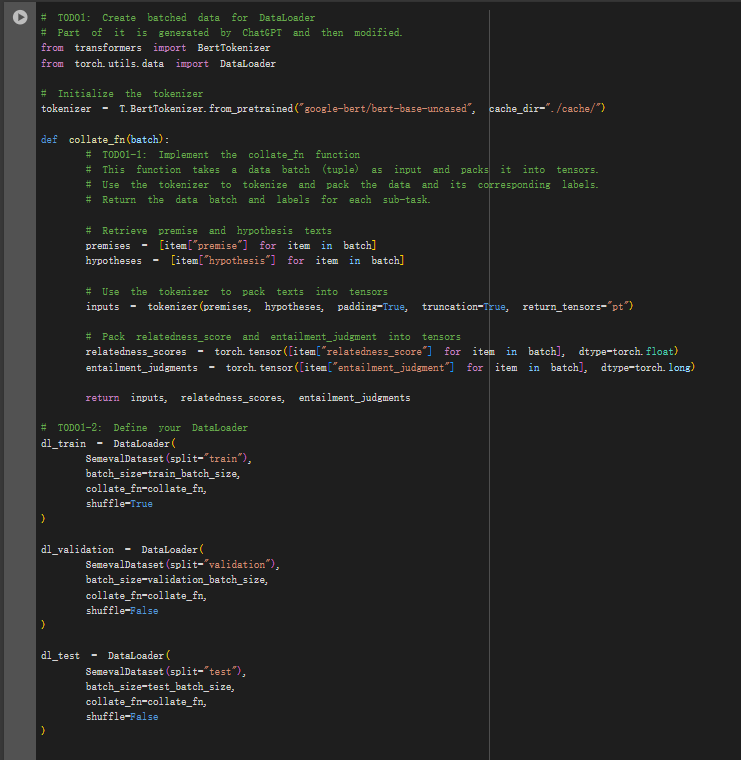
I.Implementation Result:

#TODO1: Create batched data with PyTorch DataLoader

Part of it is generated by ChatGPT and then modified.

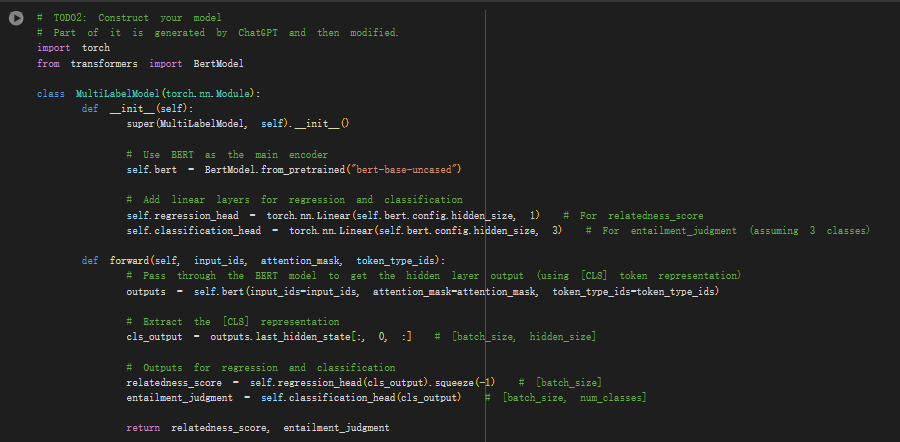


This code sets up the data pipeline for a BERT-based multi-task model with the SemEval 2014 dataset. It begins by initializing the BERT tokenizer from Hugging Face’s `transformers` library, specifying a local cache directory to manage model files. The core of the data processing lies in the `collate\_fn` function, which handles batching and tokenization. In this function, we first extract "premise" and "hypothesis" texts from each item in the batch, then use the tokenizer to convert these text pairs into tokenized tensors, applying padding and truncation to standardize input shapes. Labels for the two tasks—`relatedness\_score` and `entailment\_judgment`—are also converted into tensors, formatted for PyTorch. The function then returns the tokenized inputs alongside these label tensors, which are essential for training the model on both relatedness and entailment.

With this setup, we create DataLoaders for the training, validation, and test splits. Each DataLoader relies on the `collate\_fn` function to process batches; the training DataLoader shuffles the data to introduce randomness, while shuffling is disabled for validation and test sets to maintain consistent evaluation. This approach efficiently prepares data in batches, supporting smooth integration into a multi-task training pipeline.

