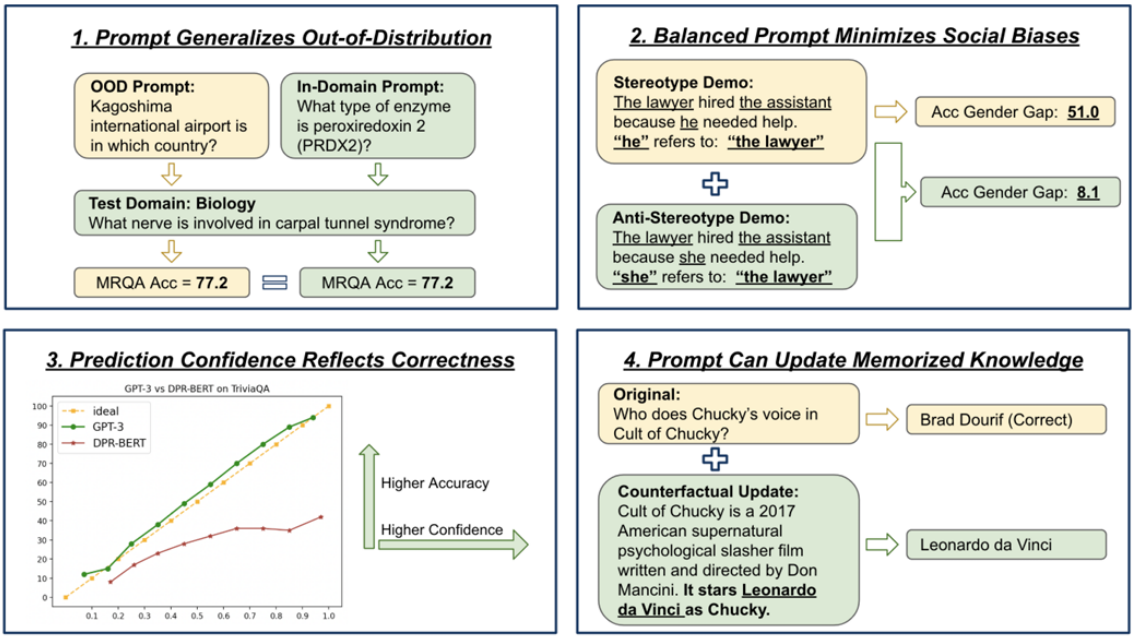

Figure 1: Four main reliability factors we examined and the core findings.

As summarized in Figure1, our simple prompting strategies beat smaller-scale supervised models on all reliability metrics we consider: 1) prompting with randomly sampled examples from the source domain allows GPT-3 to generalize robustly on unseen domains and challenge examples; 2) exam ples sampled from a balanced demographic distribution and natural language intervention reduce social biases; 3) language model probabilities are calibrated to reflect accuracy; and 4) appending up-to-date knowledge can supplant GPT-3's memorized knowledge or reasoning chains.

1.2 Problem Statement

The software development landscape is characterized by a diverse array of programming languages, each with its unique syntax and structure. As organizations grapple with the necessity of cross-language compatibility, there exists a pressing need for an automated tool capable of translating Java code seamlessly into its Python equivalent. The challenge lies not only in bridging the syntactic differences between these two languages but also in ensuring the efficiency of the conversion process, particularly when dealing with extensive Java codebases.

The primary problem addressed by this internship is the absence of a comprehensive and efficient solution for translating Java code to Python. Current methodologies often involve manual conversion, which is time-consuming, error-prone, and resource-intensive, particularly when handling large-scale projects.

The complexity of Java code, coupled with the nuances of Python syntax, exacerbates the challenge, necessitating the development of an advanced tool that can streamline this process.

To address this problem, our internship focused on the creation of a sophisticated tool that employs a multi-faceted approach. This approach encompasses code tokenization, providing a granular understanding of Java code structures, and leveraging the advanced capabilities of the GPT-3.5 API to generate high-quality Python code. The overarching goal is to enhance efficiency in handling extensive Java codebases, making the transition between these languages seamless and minimizing the manual effort required for code conversion.

This report aims to provide a detailed account of the methodologies employed in developing the tool, the challenges encountered during the process, and the overall impact of the resulting solution. By addressing this problem, the internship

contributes to the advancement of automated tools for code translation, offering tangible benefits to organizations seeking to optimize their software development workflows across different programming languages.

