Code to Code Conversion from Java to Python Using T5-Small

Anirudh Dambal[1], Harish Patil[1], Pavan Bhakta[1], and Anusha Adarakatti[1]

[1]School of Computer Science and Engineering, KLE Technological University, Hubballi, Karnataka, India, 580031

This paper introduces a new method of code-to-code translation from Java to Python using the T5-small transformer model, based on a custom-generated dataset. Previous works often depend on language-specific libraries or attributes; however, our dataset prohibits such inbuilt functionalities to promote the creation of fully translatable, algorithmically pure code. The dataset was generated with the help of the Gemini model, where sample text was initially translated into pseudocode, then coded into both Java and Python without language-specific shortcuts. This dataset was next used to train a T5-small model, which learned generalized patterns and structures for code translation. Evaluation results indicate the potential for using the T5-small model to translate Java code into Python with good accuracy in producing semantically equivalent and syntactically correct code. Our approach opens a promising direction for tool-assisted cross-language code translation, with significant implications for software development, maintenance, and education in multilingual coding environments.

Keywords: Transpilation, Encoder-Decoder Architecture, Multilingual Code Translation, Custom Dataset Generation, Pseudocode-Based Training, Gemini API, Cross-Language Programming

1 Introduction

Modern software development increasingly relies on specialized programming languages tailored to specific domains. These Domain-Specific Languages (DSLs) improve code clarity, usability, and performance by providing optimizations and abstractions suited to their target applications. Examples include Spark for distributed computing, NumPy for array processing, TACO for tensor processing, and P4 for network packet processing. While these DSLs offer significant benefits, integrating them into existing workflows often requires rewriting legacy code—a tedious and error-prone process that risks introducing bugs and altering the original code's behavior. This challenge of translating code between high-level languages is commonly referred to as transpilation.

Recent advancements in Large Language Models (LLMs), such as BERT (Devlin et al., 2019) and GPT-3 (Brown et al., 2020), have shown promise in automating complex programming tasks, including code generation, repair, and testing. However, generating reliable and formally correct code remains a significant hurdle. Most existing research focuses on code generation without guarantees of correctness (e.g., Li et al., 2022, 2023; Rozière et al., 2024), or on generating annotations and invariants for existing code (Pei et al., 2023; Chakraborty et al., 2023). Additionally, formal verification tools rely on specialized languages like SMT-LIB and Dafny, which are often underrepresented in LLM training datasets, further complicating the application of these models in verification contexts.

The success of pre-trained models in natural language processing, such as BERT, GPT, and T5 (Raffel et al., 2020), has inspired researchers to adapt similar methods for programming languages. Efforts in this direction (e.g., Svyatkovskiy et al., 2020; Kanade et al., 2020; Feng et al., 2020) have produced encouraging results, particularly for tasks like code translation and synthesis.

Building on these advancements, this paper explores the use of the T5 transformer model for transpiling code between two syntactically distinct languages: Java and Python. We curate a dataset of paired Java and Python programs and train the T5 model to translate Java code into its Python equivalent, demonstrating its potential for reliable transpilation in real-world scenarios.

2 Related work

The application of machine learning models to programming tasks has seen significant advancements in recent years. Early work explored novel ways to represent code for machine learning, leveraging hierarchical structures such as Abstract Syntax Trees (ASTs). Building on these representations, researchers like Tai et al. (2015) introduced Tree-LSTM models, which encode tree structures effectively by processing nodes recursively, starting from the leaves. This foundational work laid the groundwork for subsequent applications in program analysis and transformation. Chen et al. (2018) extended these ideas by proposing a tree-to-tree encoder-decoder framework tailored for code, enabling structured transformations between different tree representations. Drissi et al. (2018) further refined this approach by introducing models that encode program trees with binarization techniques, such as the Left-Child Right-Sibling representation, demonstrating improvements in tasks like program translation. Together, these works highlighted the potential of structured neural models for code understanding and generation. The advent of neural machine translation (NMT) frameworks inspired applications beyond natural language. Tufano et al. (2018) applied NMT to software maintenance tasks, focusing on automated bug fixing. By mining GitHub for bug-fix commits and abstracting code changes into generalizable patterns, they trained models capable of generating bug-fixing patches that closely mimic developer edits. Their approach, which utilized tools like GumTree for edit tracking, achieved high syntactic correctness in generated patches and showcased the utility of NMT for learning transformations at the AST level. In parallel, Lachaux et al. (2020) introduced TransCoder, a system for translating functions between programming languages without requiring parallel training data. By leveraging monolingual source code and unsupervised learning techniques, TransCoder captured complex, language-specific patterns and outperformed rule-based commercial systems. This work also contributed valuable resources, including a dataset of parallel functions with unit tests, to facilitate further research in code translation. Katz et al. (2019) tackled the problem of decompilation, where low-level code is converted back into a high-level programming language. Traditional decompilers relied on expert-designed rule-based methods, which were expensive to adapt to new languages or constructs. Katz and colleagues proposed a neural approach, training NMT models to perform decompilation by reversing compiler processes. Their framework demonstrated high success rates on tasks such as translating LLVM IR and x86 assembly back to source code, proving the viability of machine learning for decompilation. Finally, Armengol-Estapé et al. (2021) explored the use of transformer models for compilation tasks, marking a shift toward leveraging large pre-trained language models in program analysis and transformation. Transformers, known for their success in natural language processing, opened new possibilities for automating complex programming tasks like code generation, translation, and repair. These advancements collectively illustrate the evolution of machine learning techniques for code-related tasks, from tree-based models to neural machine translation and transformers. Each step has contributed to a broader understanding of how programs can be represented, translated, and optimized, laying the foundation for the methods explored in this work.