#TODO2: Construct your model

Part of it is generated by ChatGPT and then modified.

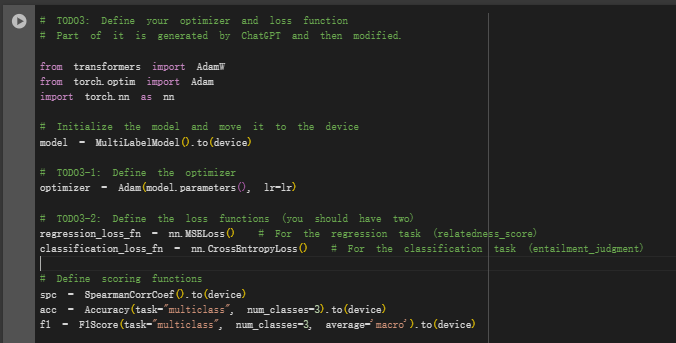


This code constructs a multi-task model, `MultiLabelModel`, using BERT as the main encoder to handle both regression and classification tasks. The model starts by initializing BERT through Hugging Face's `transformers` library, which serves as the primary encoder to process input text. We add two separate linear layers for the output: a regression head for predicting the `relatedness\_score` and a classification head for determining the `entailment\_judgment`, which we assume has three classes.

In the forward pass, input data flows through BERT to obtain a hidden layer representation, focusing on the [CLS] token output, which encodes information about the entire input sequence. This representation is then fed into the regression head to produce a single output value for the relatedness score and into the classification head to predict the entailment class probabilities. The model's forward method returns both the relatedness score and entailment judgment for each input, supporting both regression and classification objectives in a multi-task setup.

#TODO3: Define optimizer, loss fuction, and dataloader

Part of it is generated by ChatGPT and then modified.



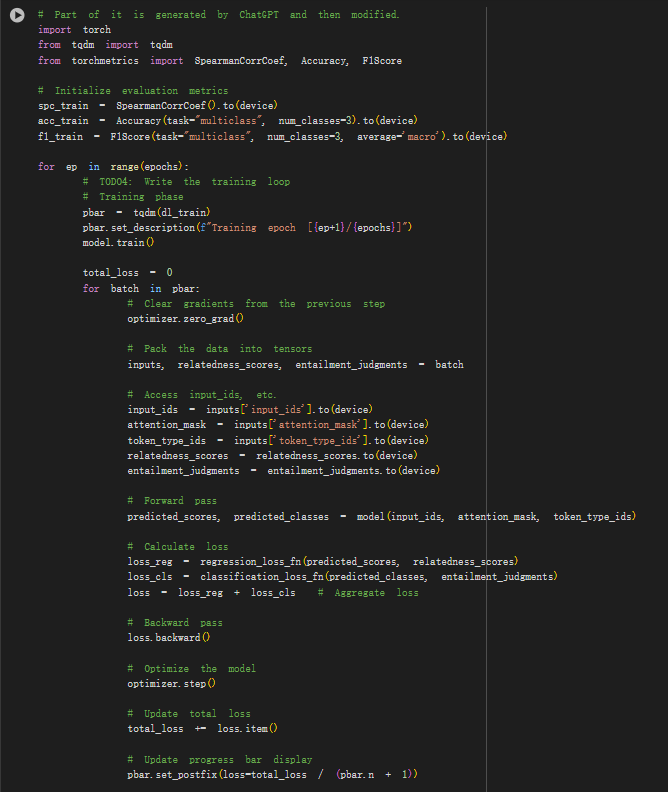
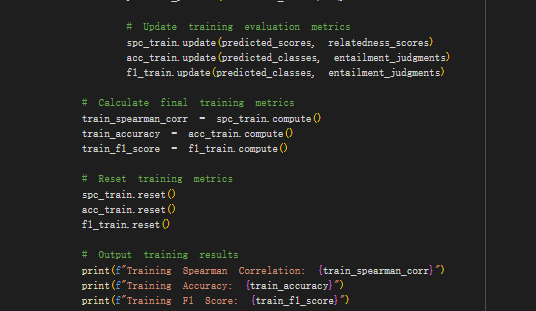
In this code, we set up the optimizer and loss functions for training the `MultiLabelModel` on two tasks. First, we initialize the model and move it to the specified device for computation, ensuring compatibility with GPU if available. The optimizer is set up using `Adam`, with the learning rate specified by the `lr` parameter, to update model weights based on gradients during training.