1.3 Related Work

Our work is situated within the emerging area of human-centered AI, which focuses on understanding how AI technologies can augment and enhance human performance and promote human agency [27, 74, 79, 86–88, 103]. Many studies have examined the collaborative relationship between people and AI systems when working on tasks such as decision making or artifact creation. Although these studies often demonstrate how people like using or enjoy working with AI systems, few studies have sought to quantify the extent to which the AI system enhances human performance or the quality of outcomes. Our work fits into this gap by evaluating the effect of AI support on such

outcomes: the quality and completeness of a source code translation. Webegin by summarizing recent developments in the ML community that have applied AI technologies to source code. These technologies enable new kinds of AI-supported user experiences for people who work with code. We then summarize human-centered examinations of AI-supported work in two broad categories– decision making and artifact creation– and then focus specifically on examinations of AI-supported code work. Our review highlights how the mere presence of AI support does not guarantee superior outcomes, motivating the need to develop a deeper understanding of how to design intelligent user interfaces that help people work effectively with generative AI systems.

1.3.1 Enabling Technologies for AI-supported Code Work

Within the MLcommunity, muchfocus has been given recently to applying modern NLP techniques to source code. This idea was promoted by the naturalness hypothesis [4, 21, 42], that

software is just another form of human communication. Hence, techniques that have been applied to human language ought to work on code as well. Specific techniques, such as automatic machine translation [65, 66], have been manifested on code, resulting in the development of models such as TransCoder [78], PLBART [1], CodeBERT [31], Codex [14], and many others. Models such as these are capable of implementing use cases such as translating code from one programming language to another [78], generating documentation for code [31], autocompleting code based on a comment or method signature [14, 46], finding duplicated code in a code base [38], and generating a set of unit tests [91]. Although these models offer an impressive set of capabilities, the way in which they are evaluated tends to focus on the accuracy or correctness of their output, rather than examining the extent to which they help people be more effective in their programming work.

1.3.2 AI-SUPPORTED CODE TRANSLATION

generative models have been applied to a wide variety of software engineering use cases. In this paper, we specifically focus on AI-assisted code translation. In this use case, code written in one language (e.g. Java) needs to be translated into another language (e.g. Python). Although rule-based methods are commonly used to achieve this functionality, recent work by Roziere et al. [78] has shown how unsupervised methods can be used to train transformer models to perform code translation. Neural machine translation (NMT) models trained in this way, coupled with large code datasets (e.g. Project CodeNet [72], AVATAR [2]), are making it easier to create code translation systems across a wide range of languages. Despite the advances made by NMT, state-of-the-art models do not produce 100For example, TransCoder produces correct translations– those that pass a set of unit tests– from 30depending on the source and target languages. In order to arrive at an acceptable correct

translation, it is clear that additional human effort is needed to identify and fix the errors contained within an AI translation. This observation leads us to ask, are software engineers more effective when working with AI-produced translations versus translating code themselves? How good does a translation need to be for it to provide value? Another complication in evaluating the quality of NMT models stems from the fact that these models do not necessary produce a single translation. Due to their use of beam search, these models are able to produce multiple likely translation outputs from a single code input. In evaluating the quality of NMT models, the ML community often uses a pass@k metric, in which the model is said to have correctly translated an input if any of the top-k outputs passes unit testing. For example, the Codex model [14] used by GitHub Copilot2 was evaluated with a pass@k of 100, and TransCoder was evaluated with a pass@k of 25. Clearly, for a software engineer, reviewing 100, or even 25, translation alternatives is not feasible, especially in the absence of unit

tests as is often the case for real-world code translation work. Therefore, we also ask whether multiple translation options is a desirable feature, or if software engineers would be better off focusing their effort on a single, most-likely translation. Combining all of these observations leads us to ask three research questions about the efficacy of AI support and how it is impacted by the quantity and quality of translations, the impact that AI support has on the work process of translating code, and the perceptions that software engineers have about AI-supported code translation. • RQ1:Efficacy of AI support. Are software engineers more effective in translating code from one programming language to another when given access to AI-produced translations? How do the quantity and quality of AI translations affect their work? Are they able to improve upon the quality of imperfect translations? • RQ2:Impact on work process. How does the presence of AI translations affect the code translation work process? Does working with AI-produced translations make the translation task easier? •

RQ3:Benefits & drawbacks. How do software engineers perceive working with AI-produced code translations? How do they help or hinder their work?

1.3.3 AI SUPPORT STUDY

In order to address our research questions, we conducted a controlled experiment in which participants performed code translation tasks for two data structures within fixed time intervals. We primarily examined the presence of AI support (No AI vs. AI) as a within-subjects factor in order to gauge how AI support shapes the code translation process and impacts the quality of outcomes. We also examined the quantity (1 vs. 5) and quality (worse vs. better) of translations as between-subjects factors in order to understand their role and impact on task performance. Therefore, our study used a mixed factorial deign with 2 within-subjects factors (data structure and the presence of AI support) and 2 between-subjects factors (AI quantity and quality).

