PyLinguist: Automated Translation of Python for Hindi Programmers

First Author
Antara Tewary
G01413546
atewary@gmu.edu

Second Author
Ankit Kumar
G01436204
akumar37@gmu.edu

Third Author
Homa Haghighi
G01145449
hhaghigh@gmu.edu

1 Introduction

Python is a popular programming language, which has gained its popularity due to many factors, which include its readability and intuitive English-like syntax that makes the code accessible to English speakers. However, this can be a significant barrier for non-English speakers who have to grapple with both the programming concepts and learning a new language. Our project addresses this issue by developing a comprehensive translation system that converts Python code from English to Hindi, making Python programming more accessible to Hindi speakers while maintaining code functionality and readability.

1.1 Task / Research Question Description

Core research questions are-
- How can we effectively translate Python's syntax, keywords, and natural language elements while preserving code functionality?
- What combination of translation techniques provides optimal results for code translation??
- How can we quantitatively and qualitatively evaluate the effectiveness of code translation?

Our project implements a two-stage translation pipeline combining keyword dictionaries, Google Translate API, and GPT models, with comprehensive evaluation metrics to assess translation quality and code functionality preservation.

1.2 Motivation & Limitations of existing work

While Python's pseudo-code nature simplifies programming for English speakers, it presents significant challenges for non-English speakers who comprise the majority of the global population. Research has shown that students learn programming concepts more effectively when using a coding language based on their native language. Current solutions like CodeInternational partially address this issue by translating comments and identifiers, but they fall short of providing a complete solution that encompasses Python's built-in functions, keywords, and error messages.

Limitations of existing work include:

- Incomplete coverage of Python's language elements
- Lack of consistency in technical term translation
- Limited handling of language-specific challenges
- Insufficient preservation of code functionality
- Absence of comprehensive evaluation metrics

1.3 Proposed Approach

We propose a two-stage pipeline for translating Python code between English and Hindi while preserving functionality. Stage 0 involves preprocessing Python code from the Hugging Face dataset to extract clean samples. In Stage 1, we use a combination of rule-based translation with a keyword dictionary and the Google Translate API, where a KeywordManager maps Python keywords and a CodeTranslator handles compound words, comments, and strings. Stage 2 refines the translations using GPT-4o Mini, which enhances the initial translations through example-based learning. This hybrid approach leverages precise keyword mapping, neural machine translation for natural language, and contextual enhancement via GPT, ensuring reliable and functional translations.

1.4 Likely challenges and mitigations

There are several challenges in trying to translate Python code between English and Hindi. Our implementation tackles these through specific mitigation strategies:-

- **Compound Word Translation**: Code often contains compound words separated by underscores. The `translate_token` method in the `CodeTranslator` class handles this by splitting compound words, translating individual components, and re-joining them with underscores to maintain code readability.
- **Code Structure Preservation**: Maintaining code formatting and structure is crucial for readability and execution. The `translate_line` method preserves indentation and line structure by measuring leading whitespace before translation and restoring it afterward.
- **Translation Reliability**: To handle potential translation failures, we apply two important features in our code. The `safe_translate` method, which includes a retry mechanism for failed translations. Next is the `CheckpointManager` class which maintains translation progress. This allows longer translation tasks to be resumed if they are interrupted.
- **Keyword Translation**: Python keywords and built-in functions need consistent translation. The `KeywordManager` class maintains a dictionary mapping between English and Hindi keywords, which use Joshua Otten's curated dataset (Otten et al., 2023b).
- **Comments and String Handling**: Code comments need a different handling than executable code. The `translate_line` method identifies comments by looking at the `'#'` character and applying separate translation rules for code and comment sections. This preserves the comments.

These challenges are addressed through specific implementation, though future work can definitely improve these solutions.

2 Related Work

- (Otten et al., 2023b)

  This paper introduced the PyLinguist framework to address the challenges of making Python accessible to non-English speakers. Their work demonstrates that Python's pseudo-code nature, while beneficial for English speakers, creates barriers for others who must master both programming concepts and English simultaneously. They propose automatically translating Python's natural language elements (keywords, error messages, identifiers) into other human languages. Their preliminary implementation enables coding in 7 additional languages and provides a roadmap for developing automated translation frameworks. Our work builds upon their vision by implementing a specific English-Hindi translation pipeline that combines rule-based translation with neural methods.