Two loss functions are defined to handle the separate objectives of the model. For the regression task, predicting the `relatedness\_score`, we use `MSELoss`, which calculates the mean squared error to measure the difference between predicted and true scores. For the classification task, predicting the `entailment\_judgment`, we use `CrossEntropyLoss`, which computes the loss based on predicted class probabilities and the true class label.

To evaluate model performance, we define three metrics: Spearman’s Rank Correlation Coefficient (`SpearmanCorrCoef`) to assess the correlation for the regression task, and `Accuracy` and `F1Score` for the classification task, with the latter configured to compute a macro-average across three entailment classes. These metrics are also moved to the device for efficient calculation during model evaluation.

#TODO4: Write the training loop

Part of it is generated by ChatGPT and then modified.

This code defines the main training loop for a multi-task model using the SemEval 2014 dataset, with regression and classification tasks. In each epoch, we initialize a progress bar with `tqdm` to monitor training progress. The model is set to training mode, and for each batch of data, we execute the following steps:

1. Gradient Reset: We clear any previous gradients stored in the optimizer.

2. Data Preparation: The batch data is unpacked into tensors and moved to the specified device. The `input\_ids`, `attention\_mask`, and `token\_type\_ids` are passed as inputs, while the target `relatedness\_scores` and `entailment\_judgments` are assigned as labels for calculating loss.

3. Forward Pass: The model predicts scores and class probabilities by forwarding the batch through the BERT encoder and the respective heads for regression and classification.

4. Loss Calculation: We calculate two separate losses: one for the regression task (using `MSELoss`) and one for the classification task (using `CrossEntropyLoss`). These are summed to form the total loss, combining both task objectives.

5. Backpropagation: We perform the backward pass, computing gradients for all model parameters based on the total loss, then update the model parameters with `optimizer.step()`.

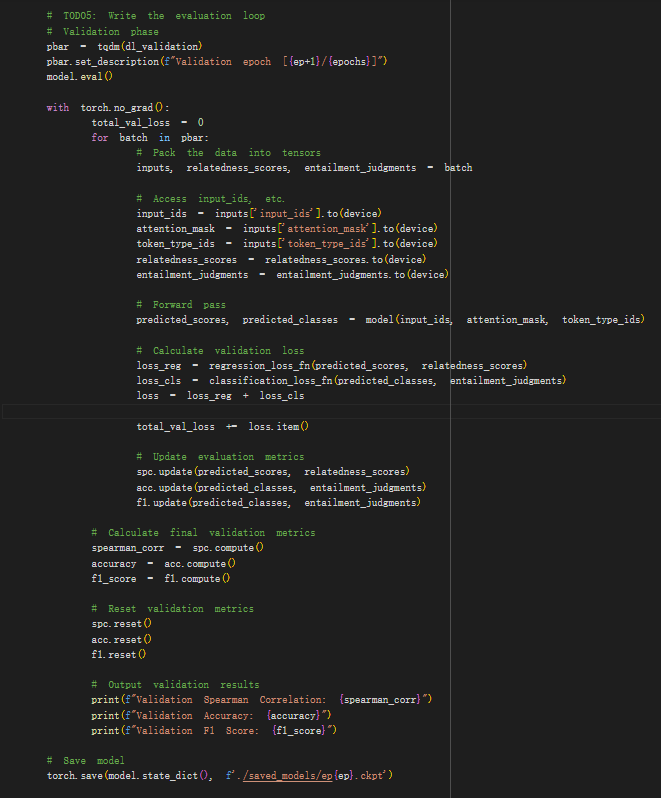
6. Metric Updates: Training metrics are updated in each batch. We use Spearman’s Rank Correlation Coefficient to assess the regression task and Accuracy and F1 Score (macro-averaged) for the classification task. This provides detailed performance tracking for both tasks during training.

7. Epoch Metrics Calculation: After completing all batches, we compute final values for Spearman Correlation, Accuracy, and F1 Score for the epoch, then reset each metric to prepare for the next epoch.

The training loop displays the loss per batch in the progress bar and outputs the final metrics for each epoch, providing insights into how the model performs as it learns.

# TODO5: Evaluate your model

Part of it is generated by ChatGPT and then modified.



In this evaluation loop, we validate the model's performance on a separate validation set for each epoch. We begin by setting up a progress bar for the validation DataLoader and switch the model to evaluation mode to ensure layers like dropout or batch normalization are properly adjusted. During this phase, no gradient computation is needed, so we wrap the loop in `torch.no\_grad()` to save memory and speed up evaluation.