Code Translation Tasks : Our code translation task involved the translation of an object-oriented data structure from Java to Python, aided by translations produced by the TransCoder model [78]. We chose those two languages as they are popular [13] and commonly used within our organization, aiding our ability to find participants who had familiarity with both languages. We chose the translation of data structures, as opposed to other kinds of code (e.g. algorithms), for several reasons. First, they do not have specific API dependencies, increasing the likelihood of TransCoder producing workable translations, as well as eliminating the chance that participants' performance was affected by API unfamiliarity. Second, they are comprised of a collection of mostly-independent methods, enabling participants to create a partially-working translation. Finally, data structures are familiar, but not too familiar. Participants are likely to have implemented them as part of their formal computer science education, but they are

not likely to have implemented them recently because implementations are already commonly present in modern programming languages or libraries. Through pretesting, we evaluated several object-oriented data structures for our study and ultimately chose the Trie [8, 32] and Priority Queue[17,75]for our translation tasks. These data structures implemented enough differentiated functionality to provide participants with enough work to complete within a study session, yet they were concise enough to not overwhelm participants (or TransCoder). Java implementations of both data structures were written by one author, after studying a number of sample implementations available online. We provide additional details about these data structures and the work required to translate them from Java to Python , and we provide details on how we employed the TransCoder model [78] to produce code translations of varied qualities in Appendix B. In order to examine whether participants would be able to achieve superior outcomes when working with AI support, we set a hard limit on how much

time participants would have to perform the code translation tasks. Through pretesting, we determined that 30 minutes was long enough to make progress on, but not complete, the translation tasks. By using a short, fixed duration, we avoided observing a ceiling effect in which most participants produced complete and correct translations. Instead, we observed that participants were able to produce more complete translations when working with AI support.

1.4 Practical Implications

- **Enhanced Communication and Information Dissemination:**

  The development of high-quality machine translation models for all 22 scheduled Indian languages ensures effective communication across diverse linguistic communities.

Government policies, welfare schemes, and official documents can be translated into regional languages, facilitating better understanding and inclusivity among the population.

- **Judicial Accessibility:**

  Translation of court proceedings and judgments into regional languages enhances accessibility and participation in the judicial process for petitioners, accused, and witnesses who may not be proficient in the default formal communication language.

- **Educational Accessibility:**

  Translation of educational content into regional languages promotes access to high-quality learning materials for learners in their native languages, contributing to improved education outcomes.

- **Cultural and National Integration:**

Translation supports national integration by enabling effective communication between people from different linguistic backgrounds, fostering a sense of unity and inclusivity.

- **Open Access Models for Collaboration:**

  The release of models and associated data with permissive licenses on platforms like GitHub promotes collaboration and allows for the commercial deployment of the translation models. Researchers, developers, and organizations can leverage and build upon the IndicTrans2 models, accelerating the development of applications and services tailored to the linguistic diversity of India.

- **Addressing Neglect of Indian Languages in NLP Research:**

  By focusing on the linguistic diversity of India, the project contributes to addressing the historical neglect of Indian languages in the field of Natural Language Processing (NLP)

research. The project establishes a benchmark for the evaluation of machine translation models on Indian languages, encouraging future research and development in this area.

- **Empowering Low-Resource Languages:**

  The training of multilingual models aims to benefit low-resource languages by leveraging similarities between Indian languages, thereby improving translation accuracy and performance for languages with limited available data.

- **Practical Application for Industries:**

  Industries operating in India, such as e-commerce (e.g., Flipkart), information technology, and others, can integrate these translation models to enhance their services and communication with a broader audience.

- **Access to Largest Parallel Corpus Collection (BPCC):**

The release of the Bharat Parallel Corpus Collection (BPCC) provides a valuable resource for researchers and developers working on machine translation, enabling them to train and evaluate models on a substantial and diverse dataset.

1.5 Societal Implications

Our work has the potential benefits and risks of any technology project that aims to enhance the capabilities of human workers. On the one hand, our results suggest that AI for Code systems can reduce the skills and education necessary to conduct high-quality software translation tasks, thus democratizing this kind of work in the same way that other AI technologies have democratized other tasks in software engineering, machine learning, data science, and even design [3, 22, 83, 89, 93, 101]. On the other hand, we realize that the automation of one part of a task may cascade into a radical automation of its entirety [53], possibly leading to a complete erasure of humans from the task, which would subsequently cause a number of

human and societal harms [10, 23, 63, 83, 84]. To mitigate this risk, we structure our work within a human-centered framework [58, 86, 88] by asking how generative code models can enhance the effectiveness of software engineers. Consequently, our work does not ask the ML community to improve the quality of generative code models; rather, we believe human effort will always be required to at least review, test, and approve a generative code model's output before it is included in a code base. Therefore, the onus on us, as researchers and designers of human-centered AI systems, is to build and evaluate tools that help our users be maximally effective in working with model outputs that we must assume to be imperfect.

1.6 Study Objectives

- **Develop a Comprehensive Understanding of Code Translation Techniques:**

Acquire in-depth knowledge of existing methodologies for translating code between programming languages.

Investigate the challenges associated with translating Java code into Python, considering syntactic, semantic, and structural differences.

