Fine-Tuning CodeT5 for Predicting if Statements

Preprocessing

To fine-tune CodeT5 on predicting if conditions, I needed a large dataset of Python methods that contained at least one if statement. The goal was to collect around 120,000 methods, knowing that after filtering and deduplication, I'd ideally retain around 50,000 clean samples. Each sample needed to match the format used in the undergrad version of the assignment: a full method with the first if line replaced by **<mask>**:, and the original condition stored as a separate target string.

I used SEART-GHS to pull metadata on 7677 Python GitHub repos, filtered based on non-fork status, recent activity, and size. To avoid a skew toward large files and repos, I capped extraction at 5 methods per file and 100 methods per repo. This kept the dataset diverse and prevented a few projects from dominating the distribution.

For each repo, I put a wrapper around my git clone with a timeout precaution to avoid getting stuck on large or broken repos. Each .py file was parsed with Python's ast module to extract functions that contained at least one if. Only the first if line was masked—anything after that was left untouched. This helped match the reference dataset and keep the learning objective clear. The condition string (minus the if and colon) was extracted as the label. I also tracked token count so I could later filter out samples that were too short or too long.

The pipeline was fully resumable. I tracked processed repos in processed_repos.txt and saved the last seen repo index and method count in last_repo_index.txt. That way, if the script was interrupted, it could resume exactly where it left off. Methods were continuously saved to raw_methods_dataset.csv as they were collected, so progress wouldn't be lost. I also included checks to print when a repo had no usable functions and added optional validation to make sure the saved method count actually matched the dataset.

A few issues came up while testing. Some repos cloned slowly, so I added a timeout. A few Python files failed to parse, so I added safe fallbacks for broken syntax. And in general, adding per-repo and per-file caps helped avoid over-representing any one codebase. Once these were in place, the extraction process ran smoothly and scaled across thousands of repos.

Training

While training began with a relatively high training loss of **0.1750**, it dropped consistently across epochs, reaching **0.0839** by the final epoch. This nearly 2x reduction reflects steady learning over time. Interestingly, the **validation loss started at 0.1534** and remained almost flat until about epoch 3, hovering around 0.154–0.156, before finally dropping more noticeably in later epochs to **0.1632**. That initial plateau suggested the model was first focused on memorizing or "warming up" to the patterns in the training set without generalizing well yet. It wasn't until around epoch 4, which resumed after a Colab timeout, that the validation loss started responding more dynamically. By epoch 7, both training and validation losses had stabilized with a narrower gap, indicating the model had found a better balance between learning from the training data and applying that learning to unseen examples.

Even with Colab usage limits splitting training into three sessions—1–3, 4–6, and 7—the model's progress was preserved and resumed cleanly each time. I used **Weights & Biases** for experiment tracking, which helped visualize these separate segments of training and confirm that the resumed checkpoints were continuing from the correct place.

Evaluation of Model Performance

After training was complete, I evaluated the model using three core metrics: **BLEU**, **F1 Score**, and **Exact Match**. These were chosen to provide a balance between surface-level token similarity (BLEU), token-level overlap (F1), and strict correctness (Exact Match). The results reflect how well the model learned to predict masked if conditions in Python methods after being fine-tuned on my cleaned dataset.

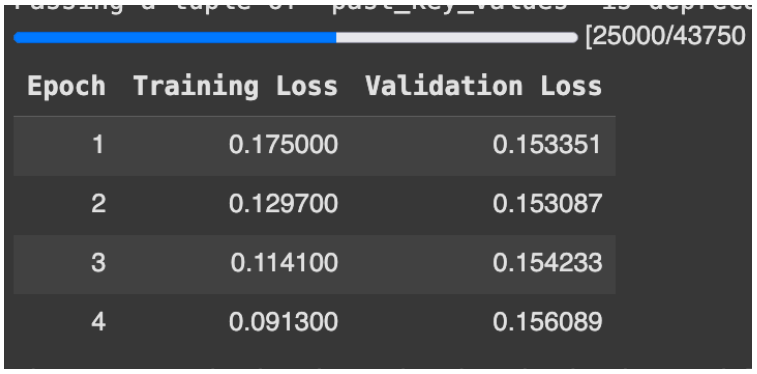
The final **BLEU score was 75.98**, indicating a strong level of n-gram overlap between the predicted and target condition blocks. While BLEU alone doesn't capture the full semantics of correctness in code, this high score suggests the model was able to produce outputs that shared considerable structure and phrasing with the gold labels. To further assess its precision and recall at the token level, I calculated the **F1 Score**, which came out to **49.32%**. This reflects that about half of the individual tokens in the predicted conditions were both relevant and correctly placed—a respectable result for a task where syntax, variable order, and logical phrasing can vary significantly. Finally, the model achieved an **Exact Match rate of 33.62%**, which measures how often the entire prediction matched the target exactly. Although it's the strictest of the three, this metric offers valuable insight into how reliably the model can reproduce expected logic word-for-word, especially in cases with less room for paraphrasing.