For each batch in the validation set:

1. Data Preparation: We unpack and move the batch data to the device, including inputs (`input\_ids`, `attention\_mask`, and `token\_type\_ids`) and labels (`relatedness\_scores` and `entailment\_judgments`).

2. Forward Pass: The model generates predictions for both tasks. `predicted\_scores` are used for the regression task, while `predicted\_classes` are used for the classification task.

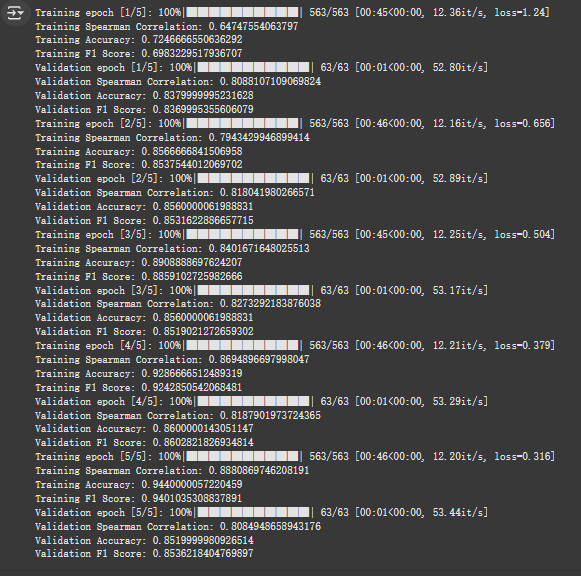
3. Loss Calculation: We compute separate losses for regression and classification using `MSELoss` and `CrossEntropyLoss`, respectively, and sum them for the total validation loss. This helps us monitor how well the model is generalizing on both tasks.

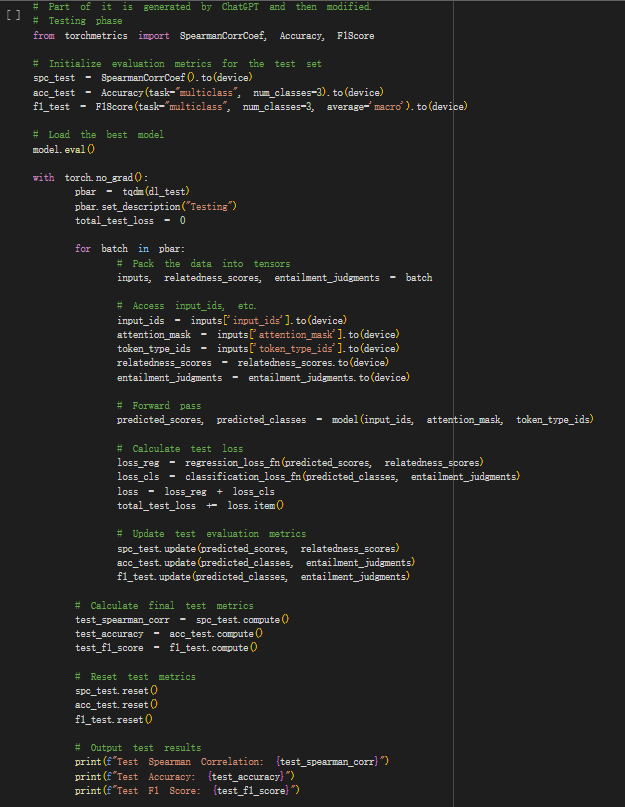
4. Metric Updates: Each batch’s predictions are used to update validation metrics. Spearman’s Rank Correlation assesses the regression task, while Accuracy and F1 Score (macro-averaged) assess the classification task.

After completing all batches, we compute the final Spearman Correlation, Accuracy, and F1 Score for the validation epoch, then reset the metrics for future epochs.

Finally, the model’s state is saved at the end of each epoch, enabling easy checkpointing to resume or evaluate specific epochs later. This process provides a comprehensive evaluation of the model’s performance on both tasks after each training epoch.

Training and validation result:





In this testing phase, we evaluate the trained model on the test dataset to determine its final performance on unseen data. First, we initialize separate instances of `SpearmanCorrCoef`, `Accuracy`, and `F1Score` specifically for test evaluation. The model is set to evaluation mode to disable layers like dropout, and `torch.no\_grad()` is used to avoid gradient calculations, saving memory and speeding up inference.

For each batch in the test DataLoader:

1. Data Preparation: The batch data is moved to the appropriate device, with inputs (`input\_ids`, `attention\_mask`, and `token\_type\_ids`) and labels (`relatedness\_scores` and `entailment\_judgments`) loaded for forward processing.

