Assignment 6

**Task 1:**

**i ) Inputs and Outputs of the Tokenizer:**  
**Input**: The tokenizer takes raw text, such as individual sentences or entire documents, as its input. In this code, train\_texts and test\_texts having tweets from the dataset as the inputs.  
**Output**: The tokenizer's primary output is a dictionary-like object containing numerical representations of the input text. This includes:  
1. input\_ids: A list of integer IDs, where each ID corresponds to a token in the tokenizer's vocabulary.  
2. attention\_mask: A list of integers (0s and 1s) indicating which tokens should be attended to by the model. Padding tokens are usually masked with 0.

**ii)** **What the Tokenizer Function Does:** The tokenizer() function performs several crucial steps:  
1. Tokenization: Apply WordPiece tokenization to capture parts of the morphological structure within tokens.  
2. Vocabulary Mapping: Each token is then mapped to a unique integer ID based on the tokenizer's vocabulary, to standardize the input so that BERT knows exactly what each token represents.  
3. Adding Special Tokens: BERT appends special tokens to understand the input structure. These include:   
 a) [CLS] (classification token): Added at the beginning of each sequence.  
 b) [SEP] (separator token): Added at the end of each sequence.

4. Handles padding and truncation to ensure all inputs are of uniform length (up to max\_length, here 512). In detail meaning is provided in the next response.

A chart of different colored squares

AI-generated content may be incorrect.

(taken from class material)

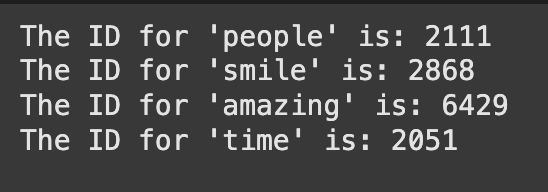
**iii) What do “truncation” and “padding” mean in the tokenizer?**  
**Truncation**: This is the process of shortening a sequence of tokens that exceeds a predefined maximum length (512 in our case). When truncation=True, the tokenizer will cut off tokens until it reaches the max\_length. This is important because BERT model has a maximum input sequence length they can process.  
**Padding**: This is the process of adding special "padding" tokens (usually represented by [PAD] and having an ID of 0) to the end of a sequence that is shorter than a predefined maximum length. When padding=True, the tokenizer will add these padding tokens until the sequence reaches the max\_length. This ensures that all input sequences in a batch have the same length, which is required for efficient processing by the model.

**iv) What is the vocabulary size of the tokenizer?**The vocabulary size of the bert-base-uncased tokenizer is 30,522. This includes individual words, subwords, and special tokens.

**v) What is the difference between the tokenizers used in CNN and in BERT?**CNNs used simpler tokenization methods. Our approach was to split text based on whitespace and punctuation, creating a vocabulary of individual words. Less sophisticated handling of out-of-vocabulary words and no inherent understanding of sub-word relationships are typical.

BERT utilized a more advanced technique called WordPiece tokenization. This allows it to break down words into smaller, meaningful sub-word units. This approach helps in several ways:  
a. Handling Out-of-Vocabulary Words: By breaking down unknown words into known subwords, BERT can still process them to some extent.  
b. Capturing Sub-word Meaning: It can recognize that "running," "runs," and "ran" share a common root ("run") and represent this through shared sub-word tokens.  
c. Efficiency: It can represent a large vocabulary more efficiently.  
Furthermore, BERT's tokenizer also adds special tokens like [CLS] and [SEP] which are crucial for the model's understanding of sequence relationships and classification tasks, something not typically done in tokenization for basic CNN models in NLP.

vi) Add code to print out the ids of the following words in the final vocabulary: "people", "smile", "amazing", and "time"



**Task 2:**i) What do TensorDataset and DataLoader do?TensorDataset: This is a PyTorch class that wraps tensors (like NumPy arrays but on PyTorch's computational graph) into a dataset-like object. It allows easy access to features and labels of the data using indexing. In short it is a way to organize input data and target labels into a structure that PyTorch can understand and iterate over.