- (Piech and Abu-El-Haija, 2019)

  This paper introduces CodeInternational, a tool designed to translate code comments and identifiers across multiple human languages, helping non-English speakers learn programming more easily. The study emphasizes the importance of making code accessible in native languages but primarily focuses on Java and comments/identifiers. Our project differs by aiming to translate the entire Python syntax, not just comments or identifiers, providing a more comprehensive solution for Python programming education in non-English languages.

- (Devanbu, 2015).

  This paper explores how code, like natural language, follows repetitive patterns, which can be modeled using statistical language models such as n-grams. The work applies these models to tasks like code completion and suggests that programming languages are predictable. While this research focuses on enhancing software development efficiency, our project applies similar language models but aims to address the language accessibility challenge by translating Python syntax for non-English speakers.

- (Ott et al., 2018)

  The authors demonstrate the application of deep learning models to analyze software repositories and improve tasks like bug prediction and code completion. Their deep learning models perform well on large software corpora. Our project builds on these deep learning concepts but uses them to handle the translation of Python's structure and syntax into multiple human languages, a task that extends beyond improving code completion.

- (Tang et al., 2024)

  UniXcoder proposes a model that utilizes code comments and abstract syntax trees (AST) to improve code representation and generation across different programming tasks. This work focuses on enhancing programming tasks like code completion through multi-modal data. In contrast, our project focuses on the translation of Python's syntax into human languages, leveraging multi-modal data like AST for language translation and accessibility rather than improving programming performance.

3 Methodology

3.1 Pipeline Overview

The approach for translating Python code between English and Hindi consists of two main stages: initial translation, and GPT-based enhancement. Each stage builds upon the previous one, ensuring quality translations while preserving code functionality.

3.1.1 Stage 0: Data Preprocessing

The dataset that we use is the python-code-dataset-500k from Hugging Face (jtatman, 2024). We load this dataset using Hugging Face datasets library. We first clean the dataset by removing unnecessary columns, like 'instruction' and 'system', to focus on the code content.

The main part of our preprocessing involves extracting clean Python code from the dataset. We implement this using a regex pattern to extract the code enclosed within these tags. The regex patterns ensure that we capture only the actual python code while removing any surrounding markup or documentation. After extraction, we perform a cleaning step using `dropna()` to remove any entires that might have been corrupted or imporperly formatted while we did the extraction process.

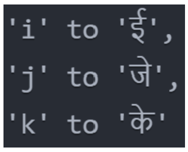


Figure 1: Translation Mapping Overview

The preprocessed dataset is stored in a structured CSV format with an `'English_code'` column. The final processed dataset contains 67,063 total entries, of which 25,549 are unique code samples.

To manage the processed data, we create a directory system. We have a `data` directory, where we maintain separate files for different evaluation configurations, specifically creating distinct files for experiments with 5, 10, 20, and 30 examples (translations_gpt_10r_5e.csv through translations_gpt_10r_30e.csv).

3.1.2 Stage 1: Initial Translation

This stage implements several interconnected steps that work together to perform basic English to Hindi code translation.

They `KeywordManager` class hangles the python-specific translations. It reads from a predefined `Joshua_Keywords.csv`, which contains 234 English-Hindi keyword pairs curated by Joshua Otten (Otten et al., 2023a). This csv file contains other languages too, but since we focus on English to Hindi translation, we remove all non-Hindi translation columns and create a dictionary that maps English keywords to their Hindi equivalents. This class also implements special case handling for single-letter variable names commonly used in Python programming, specifically mapping - `CodeTranslator` class handles the translation of words by first splitting them at underscore (_). translating each component individually, then rejoining them with (_). It also takes care of space that might be introduced during translation.

`safe_translate` method implements a retry mechanism that attempts each transaction upto three times. If a transaction fails after al retries, it returns the original text rather than failing completely, ensuring that the translation process does not stop abruptly.