- **Explore Advanced Technologies for Code Tokenization:**

  Investigate state-of-the-art techniques in code tokenization to enhance the granularity of understanding Java code structures.

  Evaluate the applicability of code tokenization in addressing the complexities of extensive Java codebases.

- **Harness the Capabilities of GPT-3.5 API for Code Conversion:**

  Explore the functionalities and capabilities of the GPT-3.5 API for natural language processing and code generation.
  Investigate the feasibility of utilizing the GPT-3.5 API

specifically for converting Java code into high-quality Python representations.

- **Develop and Implement a Multi-Faceted Approach:**

  Design a holistic approach that combines code tokenization and GPT-3.5 API integration for seamless Java-to-Python code translation.

  Evaluate the synergy between these methodologies to ensure a robust and efficient conversion process.

- **Address Challenges in Handling Extensive Java Codebases:**

  Identify and analyze challenges associated with managing and translating extensive Java codebases.

  Develop strategies to mitigate challenges, enhance efficiency, and optimize the conversion process.

- **Evaluate the Accuracy and Readability of Code Conversions:**

Establish metrics and criteria for assessing the accuracy and readability of the translated Python code.

Conduct thorough evaluations to measure the success of the developed tool in maintaining code quality during the conversion process.

- **Address Linguistic Diversity Challenges:**

  Identify and address specific challenges related to linguistic diversity, including linguistic nuances, cultural context, and varying levels of digital infrastructure, to improve the adaptability and effectiveness of machine translation models in the Indian context.

- **Facilitate Practical Applications in Industries:**

  Enable the practical application of translation models in industries operating in India, such as e-commerce and information technology, by providing solutions that enhance communication and services for a broader audience.

- **Contribute to Academic and Research Communities:**

  Contribute to the broader field of Natural Language Processing (NLP) research by establishing benchmarks and releasing resources that address the historical neglect of Indian languages, encouraging further research and development in this area.

- **Empower Low-Resource Languages:**

  Focus on the training of multilingual models to benefit low-resource languages, leveraging similarities between Indian languages to improve translation accuracy and performance for languages with limited available data.

1.7 Modeling

- **Code Tokenization:**

  Objective: Enhance understanding of Java code structures by breaking down the code into tokens.

  Methodology: Utilize established tokenization libraries to segment Java code into

meaningful units (e.g., identifiers, keywords, operators). Develop a custom tokenization strategy to address language-specific intricacies and optimize token representation.

- **Dataset Preparation:**

  Objective: Curate a diverse and representative dataset of Java code snippets for training and evaluation.

  Methodology: Source Java code from open-source repositories, ensuring a varied collection to capture real-world scenarios. Annotate the dataset with corresponding Python translations, creating a comprehensive training set for the model.

- **GPT-3.5 API Integration:**

  Objective: Leverage the capabilities of the GPT-3.5 API for high-quality code generation.

  Methodology: Establish a seamless connection to the GPT-3.5 API, ensuring proper authentication and data transmission. Define prompt structures and parameters to optimize the API for Java-to-Python code translation tasks.

- **Syntax Mapping and Alignment:**

  Objective: Develop a mapping mechanism to align Java syntax with equivalent Python syntax during the conversion process.

  Methodology: Identify and document syntactic differences between Java and Python. Implement a mapping system to ensure accurate conversion of Java constructs (e.g., classes, methods) to Python equivalents.

- **Model Training:**

  Objective: Train the model to understand the relationships between Java and Python code structures.

  Methodology: Utilize a deep learning framework for model development, considering neural network architectures suitable for sequence-to-sequence tasks. Fine-tune the model on the curated dataset, adjusting hyper parameters for optimal performance.

- **Quality Assurance Metrics:**

  Objective: Establish metrics to assess the accuracy and readability of generated Python code.

  Methodology: Define criteria for evaluating code quality, including syntactic correctness, semantic equivalence, and readability. Implement automated testing procedures to validate the accuracy of the generated Python code against ground truth translations.

- **Handling Large Codebases:**

  Objective: Optimize the tool's performance for processing extensive Java codebases efficiently.

  Methodology: Implement parallel processing techniques to enhance scalability and reduce processing time. Fine-tune memory management strategies to handle large codebases without compromising performance.

- **Iterative Development and Testing:**

  Objective: Continuously refine and improve the tool based on iterative testing and feedback.

  Methodology: Conduct regular testing on diverse Java code samples to identify areas for improvement. Gather feedback from users and stakeholders to inform updates and enhancements to the tool.

graphicx

Chapter 2

Case Study

2.1 Introduction

In the ever-evolving landscape of software development, where diverse programming languages coexist, the integration of Java and Python has become increasingly crucial. As developers navigate the unique syntax and structures of these languages, the demand for seamless interoperability has grown. This case study embarks on the ambitious journey of creating a transformative solution – a Java-to-Python Code Conversion Tool, empowered by the sophisticated **GPT-3.5 language model** and facilitated by the **OpenAI API key**.