DataLoader: This class takes our TensorDataset and provides an iterable over it. It applies the process of:  
a. Batching: Grouping the data samples into batches of a specified size. We have taken it as 8 for both train and test tensors.  
b. Shuffling: Randomly shuffling the order of the data samples in each epoch to prevent the model  
from learning patterns related to our data order. We have applied shuffling to our train tensor.  
**ii) What are the inputs and outputs of TensorDataset and DataLoader?**  
**a. TensorDataset:**  
Input:It takes three tensors: input\_ids, and attention\_mask from the encodings generated through tokenization, and encoded\_labels, which are just labelencoded intensity class.  
Output: It returns a TensorDataset object. It returns a tuple containing the input IDs, attention mask, and encoded label for the whole train and test sample. Basically datasets that can be used by the BERT model.  
**b. DataLoader:**Input: It takes a Dataset object ( train\_dataset and test\_dataset), a batch\_size, and a boolean shuffle parameter.  
Output: It returns an iterable. When iterated over this iterable (e.g., using a for loop), it yields batches of data. Each batch is typically a tuple of tensors, where each tensor in the tuple contains a batch of the corresponding data (e.g., a batch of input\_ids, a batch of attention\_mask, and a batch of encoded\_labels). The size of each tensor in the batch will be equal to the batch\_size (unless it's the last batch and the total number of samples is not perfectly divisible by the batch\_size).

**iii) What do “input\_ids” and “attention\_mask” of train\_encodings/test\_encodings denote?**

1. input\_ids: This is a tensor containing the numerical IDs of the tokens in each input sequence (tweet). Each word or subword in the tokenized text is mapped to an integer based on the tokenizer's vocabulary. These IDs are the primary input to the BERT model.
2. attention\_mask: This is a tensor (or a list of lists) of the same shape as input\_ids. It contains binary values (0s and 1s). A value of 1 indicates that the corresponding token in input\_ids should be attended to by the model, while a value of 0 indicates that the corresponding token is a padding token and should be ignored by the model's attention mechanism. This is crucial because we padded sequences to have the same length, and the model needs to know which parts of the input are actual content and which are just padding.

**iv) What does “batch\_size” mean?**  
batch\_size is an integer that defines the number of data samples that will be processed together in one forward and backward pass during training. Instead of feeding the entire dataset to the model at once, which can be computationally expensive and memory-intensive, we divide the data into smaller chunks (batches). The model updates its weights after processing each batch. A batch\_size of 8 means that the DataLoader will provide 8 samples at a time to the model.

**v) What does “shuffle” mean? Why set “shuffle” of train\_loader as True while set “shuffle” of test\_loader as False?**  
shuffle: This is a boolean parameter that determines whether the order of the data samples in the dataset is randomized before each epoch (iteration over the entire dataset).  
shuffle=True for train\_loader is done to reduce correlation between batches. By shuffling the data, we ensure that consecutive batches during training are less likely to contain similar samples. This can help the model learn more robust and generalizable features. Also we can prevent the model from learning the order of the data. If the training data is not shuffled, the model might inadvertently learn patterns related to the order in which the samples appear, which would not be useful for unseen data.  
shuffle=False for test\_loader: We want to evaluate the model on the test set in the same order each time to ensure that our evaluation metrics are consistent and comparable across different epoch runs or model versions.

**Task 3:  
i) What does “AdamW” do?**It updates the model’s parameters (weights) during training to minimize the loss. Just like we used Adam in our last assignment.

**ii) What is a scheduler?**  
A scheduler is a mechanism that adjusts the learning rate during training to improve performance and convergence. Instead of keeping the learning rate constant, a scheduler modifies it dynamically at different stages of training.

**iii)** **What does “num\_warmup\_steps=100” mean?**num\_warmup\_steps specifies the number of training steps at the beginning of training during which the learning rate will be gradually increased from an initial small value to the initial learning rate set in the optimizer (5e-5, in our case).We havenum\_warmup\_steps=100 which means that there will be 100 training steps before we get to actual learning rate.