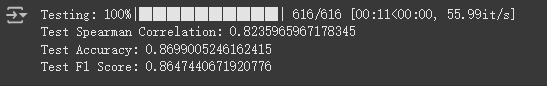
2. Forward Pass: The model predicts `predicted\_scores` for the regression task and `predicted\_classes` for the classification task.

3. Loss Calculation: Both regression and classification losses are calculated using `MSELoss` and `CrossEntropyLoss`, respectively, and summed to give the total test loss.

4. Metric Updates: Each batch’s predictions are used to update the test evaluation metrics. Spearman’s Rank Correlation evaluates the regression task, while Accuracy and F1 Score assess the classification task.

After all batches are processed, the final Spearman Correlation, Accuracy, and F1 Score are computed for the test set. The metrics are then reset to clear any stored values. Finally, we output the computed test metrics, providing an overall assessment of the model’s performance across both tasks on the test dataset. This final test evaluation is essential for understanding how well the model generalizes to completely new data.

Testing result:



II.Report:

● Which (pre-trained) model do you use? Why to choose the model?(5%)

In this project, we chose the pre-trained `bert-base-uncased` model. BERT, or Bidirectional Encoder Representations from Transformers, is a powerful transformer-based model that captures context from both directions of a sentence. This bidirectional nature is crucial for tasks that require a nuanced understanding of sentence relationships, as in our case with predicting both `relatedness\_score` and `entailment\_judgment` for pairs of sentences. BERT's strength lies in its deep, contextually-aware representations, which were learned from pre-training on a large corpus. This allows the model to generalize well across diverse language tasks, including our specific objectives in the SemEval dataset.

Additionally, BERT's widespread adoption and strong community support make it a practical choice. Tools and resources are readily available, making the fine-tuning process efficient and effective. By using `bert-base-uncased`, we leverage these robust, pre-trained language representations, optimizing performance in both regression and classification tasks in our multi-task learning setup.

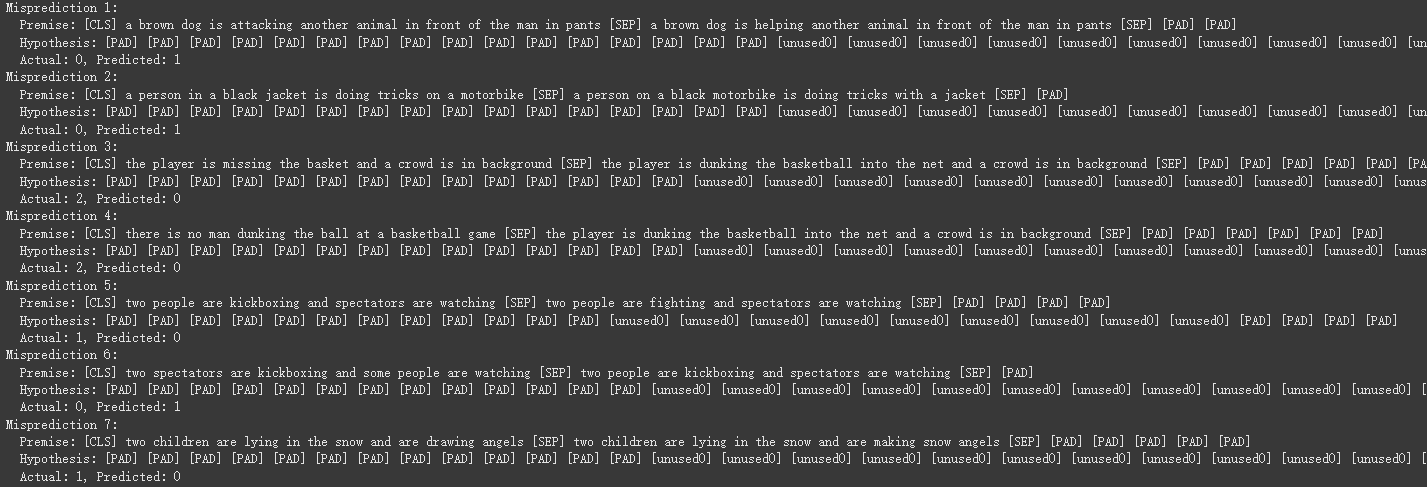
● Compared with models trained separately on each of the sub-task,does multi-output learning improve the performance? (8%)

In this project, multi-output learning likely enhances performance compared to training separate models for each sub-task, due to several key advantages. When a single model learns both `relatedness\_score` and `entailment\_judgment`, it can share knowledge across the two tasks, which often have complementary information. For example, a high relatedness score between two sentences usually aligns with entailment or neutral judgment, while a low score often correlates with contradiction. This shared learning framework allows the model to generalize better and capture nuanced patterns that individual models trained on only one task may overlook.