`translate_line` method handles the code structure by preserving and measuring the indentation of each line. It also separates code and comments, denoted by `#`, and applies different translation rules to each section.

To manage the translation process, we use the `CheckpointManager` class, which maintains the translation progress. This allows us to resume longer translation tasks if they are interrupted.

3.1.3 Stage 2: GPT-based Enhancement

The second stage uses GPT-4o mini model (OpenAI, 2024) to refine and improve the output from Stage 1. `GPTTranslator` class uses previously translated examples which serve as a reference material for the model. We use 5,10,20 0r 30 examples to study the impact of example quantity on the translation quality.

The `create_prompt` creates carefully formatted prompts for the GPT model (Appendix 6). Each prompt contains three key sections: example translations showing the desired transformation pattern, explicit instructions for maintaining code structure and handling translations, and the partially translated code that needs enhancement. This structured approach helps guide the model toward producing consistent and reliable translations.

The translation process is done by `translate_code` method, which applied they keyword replacement strategy from Stage 1, then creates an appropriate prompt for GPT 4o Mini. The model runs with tempaerature 0 to maximize the consistency in output.

Our test case for the translation process is 10 rows of English code. When we ask GPT model to translate these 10 rows of code, we give to it 5,10,20 and 30 examples of code translation from Stage 1 as a reference in the prompt. After adding refined prompts, we run the model to get the translated code. Keeping the test case of 10 rows of code as constant, we are able to generate metrics for different example counts.

3.2 Evaluation Framework

3.2.1 Dataset and Test Configuration

- **Total Dataset**: 500k Python code samples from hugging face (jtatman, 2024)
- **Test Configuration**:
  - Translation set: 10 samples selected for translation
  - Example sets: Varying sizes (5,10,20,30) that are given to GPT model for reference
  - Human evaluation set: 20 samples evaluated by human evaluators
- **Keyword Dictionary**: 234 pre-mapped English-Hindi keyword pairs translated by Joshua Otten.

3.2.2 Human Evaluation Framework

Two bilingual evaluators (Hindi-English) with 4 years of experience in Python programming evaluated 20 code samples generated by our model. They followed the following rating scale-

Rating Scale(1-5):

- **1**: Unusable and incorrect translation
- **2**: Partially correct translation, major revisions needed
- **3**: Mostly correct translation, minor revisions needed
- **4**: Good translations with minimal revisions needed
- **5**: Perfect translation, no revisions needed

Evaluation Criteria:

- **Syntax Correctness (SC)**: Proper keyword translation, code structure preservation and indentation accuracy
- **Semantic Preservation (SP)**: Logic preservation, variable scope maintenance and function behavior consistency
- **Hindi Language Quality (HLQ)**: Evaluating natural hindi expressions, the technical term consistency, comment clarity and code readability

3.2.3 Technical Evaluation Framework

Our technical evaluation framework consists of the following key components:

Syntax Validation

- **AST(Abstract Syntax Tree) Validation**: Tests if translated code produces valid Python AST, providing a binary outcome (Valid/Invalid) and capturing specific syntax errors.

This is crucial as it verifies that our translations maintain Python's syntactic rules, ensuring the code remains executable.

- **Token Structure Analysis**: Analyzes code structure using Python's tokenize module to quantify token distributions (NAME, STRING, NUMBER, NEWLINE). This helps verify that the structural integrity of the code is preserved during translation, particularly important for maintaining code readability and debugging capabilities.

Semantic Testing

- **Component-level Similarity**: Rather than relying on unit tests which would be impractical given the diverse nature of code samples, we evaluate semantic preservation through component analysis:
  - Comment preservation: ensures maintenance of code documentation
  - Function name consistency: verifies logical structure preservation
  - String literal preservation: confirms data integrity

Translation Quality Metrics

- **BLEU Score**: Computes sentence-level BLEU score between original and back-translated code after tokenization. While BLEU scores are traditionally used for natural language translation, they provide a standardized metric for comparing code structure preservation.