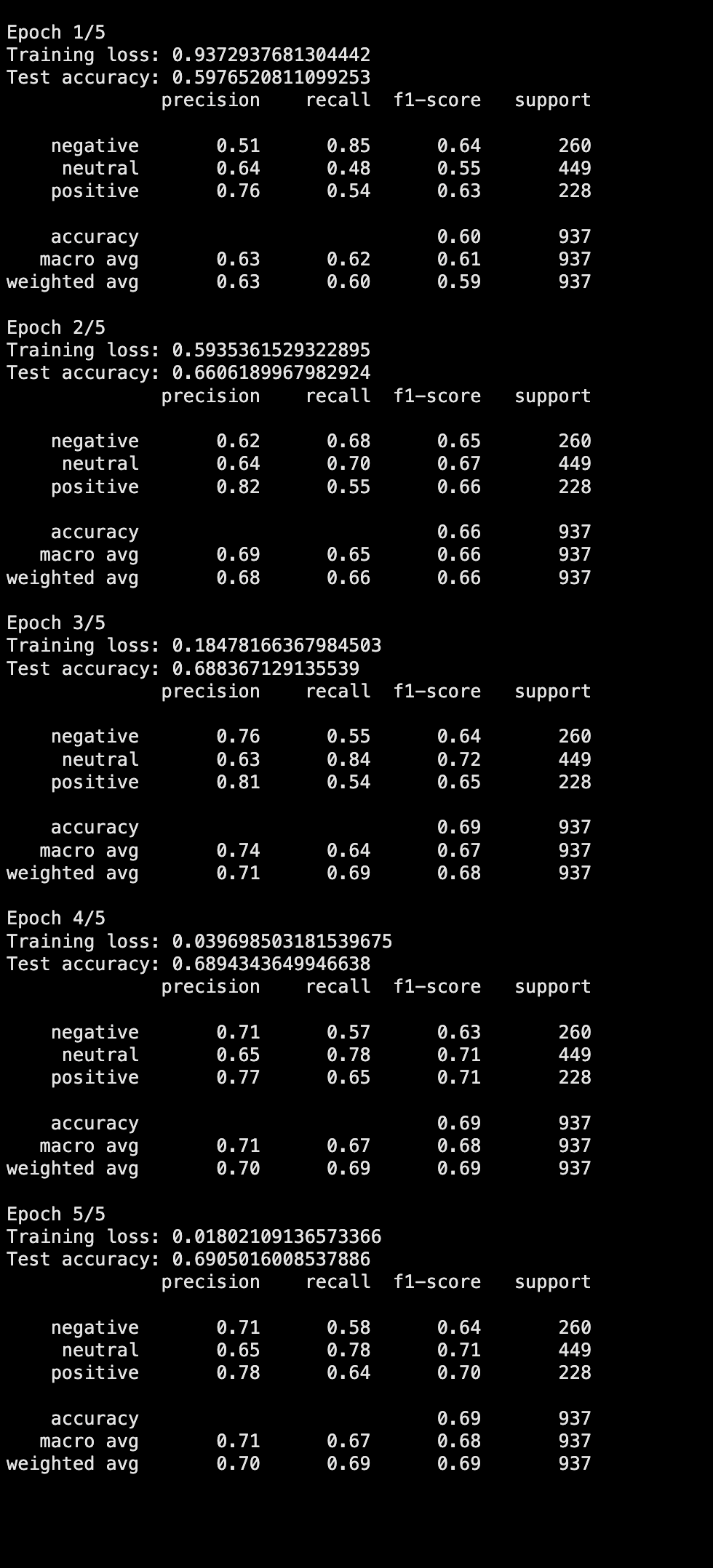
**Will the learning rate set with the optimizer affect the trained model performance?**Yes, the learning rate is one of the most critical hyperparameters in training models, and the learning rate set in the optimizer directly and significantly affects the trained model's performance.A learning rate that is too high can cause the training process to be unstable, with the loss oscillating or even diverging. Whereasa learning rate that is too low can make the training process very slow, potentially taking a very long time to converge.

**Task 4:**  
**i) What are the inputs and outputs of the BertForSequenceClassification forward function? How does BertForSequenceClassification calculate the loss?**  
a. input\_ids: Indices of input sequence tokens in the vocabulary. These are the numerical representations of training Tweet text.  
attention\_mask: Mask to avoid performing attention on padding token indices. Mask values selected in [0, 1]: 1 for tokens that are NOT masked, 0 for MASKED tokens, padding in this case.     
labels: Labels for computing the sequence classification loss. Indices should be in [0, 2] as we have three classification labels.

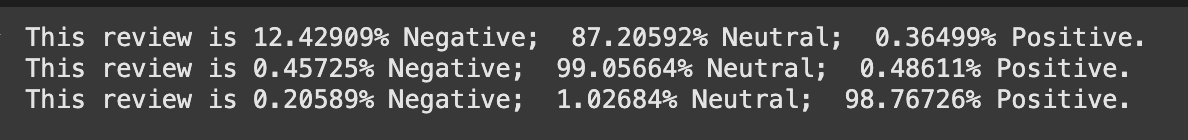
The forward function produces the following outputs:  
loss: Classification loss, how far was the prediction from actual value.  
logits: Classification scores. These are the raw, unnormalized predictions of the model for each class based on which loss will be calculated.

b. How BertForSequenceClassification calculates the loss:  
Labels are provided to the forward function; the model calculates the loss using the Cross-Entropy Loss (torch.nn.CrossEntropyLoss) function under the hood.