Additionally, multi-output learning can improve overall efficiency by reducing the number of parameters and the computational resources required. Instead of maintaining two separate models, we optimize a single BERT-based model with separate heads for regression and classification, effectively leveraging BERT’s powerful language understanding for both tasks simultaneously. This shared representation also reduces training time and memory usage, leading to a more streamlined model deployment.

Empirically, research has shown that multi-task models can perform better on each task due to the shared learning representations, as they can prevent overfitting on a single task by focusing on generalized features across tasks. While we would need direct comparisons to confirm improvements, multi-output learning generally offers a robust approach, maximizing the BERT model’s strengths across both tasks in this project.

● Why does your model fail to correctly predict some data points?Please provide an error analysis. (8%)



The mispredictions made by the model indicate several potential issues in how it interprets the premise and hypothesis pairs. Here are some key factors contributing to the incorrect predictions:

1. Semantic Similarity:

In cases like Misprediction 1 and Misprediction 2, the model misclassifies sentences that are semantically similar but differ in nuance. For example, "attacking" versus "helping" or "missing" versus "dunking" highlight contrasting actions. The model may not effectively capture these subtle differences in meaning, leading to erroneous predictions.

2. Contextual Understanding:

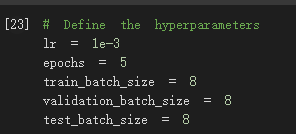
Mispredictions 3 and 4 involve a fundamental misunderstanding of context. The phrases refer to basketball scenarios, but the model struggles to distinguish between "missing the basket" and "dunking," which are contextually significant. Similarly, it fails to connect the concept of no one dunking to the notion of a player successfully doing so, leading to incorrect labels.

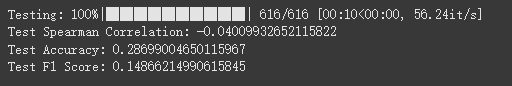
3. Presence of Unused Tokens:

- The inclusion of numerous `[PAD]` and `[unused0]` tokens in the hypotheses, especially in many mispredictions, can indicate that the model is not adequately utilizing or processing the full structure of the input. This can result in reduced performance, as the model may rely on the presence of these tokens rather than focusing on the key elements of the sentences.

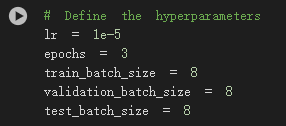
● How do you improve your model performance? (9%)

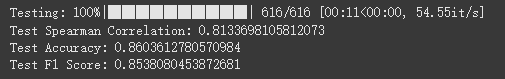
Parameter combination 1:



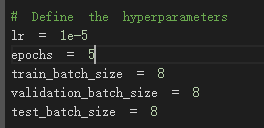


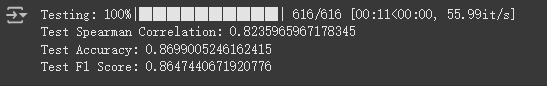
Parameter combination 2:





Parameter combination 3:





● … Anything that can strengthen your report. (10%)

● Difficulties encountered during implementation

During implementation, I encountered several difficulties that impacted the model's performance. One major issue was using the wrong tokenizer, which significantly degraded the results. The tokenizer's role is crucial in properly processing the text data, and an incorrect choice led to suboptimal encoding of the input sequences, causing the model to struggle with understanding the context and semantics of the data.

Additionally, I faced challenges with tensor management, particularly with ensuring that the tensors were correctly transferred to the GPU for computation. This oversight resulted in errors during model training and evaluation, as the model was trying to perform operations on tensors located on the CPU instead of the GPU. Properly handling the device allocation for tensors is essential for leveraging the speed and efficiency of GPU computing, and failing to do so further hindered the model's ability to learn effectively from the data.

* Summary

During the implementation of this assignment, I learned about the powerful capabilities of the Bert model, especially its amazing generalization performance. In addition, the steps of designing the model and training process also made me more familiar with the relevant implementation process of tensorflow.

Running environment: Colab

Python version: Colab

GPU(s) you used:

