Python to Java Code Translation

Ansh Agrawal **Eric So** **Hoa Nguyen** **Karanvir Khanna**
University of Toronto

Abstract

This project explores neural program translation from Python to Java using Transformer encoder–decoder architecture. Trained on Java and Python subsets of IBM's Project CodeNet programming problem submissions Puri et al. [2021]IBM Research [2021], our model tokenizes source code using a shared 8,000-token SentencePiece Byte-Pair Encoding (BPE) vocabulary Kudo and Richardson [2018]. The encoder reads Python tokens while the decoder generates Java tokens through masked self-attention and cross-attention mechanisms. Java tokens are merged into the target Java code with the goal of replicating the same functionality as the original Python function.

1 Introduction

The growing diversity of programming languages presents an ongoing challenge for software maintenance. Manually translating from Python to Java is time-consuming and prone to error, especially given their differences in syntax. Automating this process would significantly improve code reuse and development overhead.

Traditionally, Abstract Syntax Tree (AST) based translation methods represent source code as hierarchical trees of syntactic elements. However, they struggle to generalize across structural variations. In contrast, deep learning offers a promising framework for this task. neural architectures such as recurrent sequence-to-sequence models and transformers can learn direct mappings between language pairs from data. Moreover, self-attention computes weighted interactions among all token pairs in a sequence, enabling the model to represent contextual dependencies regardless of their distance.

Recent advances in Neural Machine Translation and pretrained code models such as PLBART and CodeT5 Ahmad et al. [2023] have demonstrated promising results for cross-language translation. Building on this progress, Ahmad et al. [2023] introduced the AVATAR dataset, a large parallel corpus for Java–Python translation, establishing strong baselines with pretrained Transformers.

In this work, we extend this research by constructing a reproducible Transformer baseline on Project CodeNet for translating Python to Java. Our goal is to measure syntactic and semantic fidelity using metrics from AVATAR.

2 Background and Related Work

While CSC413 covers general sequence modeling with recurrent networks, programming languages introduce additional structure and constraints. Therefore, understanding how neural models learn relationships in code requires specialized datasets, preprocessing, and evaluation metrics. In code translation, AST organizes the program into a hierarchical tree where each node represents a syntactic construct. Therefore, many traditional code translation systems rely on them to ensure syntactic validity when converting between languages. However, deep learning models such as Transformers can often learn similar structural relationships directly from raw token sequences, reducing the need for explicit AST parsing. To measure translation quality, we use Bilingual Evaluation Understudy

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

(BLEU) and CodeBLEU. The BLEU score measures the n-gram overlap between predicted and reference code, which captures syntactic similarity. It rewards outputs whose token sequences closely match the reference but does not consider meaning or structure. CodeBLEU Ren et al. [2020] extends BLEU for programming languages by integrating syntactic and semantic analysis. This ensures that translated code is not only textually similar but also structurally valid and executable. Early research on code translation adapted sequence-to-sequence architectures Vaswani et al. [2017] with attention mechanisms. The Transformer architecture later replaced recurrence with self-attention, enabling direct global context modeling and greater parallelization. These advances made Transformer-based models well-suited for translating code, where dependencies among variables and control structures may span many tokens. In the AVATAR dataset, the TransCoder-ST Rozière et al. [2021] model achieved BLEU = 66.0 on the Python to Java direction

. The results show that these Transformers are doing well. However, we believe that the performance can be improved.

Our project builds upon Project CodeNet with the goal of improving from the models in AVATAR. We evaluate translation quality using the same metrics as AVATAR such as BLEU and CodeBLEU, then analyze how a our Transformer performs relative to pretrained benchmarks.

3 Dataset Description and Preprocessing

Unless noted otherwise, the following sections are based on the data and metadata provided in the Project CodeNet GitHub repository at `https://github.com/IBM/Project_CodeNet` IBM Research [2021].

Dataset Scale: The complete Project CodeNet contains approximately 13.9 million code submissions across 4,053 distinct programming problems. The submissions span 55 programming languages, with the six most popular languages (C++, Python, Java, C, Ruby, C#) accounting for 95% of all submissions. Our local subset focuses on Python (3,286,314 submissions) and Java (712,153 submissions).

Pair Construction: For each problem that has at least one Java submission and at least one Python submission, we take the Cartesian product of all Java and all Python solutions for that problem to generate candidate Java-Python pairs. The dataset includes 3,286,314 Python submissions and 712,153 Java submissions, yielding a total of roughly **2,664,492,206** Java-Python pairs derived from **same-problem** Cartesian products.