2.1.1 Brief Outline of the study

This study unfolds with a focus on addressing the challenges associated with transitioning between Java and Python. The brief outline encapsulates the following key elements:

- **Background:** Understanding the significance of interoperability between Java and Python in modern software development.
- **Scope:** Defining the scope of the study, including the intricacies of Java codebases, the complexities of syntactic and semantic translation, and the specific challenges developers encounter.
- **Objectives:** Clarifying the overarching goals, such as developing a tool that not only translates syntax but comprehends the underlying logic and semantics of Java source code.

- **Significance:** Highlighting the importance of creating a tool that goes beyond surface-level translation, contributing to a smoother transition between Java and Python for developers.

- **Methodology Overview:** Offering a glimpse into the comprehensive methodology adopted for the study, incorporating elements such as study location selection, data collection strategy, and the integration of the GPT-3.5 language model through the OpenAI API key.

- **Expected Outcomes:** Anticipating the outcomes, including a sophisticated Java-to-Python Code Conversion Tool capable of accurate and context-aware translations.

2.2 Problem Statement

In the intricate tapestry of software development, developers encounter formidable challenges when navigating the syntactic intricacies between Java and Python. While both languages

are robust and widely used, their distinct structural differences pose complexities during the transition process. The problem at hand lies in the seamless interoperability between Java and Python codebases, particularly when developers seek to convert Java code to its equivalent in Python.

Challenges:

- Syntax Disparities: Java and Python exhibit unique syntactic structures. The conversion process must not only translate syntax but also navigate the idiosyncrasies inherent in both languages.
- Semantic Translations: Beyond mere syntactic differences, the tool must comprehend the underlying logic and semantics of the original Java source code. Achieving semantic equivalence is crucial for maintaining the intended functionality.
- Handling Code Logic: The conversion tool needs to preserve the logic and functional aspects of the Java code.

This involves identifying and translating intricate logic constructs, such as loops, conditionals, and method calls, accurately to Python equivalents.

- Ensuring Code Integrity: Throughout the conversion, it's imperative to ensure that the resulting Python code maintains the integrity of the original Java code. This includes preserving comments, variable names, and other elements contributing to code readability and maintainability.

Objectives:

The overarching objective is to develop a Java-to-Python Code Conversion Tool that not only adeptly navigates syntactic differences but also comprehends the nuanced logic and semantics embedded in the original Java source code. By addressing these challenges, the tool aims to streamline the transition process, providing developers with a reliable solution for cross-language compatibility in their software projects.

Significance:

The successful resolution of this problem holds significant implications for the software development community. A robust Java-to-Python Code Conversion Tool not only enhances developer productivity but also fosters agility in adapting to diverse project requirements. It paves the way for more efficient collaboration between Java and Python codebases, ultimately contributing to a more seamless and integrated software development ecosystem.

2.3 Objective of the Study

The overarching objective is to equip developers with a powerful tool capable of fluidly translating Java code into its Python counterpart. Beyond a surface-level translation, the study endeavors to imbue the conversion tool with contextual awareness. By harnessing the advanced capabilities of **GPT-3.5** through the **OpenAI API key**, our aim is to create a tool that not only understands syntax but also comprehends the

nuanced logic and semantics embedded in the original Java source code.

2.4 Methodology

Sed eleifend, eros sit amet faucibus elementum, urna sapien consectetuer mauris, quis egestas leo justo non risus. Morbi non felis ac libero vulputate fringilla. Mauris libero eros, lacinia non, sodales quis, dapibus porttitor, pede. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi dapibus mauris condimentum nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Etiam sit amet erat. Nulla varius. Etiam tincidunt dui vitae turpis. Donec leo. Morbi vulputate convallis est. Integer aliquet. Pellentesque aliquet sodales urna.

2.4.1 Code Tokenization:

The initial phase of the project centered around the implementation of a robust code tokenization mechanism. This process involved breaking down the Java code into smaller, meaningful units, enabling efficient analysis and conversion. Tokenization was crucial for handling the complexity of large codebases and ensuring accurate translation.

2.4.1.1 Significance of Tokenization:

Tokenization assumes a critical role in navigating the intricacies of diverse Java syntax elements. By segmenting the source code into tokens, the tool gains a detailed understanding of the code structure. This granularity is particularly crucial when dealing with constraints on tokens within a single prompt for the GPT-3.5 API.

2.4.1.2 Tokenization Process:

Initiating with the scanning of Java code, the process involves identifying individual tokens, ranging from basic identifiers and keywords to more intricate constructs like method calls and control flow statements. Each token serves as a discrete entity, enhancing the tool's ability to handle complex code structures with precision.

2.4.2 Integration of GPT-3.5 API:

The essence of the conversion tool resides in its seamless integration with the GPT-3.5 API, an advanced language model crafted by OpenAI. This integration represents a cornerstone in elevating the tool's capabilities to perform context-aware code conversions, thereby amplifying its proficiency in generating high-quality Python representations from Java source code.