- **Back Translation Assessment**: Evaluates round-trip translation quality using a composite score combining BLEU score, syntax validation, and semantic similarity. This approach provides a comprehensive view of translation quality that wouldn't be possible with unit tests alone, as it considers both structural and semantic aspects of the code.

4 Experiments

4.1 Datasets

- **Primary Code Dataset**:
  - Name: python-code-dataset-500k
  - Source: Hugging Face (jtatman, 2024)
  - Access: Public dataset, accessed via Hugging Face datasets library
  - Usage in code: for extracting Python code examples from `output` column - underlinePreprocessing: We extract the python code enclosed in the `<pythoncode>` tags using regex
  - Dataset split: No specific tran/dev/test splits implemented

- **Keyword Dataset**:
  - Name: Joshua_Keyword.csv
  - Content: 234 English-Hindi keyword pairs
  - Usage in code: for direct translation of Python keywords
  - Preprocessing: Curated by Joshua Otten (Otten et al., 2023b)
  - Access: Local file, obtained from (Otten et al., 2023a)

4.2 Implementation

Our implementation builds upon the Universal Python framework proposed by (Otten et al., 2023a). We use the Hugging Face (jtatman, 2024) for training and Google Translate API (Google, 2024) and GPT-4o Mini (OpenAI, 2024) for translation. The keyword mapping dictionary is taken from Otten's work (Otten et al., 2023b) on multi-lingual python translation.

Link to our repository:PyLinguist Code

5 Results

5.1 Discussion

Our evaluation of the Python-to-Hindi translation system yielded consistent performance across different configurations, as shown in Table 1 and visualized in Fig. 3(a), with all metrics maintaining scores above 0.65. The violin plots in Fig. 3(b) reveal the distribution characteristics of our evaluation metrics, showing wider variation in BLEU scores compared to more tightly clustered semantic and overall scores, while maintaining consistent median values across configurations. The semantic similarity analysis, depicted in the heatmap in Fig. 4(a), demonstrates robust performance with mean scores ranging from 0.907 to 0.937, maximum scores consistently reaching 1.0, and low standard deviations between 0.111 and 0.209. Fig. 4 illustrates the syntax validation results, showing near-perfect syntax validity across all configurations, with only the 20-example configuration showing a slight 0.1% invalid syntax rate while all other configurations maintained 100% validity. This comprehensive evaluation, supported by both the tabulated results in 1 and the visual representations across all four

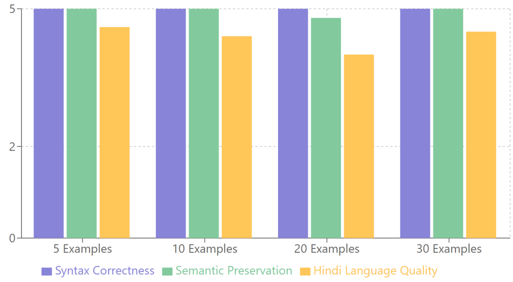


Figure 2: Human Evaluation Results

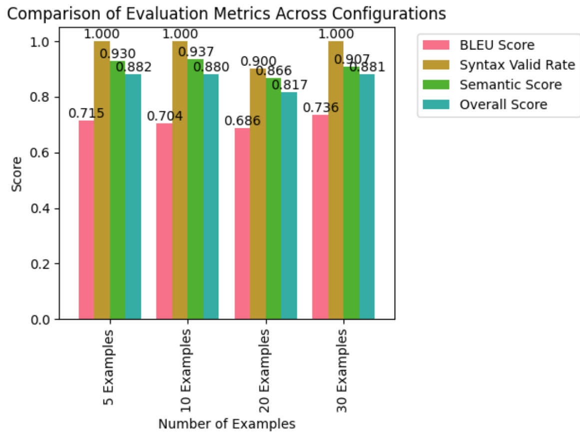
figures, indicates strong and stable performance of our translation system, particularly in maintaining semantic similarity and syntactic correctness across different example counts.