3 Proposed work

This work presents a transpilation framework for automatic translation between high-level programming languages, namely Java and Python, using a fine-tuned T5-small model on a generated dataset of the two respective code pairs. The framework deploys T5-small's transformer-based encoder-decoder architecture in order to capture syntactic and semantic differences existing between the two above languages. Its implementation focuses both on training and inference, whereby a robust pipeline is achieved for translating Java code into its Python equivalent.

3.1 Dataset Preparation

3.1.1 Overview

The dataset developed for the purpose of this research was generated so that it could ensure diversity and semantic richness of paired Java and Python code snippets that are grounded in pseudocode instructions. Through Google's Gemini API, the generation of the dataset automatically translates pseudocode into both Java and Python implementations in the creation of highly quality source-aligned programming examples. The resulting dataset consists of three components: pseudocode, Java code, and Python code. The pseudocode column was not directly utilized during the training process. Instead, only the Java and Python code pairs were employed for model training, as the focus was on translating between these two programming languages

3.1.2 Data Sources

The pseudocode instructions were sourced from two collections:

1. `codeparrot/xlcost-text-to-code/Java-program-level` (subset with 11K rows) [huggingface.co]: A collection of pseudocode entries specifically curated to be well-suited for Java-style implementations.

2. `codeparrot/xlcost-text-to-code/Python-program-level` (subset with 10.6K rows) [huggingface.co]: A similar collection designed for Python-oriented tasks.

Each entry in these collections is represented as a dictionary containing:

- `text`: The pseudocode instruction.
- `code`: Corresponding code snippets derived from existing resources.

3.1.3 Data Generation Pipeline

The data preparation process included the following steps:

1. **Configuration and API Integration:** The Google Gemini API (model version `gemini-1.5-flash`) was configured with an API key to serve as the generative backend for converting pseudocode to programming language implementations. Requests to the API were structured as detailed prompts, instructing the model to generate code directly fulfilling the pseudocode requirements.

2. **Dataset Initialization:** A pandas DataFrame was initialized with three columns: `Pseudo Code`, `Java`, and `Python`. This served as the core structure for storing and updating the dataset as new code samples were generated.

3. **Code Generation Process:** For each pseudocode entry:
    - A prompt was sent to the Gemini API to generate a Python implementation, explicitly requesting a functional code block without comments or extraneous text.
    - A second prompt generated the corresponding Java implementation under the same constraints.
    - The responses were post-processed to ensure clean and consistent formatting.

4. **API Rate Limiting:** To avoid overwhelming the API and ensure adherence to rate-limiting constraints, a delay of 4 seconds was enforced between consecutive requests.

5. **Balancing Java and Python Sources:** The process was repeated for both the Java-oriented and Python-oriented pseudocode collections, with 667 examples processed from each collection to ensure balance and diversity across languages.

3.1.4 Dataset Statistics

The final dataset contains 667 rows, each representing a pseudocode instruction alongside its semantically equivalent Java and Python implementations. This structured dataset serves as the foundation for training and evaluating the T5-based code translation model. As noted earlier, pseudocode column was not used in training, rather used only for prompts to Gemini API, which generated the corresponding Java and Python program snippets.

Pseudocode	Java Snippet	Python Snippet
`Maximum Prefix Sum possible by merging two given arrays; Java Program to implement the above approach; Stores the maximum prefix sum of the array A []; Traverse the array A[] Stores the maximum prefix sum of the array B []; Traverse the array B[];`	`public static int maxPrefixSum(int[] A, int[] B) { int maxPrefixSumA = 0, maxPrefixSumB = 0; for (int i = 0; i < A.length; i++) { maxPrefixSumA = Math.max(maxPrefixSumA, maxPrefixSumA + A[i]); } for (int i = 0; i < B.length; i++) { maxPrefixSumB = Math.max(maxPrefixSumB, maxPrefixSumB + B[i]); } return Math.max(maxPrefixSumA, maxPrefixSumB); }`	`def max_prefix_sum(A, B): max_prefix_sum_A = 0 for i in range(len(A)): max_prefix_sum_A = max( max_prefix_sum_A, max_prefix_sum_A + A[i]) max_prefix_sum_B = 0 for i in range(len(B)): max_prefix_sum_B = max( max_prefix_sum_B, max_prefix_sum_B + B[i]) return max( max_prefix_sum_A, max_prefix_sum_B)`
`Ways to remove one element from a binary string so that XOR becomes zero | Java program to count number of ways to remove an element so that XOR of remaining string becomes 0. ; Returns number of ways in which XOR become ZERO by remove 1 element ; Counting number of 0 and 1 ; If count of ones is even then return count of zero else count of one ;`	`public static int countWays(String str) { int count0 = 0; int count1 = 0; for (int i = 0; i < str.length(); i++) { if (str.charAt(i) == '0') { count0++; } else { count1++; } } if (count1 % 2 == 0) { return count0; } else { return count1; } }`	`def count_ways(s): count_zero = s.count('0') count_one = s.count('1') if count_one % 2 == 0: return count_zero else: return count_one`

Table 1: Sample code snippets from generated dataset

3.1.5 Key Advantages

1. **Quality and Consistency:** The use of Gemini API ensures high-quality translations, reducing noise often found in manually paired datasets.
2. **Diversity:** The dataset captures variations in pseudocode expression and their implementations in two syntactically distinct languages, ensuring robust model training.
3. **Scalability:** The automated pipeline allows for easy scaling to include additional

examples or other languages in future iterations.

This dataset is critical in providing the T5 model with sufficient paired training samples to learn effective code translation between Java and Python.