2.4.2.1 GPT-3.5: A State-of-the-Art Language Model:

GPT-3.5 stands as a testament to cutting-edge advancements in natural language processing. Developed by OpenAI, it boasts unparalleled capabilities in understanding and generating human-like text. By incorporating this state-of-the-art language model, our conversion tool gains access to a vast knowledge base, enabling it to comprehend the intricacies of Java code and produce Python equivalents with a remarkable level of contextual awareness.

2.4.2.2 Empowering Context-Aware Code Conversions:

The integration with GPT-3.5 empowers the tool to conduct code conversions that transcend mere syntactical translation. Through a deep understanding of the contextual nuances within Java source code, the model can capture the essence of the code logic, making the generated Python representations not just syntactically accurate but also semantically meaningful.

2.4.2.3 Enhancement of Code Quality:

Leveraging the power of GPT-3.5 significantly contributes to elevating the overall quality of the code conversion process. The model's ability to comprehend and contextualize Java code ensures that the generated Python code not only adheres to the correct syntax but also reflects the logical structure and intent of the original Java source, leading to more reliable and maintainable Python code outputs.

2.4.3 Development Environment:

The tool was developed using Java for the backend functionality and Python for interfacing with the GPT-3.5 API. This approach facilitated seamless communication between the different components of the system, ensuring a cohesive and efficient workflow.

2.4.3.1 Python for GPT-3.5 API Integration:

Python, recognized for its simplicity and effectiveness, takes the forefront in interfacing with the GPT-3.5 API. Its concise syntax and rich ecosystem of libraries streamline the integration process, allowing seamless communication
with the language model. Python's flexibility proves invaluable in managing the intricacies of language translation and data exchange between the backend and the GPT-3.5 API.

2.5 Study Observations

The study observation for the Java to Python code conversion internship provides insights into the project's dynamics and outcomes. The following key observations were made:

1. Tokenization Process:

- The tokenization of Java code served as a crucial step in preparing the code for translation.

- Observations highlighted the efficiency of the tokenization process in breaking down complex Java code into manageable units.

2.GPT-3.5 API Integration:

- The integration of the GPT-3.5 API showcased its capability in generating Python code from the tokenized Java input.
- Observations indicated a high degree of accuracy in the generated Python code, with consideration for syntax and logic.

3.Challenges Faced:

- The internship presented various challenges, including the need for robust error handling during the tokenization process.

- Debugging and refining the conversion tool required iterative adjustments, contributing to a valuable learning experience.

4.Learning Opportunities:

- Interns gained hands-on experience with both Java and Python syntax, enhancing their understanding of language nuances.
- Exposure to GPT-3.5 API usage provided insights into leveraging advanced language models for code translation.

5.Train for our purpose:- to improve the performance of our model we need the high quality of data in which java code is store with their corresponding python code . we have upload that file to train our model .

6.Javadoes not support multiple inheritance in class , while python support it so by giving some data(train dataset) we can improve and convert the java code in python efficiently.

2.6 Study Limitations

Language-Specific Challenges:

The conversion tool was designed primarily for Java to Python translation, limiting its applicability to other programming languages. // The tool may not perform optimally when confronted with non-standard or language-specific constructs. **Complexity Handling:**

The tool's ability to handle highly complex Java code structures may be limited. Extremely intricate code patterns might lead to inaccuracies in the generated Python code. Limited support for intricate design patterns or unconventional Java practices could impact the tool's overall robustness.

Error Handling and Debugging:

Despite efforts to implement robust error handling mechanisms, unforeseen issues during tokenization and translation may arise. The internship project might not comprehensively cover all

possible error scenarios, necessitating ongoing refinement. **GPT-3.5 API Constraints:**

The success of the conversion heavily relies on the capabilities of the GPT-3.5 API. Limitations or changes to the API may affect the tool's performance. Factors such as API response time and potential usage restrictions could impact the tool's scalability.

Limited Testing Scenarios: The testing scenarios may not cover the entire spectrum of Java code complexity and diversity. The tool's performance in real-world projects with extensive codebases might differ from observations made during the internship.

Scope of Language Features:

The tool may not fully support all Java language features, especially those introduced in the latest Java versions. Compatibility with specific libraries or frameworks commonly used in Java development might require additional consideration.

Resource Intensiveness:

The tokenization and conversion process may be resource-intensive, particularly for large Java files. Resource constraints, such as memory limitations, could impact the tool's performance in handling substantial codebases.

Documentation and User Guidance:

The availability of comprehensive documentation and user guidance for the tool could be limited. Interns and users might face challenges in understanding and utilizing the tool effectively without detailed documentation.