5.1.1 Human Evaluation Results

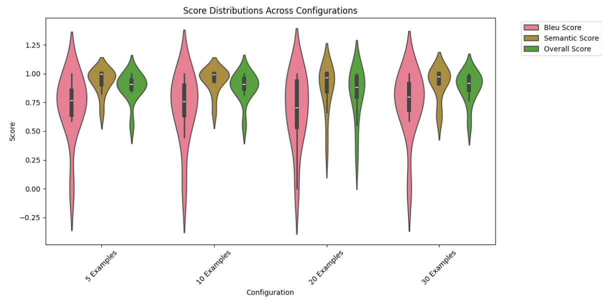
The human evaluation of the translated code samples revealed consistently strong performance across all metrics, as shown in Figure 2 and detailed in our evaluation data (Kumar Ankit, 2024). Syntax Correctness achieved a perfect score of 5.0 across all example set configurations (5, 10, 20, and 30 examples), indicating flawless preservation of code structure and keyword translation. Semantic Preservation also maintained near-perfect scores (averaging 4.95 out of 5.0), with only a slight dip to 4.8 in the 20-example configuration, demonstrating robust preservation of program logic and function behavior. Hindi Language Quality, while still scoring well, showed relatively lower scores ranging from 4.0 to 4.6, with the highest quality observed in the 5-example configuration. Notably, increasing the number of example sets did not necessarily improve translation quality, with the 5-example configuration achieving the best overall balance across all metrics. The evaluation revealed that while the system excels in maintaining code functionality and structure, there remains room for improvement in Hindi language quality, particularly in technical term consistency and natural expression.

Table 1: Translation Quality Results Across Different Example Counts

Metrics	5 Ex.	10 Ex.	20 Ex.	30 Ex.
BLEU Score	0.7150	0.7035	0.6860	0.7357
Syntax Valid Rate	1.0000	1.0000	0.9000	1.0000
Semantic Similarity	0.9296	0.9370	0.8662	0.9070
Overall Score	0.8815	0.8802	0.8174	0.8809



(a) Overall Metrics Comparison



(b) Score Distributions

Figure 3: Performance Metrics Analysis

5.2 Resources

This project was developed using the following resources:

5.2.1 Computational Resources

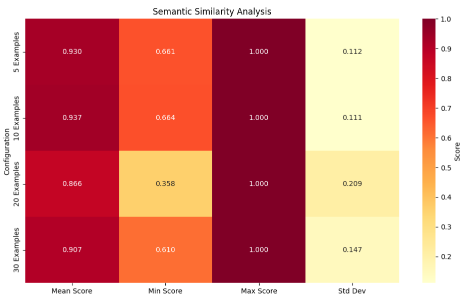
- Python environment with standard libraries and specific packages like nltk, transformers, and deep-translator
- OpenAI's GPT-4o-mini model for translation enhancement
- Local CPU computation
- Streamlit for frontend development

5.2.2 Time and Development

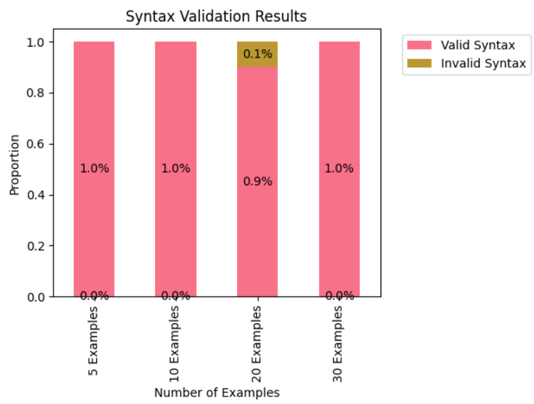
- Development Time: 4-5 weeks
- Total Development Hours: 120 hours
- Development Environment: Jupyter Notebook, VSCode

5.3 Error Analysis

Our analysis revealed several interesting cases where our translation system both succeeded and