3.1 Data Organization and Structure

The dataset follows a hierarchical directory structure:

```
BigCodeNet/
|-- data/
|   |-- p00000/
|   |   |-- Python/ s########.py
|   |   `-- Java/   s########.java
|   |-- p00001/ ...
|   `-- ... (4,053 problems)
|-- metadata/
|   |-- problem_list.csv
|   `-- p00000.csv, p00001.csv, ... (4,054 CSVs)
|-- problem_descriptions/
|   `-- p00000.html, ... (3,999 HTMLs)
`-- derived/
    |-- duplicates/
    |-- input_output/
    `-- benchmarks/
```

3.2 Features of the Dataset

The dataset includes both problem-level and submission-level metadata. Problem-level information is stored in `problem_list.csv`, where each **Problem ID** (`p[0-9]{5}`, e.g., `p00001`) acts as an

anonymized unique identifier used to construct Python–Java code pairs. Submission-level metadata is stored in per-problem CSV files (`p?????.csv`). Each submission is identified by a unique **Submission ID** (`s[0-9]{9}`) used for pairing and traceability. The **Language** field in the metadata specifies both the original and standardized language name with its file extension, allowing us to restrict the dataset to Python and Java. The **Execution Status** field in the metadata records one of twelve outcomes, including Accepted (AC, approximately 53.6%), Wrong Answer (WA, approximately 29.5%), and various error types such as Compile Error (CE), Runtime Error (RE), Time Limit Exceeded (TLE), and Memory Limit Exceeded (MLE). To ensure high-quality supervision, we will only use Accepted submissions when building translation pairs for training.

3.3 Data Quality

Duplicate Detection: The `derived/duplicates/` directory contains near-duplicate analysis using token-based Jaccard similarity as noted in the `README` file in the dataset.

Algorithmic Complexity: Problem content spans a broad range of algorithmic complexity. Categories include basic I/O and arithmetic, core data structures such as arrays, sorting, searching, stacks, queues, trees, and graphs, classical algorithms like dynamic programming, greedy methods, graph traversal, and string manipulation, and advanced topics such as number theory, computational geometry, and optimization.

3.4 Data Cleaning and Preprocessing Pipeline

From 4,053 problem metadata files, we select only those with both Python and Java submissions, filtering to *Accepted* ones to ensure correctness. For each problem, we take the Cartesian product of all accepted Python and Java solutions to form candidate pairs. To avoid overfitting, we split data by `problem_id` rather than `submission_id`, since solutions to the same problem often share structure and logic. We randomly permute the list of problems with a fixed seed and divide it into 70% training, 15% validation, and 15% test splits, letting each pair inherit its problem's split. Important note, given the 2.66 billion possible pairs, we sample a smaller, computationally feasible subset for model training, trimming problems with disproportionately many Python or Java submissions so that all problems contribute roughly similar numbers of pairs, this again avoids overfitting the model to a certain genre of problem.

Key challenges for Python $\rightarrow$ Java translation include the **syntax gap**, such as dynamic versus static typing, indentation versus braces, and differing standard libraries; maintaining **semantic fidelity** to preserve algorithmic logic, complexity, idioms, and data structures; handling **one-to-many mappings**, where a single Python construct can correspond to multiple valid Java forms (for example, a list may translate to either an array or an `ArrayList`); and managing **length mismatch**, since Java implementations are typically 1.5–3 times longer due to explicit types and boilerplate.

4 Model Architecture

We use a **Transformer encoder-decoder** model Vaswani et al. [2017] for Python $\rightarrow$ Java translation as this architecture supports direct translation from the given Python code. The encoder reads the Python program, while the decoder generates the Java program. Inputs and outputs are tokenized by lexing Python and Java into code tokens (identifiers, literals, and punctuation) and Python indentation is explicitly serialized using `<INDENT>` and `<DEDENT>` to preserve indentation and provide cues for mapping it to Java's braces. We then apply a subword tokenizer (SentencePiece BPE) with a shared vocabulary of $\tilde{8}$k trained on a combined Python and Java corpus Kudo and Richardson [2018], so long identifiers can be split into reusable parts, removing unknowns and improving name copying across languages. We cap the source and target sequence lengths at 512 tokens to control the attention cost. The network is a pre-LayerNorm transformer (L_{enc}=4, L_{dec}=4, d_{model}=384, n_{heads}=6, d_{ff}=1536, and dropout = 0.1.) and is small enough to be trained on a single GPU yet *large* enough to capture long-range code dependencies. The overall architecture is illustrated in Figure 1.