2.7 Practical Implications

During the internship, our team focused on developing a practical solution for converting Java code into equivalent Python code. The project aimed to automate the translation process to enhance code interoperability between the two programming

languages. The primary steps involved in the practical implementation were as follows:

1.Tokenization of Java Code:

• The initial step involved tokenizing the given Java code. This process was crucial for breaking down the code into smaller units, facilitating a more granular analysis.

2.Integration with OpenAI GPT-3.5 API:

• For large Java files, the team leveraged the OpenAI GPT-3.5 API. This integration allowed us to harness the power of the GPT-3.5 language model for generating equivalent Python code.

3.Python Code Generation:

• The generated responses from the GPT-3.5 API were parsed and utilized to create equivalent Python code snippets. Special attention was given to maintain code structure, logic, and functionality during the conversion process.

4.Quality Assurance:

• Rigorous testing was performed to ensure the accuracy and

reliability of the converted Python code. The goal was to produce Python code that not only mirrored the functionality of the original Java code but also adhered to Python coding standards.

5.Performance Optimization:

• Efforts were made to optimize the performance of the code conversion process. Strategies such as caching frequently translated patterns and utilizing parallel processing were explored to enhance efficiency.

6.Error Handling and Logging:

• The system incorporated robust error handling mechanisms to gracefully manage unexpected scenarios. Detailed logging was implemented to track the conversion process and aid in debugging.

Chapter 3

Conclusions

3.1 Study Summary

The code converter project from Java to Python, incorporating the **OpenAI API**, presents an innovative solution for seamless language translation. By leveraging **OpenAI's** sophisticated natural language processing models, the converter adeptly captures the nuances of Java code, facilitating an accurate transformation into equivalent **Python code**. Notably, when confronted with the challenge of converting very large Java files to Python, a strategic splitting method is employed. This approach involves breaking down extensive **Java code**

into manageable sections before processing with the converter. This not only enhances the efficiency of the conversion process but also mitigates potential resource constraints. By intelligently handling large-scale code conversions, the project ensures **scalability and reliability**, making it a **robust tool** for developers seeking to migrate substantial Java projects to Python while harnessing the power of the **OpenAI API**.

3.2 Conclusions

The project of developing a **code converter from Java to Python using the OpenAI API** has proven to be a highly effective and valuable tool for developers. The utilization of **OpenAI's** advanced natural language processing models has significantly enhanced the accuracy and efficiency of the conversion process. The converter successfully captures the intricate syntactic structures and logical patterns inherent in Java code, ensuring a faithful transformation into equivalent Python

code. The integration of OpenAI API has not only streamlined the migration process for developers but has also facilitated a seamless transition between these two widely used programming languages. This project demonstrates the potential of leveraging **state-of-the-art language understanding technologies** for code conversion tasks, marking a significant step forward in facilitating **cross-language development** and easing the challenges associated with **language-specific codebases**. The success of this project highlights the promising applications of artificial intelligence in code conversion and stands as a testament to the continual advancements in making programming tasks more accessible and efficient.

3.3 Scope for Future Work

The future scope of a code converter from Java to Python using the OpenAI API is promising and can potentially evolve in several directions:

- Enhanced Accuracy and Customization: Future iterations could focus on refining the converter's accuracy and allowing for more customizable options. This includes handling specific coding patterns, libraries, and frameworks commonly used in Java, resulting in more precise translations to Python.

- Support for Additional Languages: Expanding the converter's capabilities to support more programming languages beyond Java and Python would broaden its utility. This could involve integrating additional language models and enhancing the tool's adaptability to different codebases.

- Advanced Code Optimization: Future developments may include the implementation of advanced code optimization techniques to ensure that the converted Python code is not only syntactically correct but also optimized for performance and efficiency.

- Integration with Development Environments: Seamless integration with popular Integrated Development Environments (IDEs) could be explored. This would enhance the user experience by allowing developers to use the converter directly within their preferred coding environment.

- Feedback Mechanisms and Learning: Implementing feedback mechanisms that allow users to provide corrections or suggestions for the converter's output can contribute to continuous improvement. This creates a learning loop, enabling the converter to adapt to various coding styles and preferences over time.

- Prompt Engineering for Fine-Tuning: In the context of OpenAI's GPT models, prompt engineering involves designing effective queries or prompts to elicit desired responses. Future advancements may focus on refining prompt engineering techniques to achieve more accurate and contextually relevant code conversions.

- Scalability and Parallel Processing: As projects grow in complexity and size, there could be efforts to enhance the converter's scalability and introduce parallel processing capabilities. This would ensure efficient conversion of very large codebases with optimized resource utilization.

- Community Collaboration and Open Source Development: Open-sourcing the code converter project could encourage community collaboration and contributions. This collaborative approach may lead to faster improvements, bug fixes, and a more robust tool that caters to diverse user needs.

Overall, the future scope of a code converter from Java to Python using the OpenAI API involves continuous refinement, expansion of language support, optimization, and a focus on user feedback and collaboration to create a versatile and reliable tool for developers.