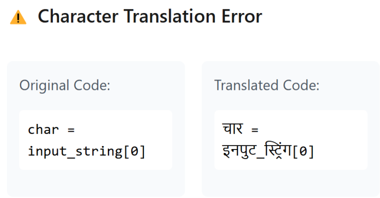


(a) Semantic Similarity Heatmap

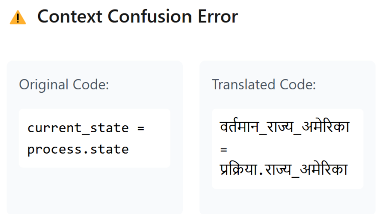


(b) Syntax Analysis

Figure 4: Syntax and Semantic Analysis

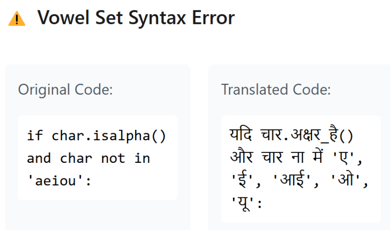


(a) Character Translation Error

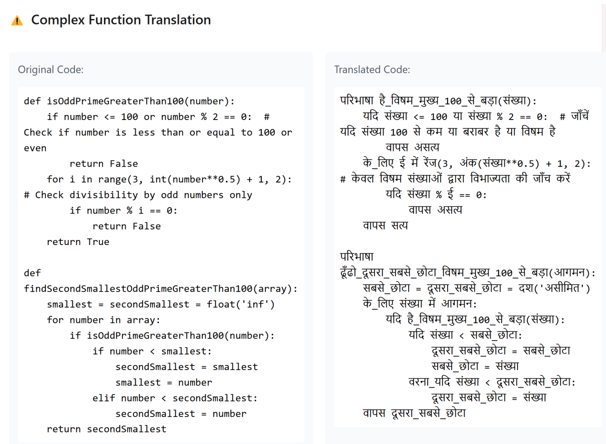


(b) Context Confusion Error

Figure 5: Failure Cases - 1



(c) Vowel Set Syntax Error



(d) Complex Function Translation

Figure 6: Failure Cases - 2

failed. Here we discuss key examples to highlight the system's strengths and limitations.

5.3.1 Failure Cases

Each of the figures (Figure 5 and 6) illustrates a different type of failure case, where the translation system struggled to convert a specific code snippet to its contextual Hindi meaning. The pipeline converts it to the Hindi equivalent of the most used english word (string, char, state, etc.) which is not always the correct translation in the context of the code. This leads to a loss of readability in the translated code.

5.3.2 Success Cases

- **Keyword Translation and Syntax Preservation**: The system demonstrated robust performance in maintaining syntactic validity across different configurations. As shown in Table 1, three out of four configurations achieved a perfect syntax valid rate of 1.0000, with only a slight decrease to 0.9000 for the 20-example configuration. This is further visualized in Figure 4(b), which illustrates the consistently high syntax validation rates.
- **Semantic Preservation**:The system showed strong capability in preserving the semantic meaning of code during translation. According to Table 1, semantic similarity scores remained high across all configurations, with the 10-example configuration achieving the highest score of 0.9370. The semantic similarity heatmap in Figure 4(a) provides a detailed view of this performance, showing consistent results across different metrics including mean scores and standard deviations.

Figure 7: Success Cases

- **Overall Translation Quality**: The system maintained high overall translation quality, as evidenced by the comprehensive metrics shown in Figure 3(a). The overall scores remained consistently above 0.81 across all configurations, with the 5-example configuration achieving the highest score of 0.8815 (Table 1). The BLEU scores, while slightly lower, still maintained respectable values ranging from 0.6860 to 0.7357, indicating good translation fidelity. The score distributions shown in Figure 3(b) further support these findings, demonstrating consistent performance across different evaluation metrics and confirming the robustness of our translation system.

6 Conclusion

Our research presents a novel approach to Python code translation between English and Hindi using a combination of rule-based keyword translation and GPT-based contextual translation. Through comprehensive human and technical evaluations, we demonstrate that this hybrid approach effectively maintains both code functionality and linguistic naturalness. summarize your contribution in three sentences.

The evaluation framework revealed several key findings:

- Perfect syntax preservation (5.0/5.0) across all test configurations indicates robust maintenance of code structure
- High semantic preservation scores (averaging 4.95/5.0) demonstrate successful retention of program logic
- Good but slightly lower Hindi language quality scores (4.0-4.6/5.0) suggest room for improvement in natural language aspects

Some limitations of our word include reliance on pre-translated keyword mappings, potential scalability challenges with larger codebases, and room for improvement in Hindi language quality. Future work could focus on expanding the keyword dictionary, enhancing the GPT model's training data, and refining the translation pipeline to address these limitations.

