Assignment Specification:
This assignment was to fine-tune the CodeT5-small model from HuggingFace to prepare it to predict the missing If-statements from masked Python functions.

Workflow:
For code where a relevant understanding of the theory or mechanics behind deep learning was needed, I reviewed prior labs and class slides as well as documentation for relevant packages. In particular I gained a better understanding of pandas dataframes, the Roberta Tokenizer, epochs, training and validation loss, and the evaluation metrics. I wrote code to perform the tasks, and to debug and clean the code I used Gemini and ChatGPT. (An observation is that ChatGPT was in general more helpful than Gemini, which sometimes misdiagnosed my issue or situation). For mundane tasks such as reading and writing files, I had ChatGPT generate code. The project was hosted in Google Colab with the A100 GPU.

Dataset Preparation:
I edited section 2 of the providedCodeT5 Jupyter notebook to load the provided testing, training, and validation datasets. The model was CodeT5 and the tokenizer was the given Roberta Tokenizer.

I changed section 4 heavily. I added a method flatten_mask_tabize to flatten, mask, and add the <TAB> token for indentation into the data. To mask the data, I tokenized the flattened and tabized methods, then replaced the first instance of the tokenized if statement in the given method that matches the tokenized target if statement. I had to implement a method to compare tokens, based on two observations: (i) the spacing in the cleaned_method and target_block columns was not consistent, (ii) Roberta adds a special character to the head of tokens that follow a space. My compare_tokens method determines whether two tokens are the same after potentially removing this special character. With ChatGPT's help I processed the flattened, masked, and tabized methods into a format that the trainer could read by adding padding and converting the data to have suitable type and organization.

Fine-tuning:
Preliminaries/Restrictions. I trained the model several times using very small percentages of the datasets to debug and improve my evaluation section. When training using the full datasets I would often run out of compute limits in Google Colab. I attempted to use the Adafactor optimizer to decrease the training time, but when I began training the estimated time until completion was similar as without the optimizer.

Retraining/Epochs. I trained the model using the full dataset twice. I will primarily discuss the settings I used for the second training. An epoch is a full pass of the training set through the model. The number of epochs used the first time was 3 and the second time was 4 - I made minimal other changes between the trainer besides this, yet my metrics went up from the .55-.65 range to the .70-.75 range. This absolutely impresses upon me the importance of many epochs. Since the range of my metrics decreased, I extrapolate the hypothesis that as more epochs are

used, the rate of increase in metrics decreases. I could not use more than 4 epochs due to limits in computing allocation by Google Colab.

Other fine-tuning. The learning rate, which adjusts how much the model changes its hyperparameters at each step of gradient descent, was kept to the default of 5e-5. The weight_decay parameter was kept at .01 to penalize large weights in order to prevent overfitting. For the training and eval sets, the batch size per device was set to 2. This is how many data points the model sees at once before updating weights. This was done to reduce memory usage, but the number was kept small to try to maximize the effect of each data point on the model. Gradient accumulation steps, the number of times backpropagation happens before updating weights, was set to 3 to try to reduce memory usage as well.

Output:
As per the project specification, a csv file named testset-results.csv was generated when checking the model against the test set. It includes columns for the masked method, the correct if-statement, the predicted if-statement, whether the correct and predicted if-statements match (the exact match boolean), the BLEU-4 prediction score, and the CodeBLEU prediction score. A file bleuresults.txt is also generated to save the metrics for the entire test set.

Results:
The F1 score is the percent of the time that the model computed the exact correct response on the testing data. The score was, 0.7407, meaning that 5000*.7407 = 3704 responses were exactly the masked If statement. Please note that the F1 score does not write to the bleuresults.txt file like the other metrics, but running the notebook will print it.

The BLEU score (ngram-match) was 0.7253817006576274. This measures the percentage of n-grams (n=1 to 4) made from the masked and model's if statements match. Being a metric based purely on the characters that appear and not their meaning, this score begs the question of how to incorporate the meaning of code (or language) into evaluation, leading to the CodeBLEU score.

The weighted ngram match was 0.729027673186805. This is the above, but weighting special relevant tokens such as if and return more heavily.

The syntax match was 0.7073335130763009. This computes the abstract syntax trees of expected and predicted if statements to try to grasp whether the model predicts a response that is syntactically consistent with the expected response.

The Dataflow match was dataflow_match: 0.7406679764243614. This is the best of the four metrics used to compute the CodeBLEU score. This metric checks whether the logic statements flow the same way in the predicted and masked if statements.

The CodeBLEU score was calculated using equal weights, being .25 times each of the four above metrics. It is a better metric for code-code machine learning tasks than the F1 score because it considers various aspects of the correctness of the model. The CodeBLEU score was 0.7256027158362737