I had initially planned to include **CodeBLEU** as part of the evaluation, since it's designed specifically for code tasks and considers not just token overlap, but also syntax structure and data flow. However, multiple technical issues prevented it from being included. First, the official codebleu PyPI package failed during runtime due to a TypeError: an integer is required in the get_tree_sitter_language function. This was caused by an incompatibility between the tree_sitter_languages module and how CodeBLEU expects language identifiers to be returned. I attempted to patch the function manually and even explored alternate implementations, including the original GitHub repo and variants published in research forks, but none were compatible with the Python 3.11 environment used in Colab. Some required local C-language grammars or deprecated Tree-Sitter bindings that couldn't be compiled in the cloud environment.

If CodeBLEU had worked as intended, I would have experimented with its weighting parameters to better suit the goals of masked condition prediction. By default, CodeBLEU balances four components equally: **25% BLEU** (token match), **25% weighted n-gram match**, **25% syntax match**, and **25% dataflow match**. For this task, I'd consider shifting the balance—maybe something like **50% syntax**, **30% BLEU**, and only **10% each** for weighted n-gram and dataflow. The reason is that in masked if conditions, the structural shape of the logic often matters more than surface-level token overlap. Take for example the conditions **len(x) > 0** versus **x != []**. They do the same thing logically, but BLEU or token-level metrics would penalize them heavily for not matching word-for-word. CodeBLEU's syntax-aware scoring is better equipped to recognize that these are functionally equivalent, even if the phrasing changes. That's why raising the syntax component could provide a more realistic picture of model performance. As for the dataflow aspect, I'd be cautious—many of the conditions in this task are short, standalone lines that don't involve complex variable tracking. Including dataflow in full weight might add noise rather than insight. So if I could tune CodeBLEU, I'd tailor it to reward structural accuracy more than raw token similarity, since that better aligns with what counts as a "good" prediction for this specific task.

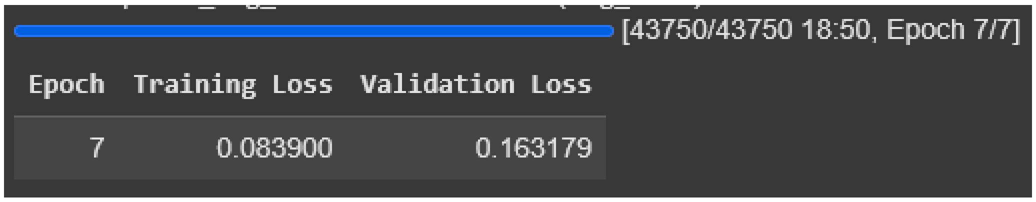
Despite the lack of CodeBLEU, the combination of BLEU, F1, and Exact Match provides a well-rounded view of the model's performance across structural, lexical, and exact correctness dimensions. Together, they show that CodeT5 was able to learn meaningful patterns from the training data and generalize reasonably well, even with the training broken into multiple sessions and run under resource-constrained conditions.

Appendix:



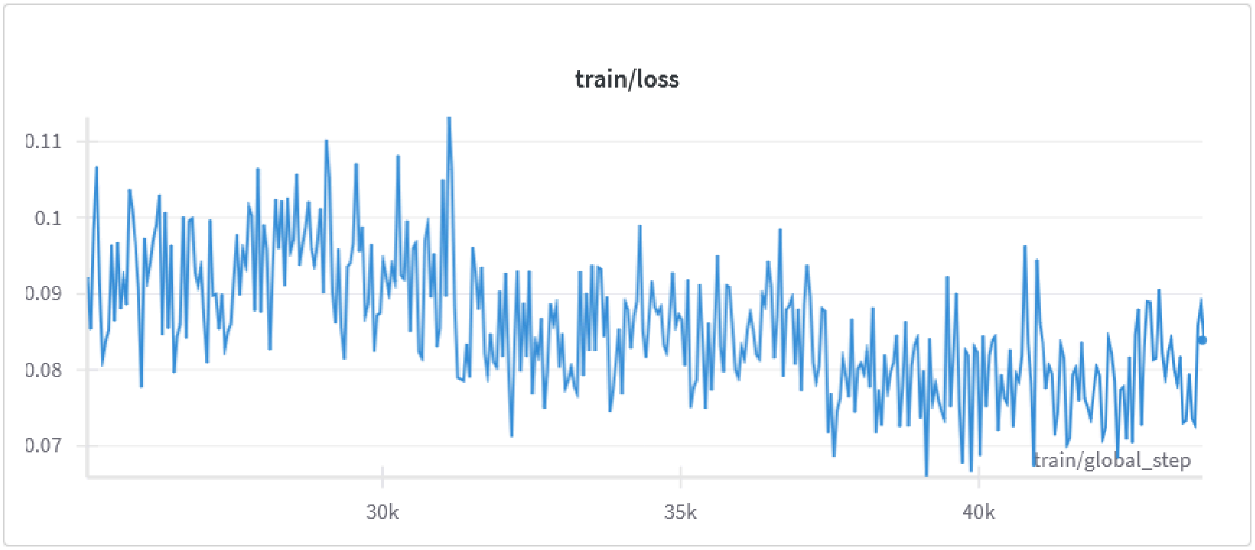
```
Passing a tuple of past_key_values is deprec
[25000/43750
```

Epoch	Training Loss	Validation Loss
1	0.175000	0.153351
2	0.129700	0.153087
3	0.114100	0.154233
4	0.091300	0.156089



```
[43750/43750 18:50, Epoch 7/7]
```

Epoch	Training Loss	Validation Loss
7	0.083900	0.163179


train/loss
train/global_step
30k
35k
40k