All team members contributed equally to the development and reporting of this project.

A Prompts and Generation Configurations

A.1 GPT-4o Mini Configuration

The following configuration was used for all GPT-4o Mini interactions:

- Model: gpt-4o-mini
- Temperature: 0
- Maximum retries: 3
- Response format: Raw code without explanations

A.2 System Prompts

A.2.1 Code Translation System Prompt

The following system prompt was used for the main translation task:

> "You are a Expert Python code translator who understands the nuanses of language in coding and converts code from English to Hindi code while preserving functionality. Return only the translated code without any explanation."

A.2.2 Back-Translation System Prompt

For evaluation purposes, the following system prompt was used:

> "You are a Python code translator converting Hindi code to English."

A.3 User Prompts

A.3.1 Main Translation Prompt Template

This template was used for each translation, with examples and code filled in dynamically:

> Complete the translation of this partially English Python code to completely Hindi python code:
>
> - Translate variable names, function names, strings and comments to Hindi
> - Join multi-word Hindi translations with underscores
> - Break down compound English words separated by underscores and translate each part into sensible Hindi and join them back with underscores
> - Preserve code structure and syntax
>
> Here are some examples of translations:
>
> [Example 1]:
> English code:
> {original_code_1}
> Hindi translated code:
> {translated_code_1}
>
> ---
>
> [Example N]:
> English code:
> {original_code_n}
> Hindi translated code:
> {translated_code_n}
>
> Now translate partially translated code to completely in Hindi:
> {code_to_translate}

A.3.2 Back-Translation Prompt Template

Used for evaluation through back-translation:

> Complete the translation of this partially translated Python code to English:
>
> - The code already has Python keywords translated to English
> - Translate remaining variable names and comments
> - Convert Hindi compound words (with underscores) to appropriate English terms
> - Preserve code structure and syntax
>
> Partially translated code:
> {partially_translated_code}

A.4 Example Configurations

Four different configurations were tested, varying the number of examples provided to the GPT model for each translation task:

- Configuration 1: 5 examples
- Configuration 2: 10 examples
- Configuration 3: 20 examples
- Configuration 4: 30 examples

References

Premkumar Devanbu. 2015. New initiative: the naturalness of software. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, page 543–546. IEEE Press.

Google. 2024. Google translate. `http://translate.google.com`. Accessed: 3-December-2024.

jtatman. 2024. Python code dataset 500k. `https://github.com/jtatman/python-code-dataset-500k`. Accessed: 05-12-2024.

Tewary Antara Kumar Ankit. 2024. Human eval feedback. `https://github.com/StringAna/PyLinguist/blob/main/Reports/Final-Report/Human_Eval_Feedback/Human_Eval_feedback.csv`. Accessed: 6-December-2024.

OpenAI. 2024. Gpt-4o mini. `https://www.openai.com`. Accessed: 3-December-2024.

Jordan Ott, Abigail Atchison, Paul Harnack, Adrienne Bergh, and Erik Linstead. 2018. A deep learning approach to identifying source code in images and video. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 376–386, New York, NY, USA. Association for Computing Machinery.

Joshua Otten, Antonios Anastasopoulos, and Kevin Moran. 2023a. Unipy web tool. `https://universal-pl.github.io/UniPy/`. [Online; accessed 10-October-2023].

Joshua Otten, Antonios Anastasopoulos, and Kevin P Moran. 2023b. Towards a universal python: Translating the natural modality of python into other human languages. In *Proceedings of ICSME 2023*, Bogota, Colombia. ICSME.

Chris Piech and Sami Abu-El-Haija. 2019. Human languages in source code: Auto-translation for localized instruction. *Proceedings of the Seventh ACM Conference on Learning @ Scale*.

Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2024. Domain adaptive code completion via language models and decoupled domain databases. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, page 421–433. IEEE Press.