Training a Transformer model for Predicting if statements

1. Introduction

This report includes the design, implementation, and evaluation of a Transformer-based model to predict the conditional expression of if statements in Python code. The model's objective is to generate the correct expression, given the context of a function with a masked if condition.

The project's core requirements included using a custom-trained tokenizer and a two-phase training process: a pre-training phase and a fine-tuning phase.

1. **Pre-training:** Trained the model on a large corpus of general Python code to learn the language's syntax and semantics using a Masked Language Modeling (MLM) objective.
2. **Fine-tuning:** Specialized the pre-trained model on a curated dataset specifically for the task of if-condition prediction.

2. Dataset Construction

The dataset construction process was divided into data collection, parsing, tokenization, and dataset generation.

2.1. Data Collection and Parsing

The data corpus was sourced from GitHub.

1. **Repository Search:** The GitHub API was queried to identify the 20 most-starred Python repositories (NUM_REPOS_TO_FETCH = 20). An API key was used and stored in the environment to ensure a high rate limit and security.
2. **File Download:** The script recursively scanned each repository's main branch for all files ending in .py. The raw text content of each file was downloaded and saved locally.
3. **Function Extraction:** The parso library was used to parse the raw Python code. Function definitions (funcdefs) were extracted from each file's Abstract Syntax Tree (AST). To ensure data quality and filter out trivial helper functions, only functions containing more than 20 tokens (MIN_FUNCTION_TOKENS = 20) were retained.
4. **Deduplication:** All extracted functions were collected into a single file (all_functions.txt) and deduplicated, resulting in a large corpus of unique Python functions.

2.2. Tokenizer Training

As per project requirements, a tokenizer was trained from scratch on the collected function corpus.

- **Type:** A Byte-Pair Encoding (BPE) tokenizer was implemented on the training corpus, using the Hugging Face tokenizers library.
- **Training:** The tokenizer was trained on the all_functions.txt file.

- **Configuration:** A vocabulary size of 20,000 (TOKENIZER_VOCAB_SIZE = 20000) was selected. Special tokens, including <pad>, <unk>, and the task-specific mask token <extra_id_0>, were added. The trained tokenizer was saved to python_tokenizer.json.

2.3. Pre-training Dataset

The pre-training dataset was created to teach the model general Python structure using a T5-style span-corruption objective.

- **Objective:** Reconstruct masked spans of code.
- **Method:** For each function, approximately 15% of its tokens were randomly selected for masking. Contiguous spans of masked tokens were replaced by a single unique sentinel token (e.g., <extra_id_0>, <extra_id_1>, etc.).
- **Size:** The target size was 150,000 instances, split into 90% training (pretrain.txt) and 10% validation (pretrain_valid.txt).

2.4. Fine-tuning Dataset

The fine-tuning dataset was built for the specific task of if-condition prediction.

- **Method:** A regular expression (re.compile(r'if\s+(.+?):')) was used to iterate through all if statements in the function corpus. For each match, the conditional expression (group 1 of the regex) was extracted as the **target label**. The original function was modified by replacing this expression with the MASK_TOKEN (<extra_id_0>), which served as the **input**.
- **Size:** The target size was 50,000 instances, split into 80% training, 10% validation, and 10% test.

3. Model Architecture and Training

3.1. Model Architecture

The model is a standard **encoder-decoder Transformer** based on the T5 architecture (T5ForConditionalGeneration). This architecture is well-suited for sequence-to-sequence tasks where the input (ie, masked code) and output (ie, the condition) have different lengths.

A small and standard T5 model was configured:

- **Vocabulary Size:** 20,000 (from the custom tokenizer)
- **Model Dimension (d_model):** 256
- **Feed-Forward Dimension (d_ff):** 1024
- **Layers (num_layers):** 4 (for both encoder and decoder)
- **Attention Heads (num_heads):** 4

3.2. Phase 1: Pre-training

The model was first trained from random initialization on the pre-training (span-corruption) dataset. To enable the model to learn the general syntax, structure, and token relationships of the Python language.

- **Hyperparameters:**
  - **Epochs:** 3 (MAX_TRAIN_EPOCHS_PRETRAIN)
  - **Learning Rate:** 5e-4
  - **Batch Size:** An effective batch size of 32 was achieved using a per-device size of 1 (BATCH_SIZE) and 32 gradient accumulation steps (GRAD_ACCUM).
  - **Optimization:** Mixed precision (fp16=True) was enabled to speed up training.

3.3. Phase 2: Fine-tuning

After pre-training, the model's learned weights were loaded, and training continued on the if-statement dataset. To specialize the model, we adapt its general language understanding to the specific task of predicting 'if' conditions. The Seq2SeqTrainer was used again for fine-tuning.

- **Hyperparameters:**
  - **Epochs:** 5 (MAX_TRAIN_EPOCHS_FINETUNE)
  - **Learning Rate:** A smaller learning rate of 1e-4 was used for stable adaptation.
  - **Batch Size:** The same effective batch size of 32 was maintained.
  - **Strategy:** The trainer was configured to load_best_model_at_end based on validation loss, ensuring the final saved model (finetuned_if_model) had the best performance on unseen data.

4. Evaluation and Results

4.1. Evaluation Methodology

The fine-tuned model's performance was assessed on the 10% held-out test set (finetune_test.txt) and a separate user-provided test set (provided_testset_to_process.csv).

- **Task:** For each item in the test set, the model was given the function with the masked if condition.
- **Generation:** The model generated the predicted condition using **beam search** with 5 beams (num_beams=5) and early_stopping=True.
- **Metric:** The primary metric was **Exact Match (EM) Accuracy**. The generated prediction was stripped of whitespace and compared to the ground-truth target label. A match was only counted if the strings were identical.

4.2. Results and Discussion

The project implemented a complete end-to-end pipeline, from raw data collection to a fine-tuned, predictive model.

- **Accuracy:** The evaluation script computes the final EM accuracy. The final CSV outputs (generated-testset.csv, provided-testset.csv) provide a detailed, case-by-case breakdown of model performance.
- **Discussion:** The two-phase training approach was critical. A model trained only on the 50,000 fine-tuning samples would likely have failed to learn Python's complex syntax. The pre-training phase provided the necessary foundation. Qualitative analysis of the output CSVs shows the model is adept at predicting simple conditions (e.g., x > 0, n % 2 == 0) but struggles with complex, multi-variable boolean logic.
- **Limitations & Future Work:**
    1. **Data Parsing & Tokenization:** The regex-based if statement extraction is simple and fails on multi-line conditions or conditions containing comments. A more robust AST-based approach would be superior.
    2. **Model Scale:** The 4-layer model is very small. Performance would almost certainly increase by scaling the architecture (e.g., d_model=512, 6-8 layers) and training on a larger dataset (e.g., 500+ repositories).
    3. **Data Source:** While popular, the 20 repositories used may not be representative of all Python code. A more diverse selection would improve generalization.