# If-Statement Prediction

Hongfei Du

October 2025

## 1 Dataset Construction

### 1.1 Repository Collection

Python repositories were automatically retrieved via the GitHub API using the query: `language:Python, stars:>100`. Each repository was shallow-cloned (depth 1) to avoid unnecessary history data and reduce noise from outdated commits. The collection process was designed to capture a wide range of programming styles while maintaining high-quality and well-documented projects. Non-relevant folders such as `tests/`, `docs/`, build scripts, and virtual environments were excluded to ensure the corpus primarily reflected real, functional code used in production settings rather than tutorial or synthetic snippets. This step provided a clean and diverse foundation for constructing both the pre-training and fine-tuning datasets.

### 1.2 Function Extraction

Functions were extracted using an AST-based pipeline that systematically parses Python files into syntax trees. Compared with simple regex-based extraction, the AST approach guarantees structural correctness and prevents incomplete or nested code fragments from being included. To ensure semantic richness and filter out trivial or low-information functions, several quality criteria were applied:

- **Language filter:** discard files with >5% Chinese in strings or >10% in comments to keep the language distribution consistent.
- **Complexity:** retain only functions containing control-flow constructs (`if/for/while/try/with`) and a minimum computed complexity score of 2, ensuring logical depth.
- **Triviality filter:** skip functions consisting solely of assignments, imports, or `pass` statements.
- **Length:** keep functions between 3 and 200 lines to balance brevity and contextual completeness.

Each valid function was serialized into a dictionary of metadata: {`repo, path, text, sha1`}, which enables traceability and deduplication across repositories. This stage produced a structurally consistent corpus where each example corresponds to a syntactically valid, self-contained function.

### 1.3 Dataset Construction

Two types of datasets were derived from the extracted functions. **For pre-training**, all function bodies were saved as independent text samples in `pretrain.jsonl`, allowing large-scale unsupervised learning of code patterns and naming conventions. **For fine-tuning**, a controlled masking procedure was applied: one random `if` condition per function was replaced with a special placeholder `<IF_MASK>`, and the original conditional expression served as the target label. This design explicitly formulates the task as sequence-to-sequence infilling, aligning it with the pre-training objective while focusing on conditional reasoning.

| Split | Portion | Purpose | Approx. Size |
|---|---|---|---|
| Pre-train | — | Self-supervised | 222k |
| Train | 80% | Supervised learning | 49k |
| Validation | 10% | Early stopping | 6k |
| Test | 10% | Final evaluation | 6k |

Table 1: Dataset statistics and splits.

### 1.4 Cleaning and Deduplication

A dedicated cleaning script sanitized all code and textual content. URLs, Base64 blobs, email addresses, file paths, and hexadecimal sequences were replaced with standardized placeholders to prevent memorization of meaningless identifiers. In addition, non-English segments and residual noise were filtered out using a lightweight language heuristic, keeping only English or code-like content. To minimize redundancy, duplicate and near-duplicate entries were removed using both exact hashing and SimHash (Hamming distance $\leq 3$). This process yielded a corpus with stable token distribution and low duplication ratio, providing a reliable basis for tokenizer training and model pre-training.

## 2. Model Design

### 2.1 Tokenizer

A ByteLevel-BPE tokenizer was trained on the pretraining corpus with a vocabulary of 30k. Special tokens included:

`<s>, </s>, <pad>, <unk>, <mask>, <IF_MASK>, <extra_id_0> ... <extra_id_99>`

These support both span infilling and the downstream conditional-prediction task.

### 2.2 Pre-training (Multi-Objective)

Using **CodeT5-small (220M)** as the base model, the script (`codet5p_multiobj.py`) performed continued pre-training with two complementary objectives:

1. **Span-Denoising (T5-style):** randomly mask 15% of tokens and reconstruct missing spans using sentinels.
2. **Identifier-Masking (LANCE-style):** replace variable, function, and class names with sentinels to promote semantic understanding.

The combined loss was a weighted mixture (0.6×Span + 0.4×Ident). Training ran for 3 epochs using AdamW (lr $= 5\times10^{-5}$, batch size 64). The pre-trained checkpoint exhibited stable convergence with average training loss $\approx 5.4$.

## 3. Fine-Tuning Procedure

Fine-tuning was performed using the same tokenizer and the pre-trained model checkpoint.

- **Input:** function body containing `<IF_MASK>`
- **Target:** masked Boolean expression

**Training configuration:**

- Learning rate $= 3 \times 10^{-4}$, epochs $= 5$
- Batch size $= 16$, beam size $= 4$
- Max source/target length $= 512/64$
- Loss = cross-entropy; normalization via AST + regex simplification

Evaluation metrics include:

- Exact Match (EM)
- BLEU and chrF (textual similarity)

# 4. Results

| Test Set | Exact Match | BLEU | chrF |
|---|---|---|---|
| Generated Test | 0.372 | 43.67 | 54.93 |
| Provided Test | 0.393 | 42.77 | 56.32 |

In addition to textual similarity metrics, each prediction was also assigned a **confidence score** based on the model's average per-token log-probability during beam generation. The exact confidence values for all samples are provided in the accompanying output files (`generated-testset.csv` and `provided-testset.csv`), which record both the predicted conditions and their corresponding confidence scores.

## 4.1 Analysis

The model demonstrates moderate success in predicting correct and logically equivalent `if` conditions. The functional equivalence (0.25–0.18) is lower than lexical similarity, indicating syntactic variation but partial logical correctness. BLEU ≈43 confirms the model learned meaningful code patterns despite limited fine-tuning data. Notably, the chrF score slightly surpasses BLEU, suggesting that the model captures fine-grained token overlap and local consistency even when overall structure diverges.

**Error cases:**

- Incorrect operator choice (`>` vs. `>=`)
- Missing compound conditions (`and`/`or`)
- Occasional overgeneration beyond one condition

## 4.2 Example

```
Input:
def is_valid(x):
    if <IF_MASK>:
        return True
    return False
Prediction: x is not None and x > 0
Reference:  x != None and x > 0
Result:     Functionally Equivalent
```

# 5. Discussion and Conclusion

The AST-based data pipeline ensured high-quality, non-trivial code samples, while the dual-objective pre-training improved representation of control-flow and identifiers. Although CodeT5-small achieved only ∼0.39 exact match, it generalized reasonably across unseen projects. Compared with a baseline trained from scratch, the pre-trained model showed faster convergence and greater stability, indicating that structural pretraining indeed facilitates logical pattern recovery. Future work may explore integrating AST-graph encoders or contrastive regularization to better capture conditional dependencies across complex functions.