Training Teacher forcing with token-wise cross-entropy is used to optimize next-token accuracy. We mask `<pad>` in attention and loss so that padding doesn't influence gradients. To optimize, we use AdamW with a short warmup followed by decay to stabilize learning on long sequences and small batches.

Inference We encode each Python program to get a fixed source representation, then generate Java code token by token. The decoder at each step uses masked self-attention over the tokens it has produced and cross-attends to the encoded Python to get the next token. We use a small beam (size 3-4) to get a few promising partial outputs and select the best sequence, then merge subwords to form the final Java code.

5 Model Architecture Figure

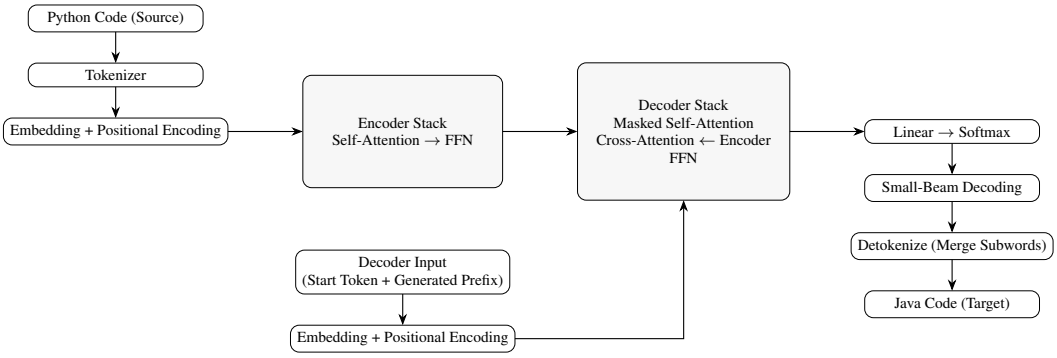

Figure 1: **Transformer encoder-decoder architecture for Python→ Java code translation.** The encoder processes tokenized Python source code using self-attention layers, while the decoder generates Java tokens autoregressively using masked self-attention and cross-attention to the encoder's output. Small-beam decoding gives the resulting Java code.

6 Ethical Considerations

6.1 Dataset Licensing and Attributions

Use of the Project CodeNet dataset, governed by an Apache License 2.0, necessitates attribution of the original license but allows modifications and derivative work to exist under a sublicense; all resulting code and models must include text of the original license. Apache 2.0 License allows commercial use and does not have copy-left requirements, which does not restrict this project's use case for academic research purposes.

6.2 Style and Framework Biases

Framework and style biases are an inherent risk when training on datasets with prevalent patterns or idioms, potentially privileging certain frameworks. Similarly, semantic gaps such as differences in platform capabilities or missing 1:1 mappings for certain features could cause models to generate brittle or incorrect outputs in edge cases, especially for features and libraries excluded from the dataset.

6.3 Privacy and Security

Privacy and security considerations remain paramount. The dataset and all training data are scanned for secrets (e.g. credentials, API keys), potential personally identifiable information, or any embedded sensitive content. As the dataset consists of submissions to online programming problems, the probability of credentials or PII existing is lower.

6.4 Mitigations

Mitigations include license compliance checks, required attribution in outputs, and clear terms of use forbidding both plagiarism and misuse. The paper and the model must include the original Apache 2.0 License from the Project CodeNet dataset in addition to this project's licenses for modified content.

All data and code are scanned for secrets and sensitive content, with flagged content removed or anonymized prior to training.

Explicit limitations and performance expectations are acknowledged: the project cannot guarantee absolute removal of all sensitive, private, or insecure elements from the data or outputs; nor can it prevent all forms of downstream misuse, reinterpretation, or error propagation due to semantic gaps or unrecognized library risks. The intended use case of this model is for educational purposes and not for production use.

7 Work Division

7.1 Ownership and Deliverables

This project is divided into deliverables so that at least one member is the owner of a deliverable and at least one dedicated peer reviewer assigned to each deliverable.

Hoa: Owns the abstract, introduction, and background & related work. Frames the problem, motivation, and justifies the approach for this problem statement. Compares and contrasts related work (code translation, semantic preservation, evaluation methods). Updates the Abstract late in the cycle so it reflects the stabilized architecture and scope.

Karan: Owns all data exploration work. Designs data cleaning and pre-processing procedures; conducts exploratory analysis and produces dataset statistics. Coordinates with Ansh on model requirements; compiles metrics; authors the Results and Limitations sections consistent with the data constraints.