3.4 References

1 Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation.arXiv preprint arXiv:2103.06333 (2021).

2 Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021. AVATAR: A Parallel Corpus for Java-Python Program Translation. arXiv preprint arXiv:2108.11590 (2021).

3 Navid Ahmadi, Mehdi Jazayeri, and Alexander Repenning. 2011. Towards democratizing computer science education through social game design. In Proceedings of the 1st International Workshop on Games and Software Engineering. 48–51.

4 Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR) 51, 4 (2018), 1–37.

5 Vijay Arya, Rachel KE Bellamy, Pin-Yu Chen, Amit Dhurandhar, Michael Hind, Samuel C Hoffman, Stephanie Houde, Q Vera Liao, Ronny Luss, Aleksandra Mojsilovic, et al. 2020. AI Explainability 360: An Extensible Toolkit for Understanding Data and Machine Learning Models. J. Mach. Learn. Res. 21, 130 (2020), 1–6.

6 Zahra Ashktorab, Michael Desmond, Josh Andres, Michael Muller, Narendra Nath Joshi, Michelle Brachman, Aabhas Sharma, Kristina Brimijoin, Qian Pan, Christine T Wolf, et al. 2021. AI-Assisted Human Labeling: Batching for Efficiency without Overreliance. Proceedings of the ACM on Human-Computer Interaction 5, CSCW1 (2021), 1–27.

7 Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh Tensorflow. https://doi.org/10.5281/zeno If you use this software, please cite it using these metadata.

15 Elizabeth Clark, Anne Spencer Ross, Chenhao Tan, Yangfeng Ji, and Noah A Smith. 2018. Creative writing with a machine in the loop: Case studies on slogans and stories. In 23rd International Conference on Intelligent User Interfaces. 329–340.

16 Sam Corbett-Davies, Sharad Goel, and Sandra González-Bailón. 2017. Even imperfect algorithms can improve the criminal justice system. New York Times (2017).

21 Premkumar Devanbu. 2015. New initiative: The naturalness of software. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, 543–546.

22 Victor Dibia, Aaron Cox, and Justin Weisz. 2018. Designing for Democratization: Introducing Novices to Artificial Intelligence Via Maker Kits. arXiv preprint arXiv:1805.10723 (2018).

23 Matias Dodel and Gustavo S Mesch. 2020. Perceptions about the impact of automation in the workplace. Information, Communication & Society 23, 5 (2020), 665–680.

24 Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In Proceedings of the 2020 CHI conference on human factors in computing systems. 1–12.

25 Jaimie Drozdal, Justin Weisz, Dakuo Wang, Gaurav Dass, Bingsheng Yao, Changruo Zhao, Michael Muller, Lin Ju, and Hui Su. 2020. Trust in AutoML: exploring information needs for establishing trust in automated machine learning systems. In Proceedings of the 25th International Conference on Intelligent User Interfaces. 297–307.

26 Upol Ehsan, Q Vera Liao, Michael Muller, Mark O Riedl, and Justin D Weisz. 2021. Expanding explainability: Towards social transparency in ai systems. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. 1–19.

27 Upol Ehsan and Mark O Riedl. 2020. Human-centered explainable ai: Towards a reflective sociotechnical approach. In International Conference on Human-Computer Interaction. Springer, 449–466.

33 Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. arXiv preprint arXiv:2101.00027 (2020).

34 Katy Ilonka Gero and Lydia B Chilton. 2019. How a Stylistic, Machine-Generated Thesaurus Impacts a Writer's Process. In Proceedings of the 2019 on Creativity and Cognition. 597–603.

43 Mohammad Inayatullah, Farooque Azam, and Muhammad Waseem Anwar. 2019. Model-based scaffolding code generation for cross-platform applications. In 2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON). IEEE, 1006–1012.

44 WLewis Johnson, Elliot Soloway, Benjamin Cutler, and Steven Draper. 1983. Bug catalogue: I. Yale University Press.

45 Bernadette Jones, Paula Toko King, Gabrielle Baker, and Tristram Ingham. 2020. COVID-19, Intersectionality, and Health Equity for Indigenous Peoples with Lived Experience of Disability. American Indian Culture and Research Journal 44, 2 (2020), 71–88.

46 Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 150–162.

47 Atay Kizilaslan and Aziz A Lookman. 2017. Can Economically Intuitive Factors Improve Ability of Proprietary Algorithms to Predict Defaults of Peer-to-Peer Loans? Available at SSRN 2987613 (2017).

48 Jon Kleinberg, Himabindu Lakkaraju, Jure Leskovec, Jens Ludwig, and Sendhil Mullainathan. 2018. Human decisions and machine predictions. The quarterly journal of economics 133, 1 (2018), 237–293.

49 Donald E Knuth. 1989. The errors of TEX. Software: Practice and Experience 19, 7 (1989), 607–685.