Ansh: Owns model architecture and experiments. Finalizes tokenizer setup, encoder and decoder specifics, attention/decoding choices; produces the architecture figure with captions; cross-references ideas with Karan; and is responsible for training the model.

Eric: Authors Ethical Considerations, Work Division, and Conclusion. Identifies and mitigates risks around licensing, privacy/security, misuse, bias/style skew, and defines responsible release practices. Document roles, and milestones to keep the project on track. Writes the conclusion to summarize the results and tie the results back to the problem statement while acknowledging the limitations.

7.2 Reviewer Roles

Reviewers play an important role in this project by ensuring deliverables are met on time as well as share context within the team. Below are the reviewer assignments for each member.

Karan reviews Abstract, Introduction, and Related Work. He checks that the problem framing matches the chosen data and model.

Hoa reviews Data, Model Architecture, Results, and Discussion. He validates dataset representativeness and preprocessing steps, confirms architecture choices are clearly justified.

Eric reviews Model Architecture, Figure, Results, and Limitations. He ensures model training steps are documented and reproducible. He also assesses the model for limitations in order to comply with ethical considerations.

Ansh reviews Ethical Considerations, Work Division, and Conclusion. He verifies that ethical mitigations map to the real pipeline and that the work split reflects dependencies and timelines.

7.3 Timeline and Milestones

The team will communicate asynchronously throughout the project, with key meeting dates being synchronous (bolded below). **Proposal:**

- **Oct 24**–27: Dataset finalized; preprocessing pipeline stub; baseline defined; architecture draft; proposal sections outlined.
- **Oct 27**: Task exploration and breakdown; Role assignment
- Nov 2: Full proposal draft (all sections complete, figures placeholder or first pass).

- Nov 3–4: Peer reviews per assignments; incorporate edits; consistency and citation pass; compile LaTeX cleanly; Reviews due on Nov 4.
- **Nov 5**: Final integration; Final full review of proposal with all incorporated changes; Review final checklist.
- Nov 7: Submit proposal.

Final Report:

- Nov 8–12: Data cleaning, formatting, and split into pairs and into training, validation, and test sets.
- Nov 13–20: Training runs; baseline comparisons; error analysis; limitations logged.
- **Nov 18**: Review of training progress, potential model changes, and alignment on next steps.
- Nov 20–25: Results, discussion, and figure polishing; ethics and documentation finalized; reproducibility checklist complete.
- Nov 26: Draft of all sections due; Peer reviews begin.
- **Dec 1**–3: Integrated final draft; peer review; formatting and references lock.
- Dec 4: Final full review from all members; Review pre-submission checklist
- Dec 5: Submit final report.

7.4 Collaboration Standards

The team has agreed to a set of collaboration standards that increases productivity. Firstly, asynchronous communication is the primary source of communication. Setting the accurate expectations of a deliverable and its timeline early on helps the team plan ahead.

Secondly, for each synchronous meeting listed above, members are expected to communicate the agenda ahead of time and ensure they are prepared with their material before the meeting.

Lastly, prior to submission, each member shares the responsibility of performing checks across the entire report regardless of the reviewer roles assigned. Members are encouraged to view the progression of other deliverables to ensure all members have a complete view of the project.

References

Wasi Uddin Ahmad, Md Golam Rabiul Tushar, Saikat Chakraborty, and Kai-Wei Chang. Avatar: A parallel corpus for java-python program translation. *Findings of ACL 2023*, 2023. URL `https://arxiv.org/abs/2108.11590`.

IBM Research. IBM/Project_CodeNet: Initial release 1.0. Zenodo, 2021. URL `https://doi.org/10.5281/zenodo.4814770`. [Software].

Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *CoRR*, abs/1808.06226, 2018. URL `http://arxiv.org/abs/1808.06226`.

Ruchir Puri, David S. Kung, Greg Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jinjun Chen, Monojit Choudhury, Laurent Decker, Veronika Thost, Luca Buratti, Shreya Pujar, Saurabh Ramji, Uri Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021. doi: 10.48550/arXiv.2105.12655. URL `https://doi.org/10.48550/arXiv.2105.12655`.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Natarajan Sundaresan, Ming Zhou, Antonio Blanco, and Shuming Ma. Codebleu: A method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020. URL `https://arxiv.org/abs/2009.10297`.

Baptiste Rozière, Jie M. Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*, 2021. URL `https://arxiv.org/abs/2110.06773`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017. URL `https://arxiv.org/abs/1706.03762`.