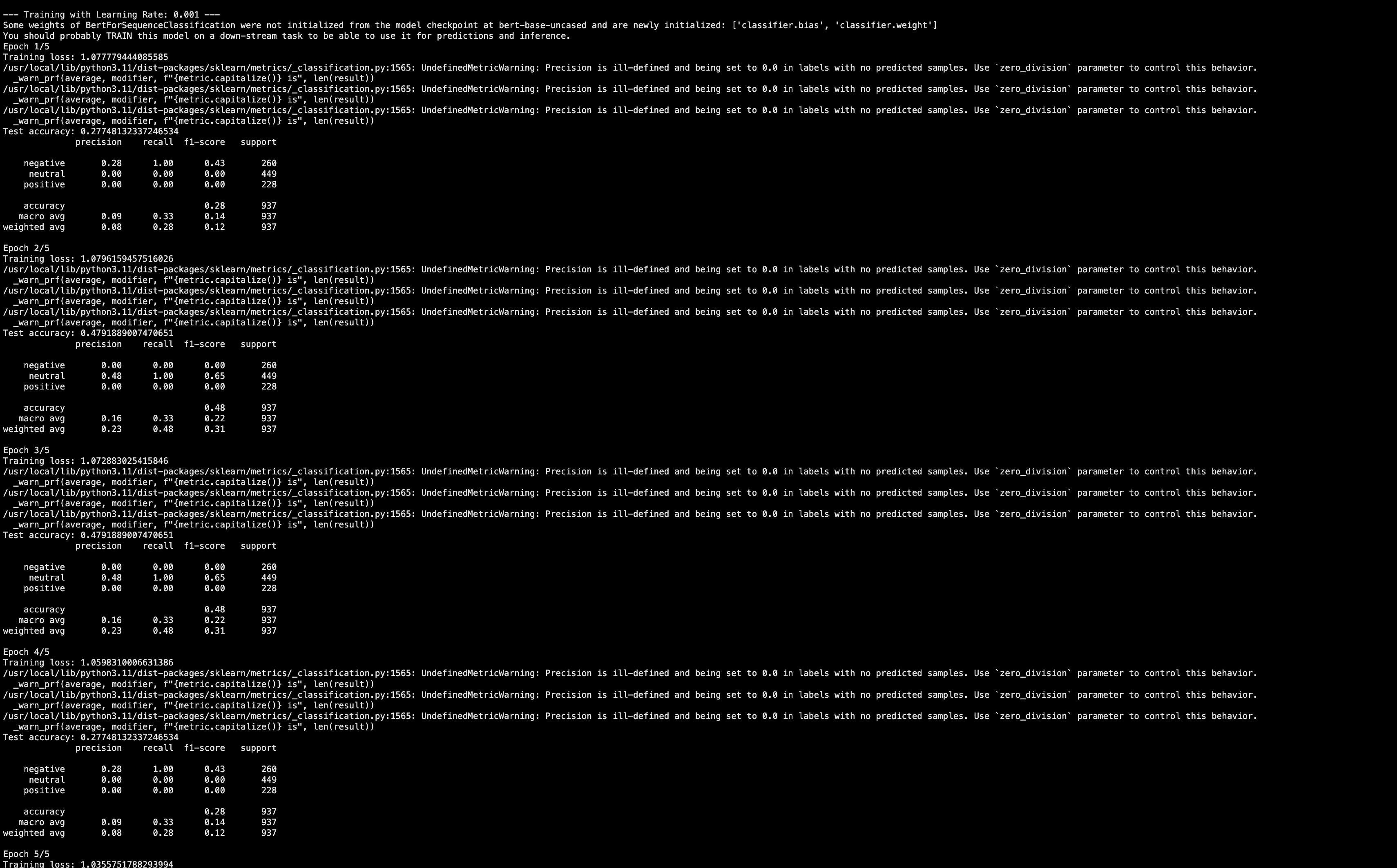
**ii) Run the code and interpret the results.**   
a. Best Test Macro F1: 0.68 at Epoch 4 and 5.  
b. As the training loss keeps decreasing, the test F1 score improves — but only up to 3rd epoch. After Epoch 3, the test F1 score stops improving significantly, even though training loss keeps falling sharply. Which could mean overfitting could start happening despite training longer.

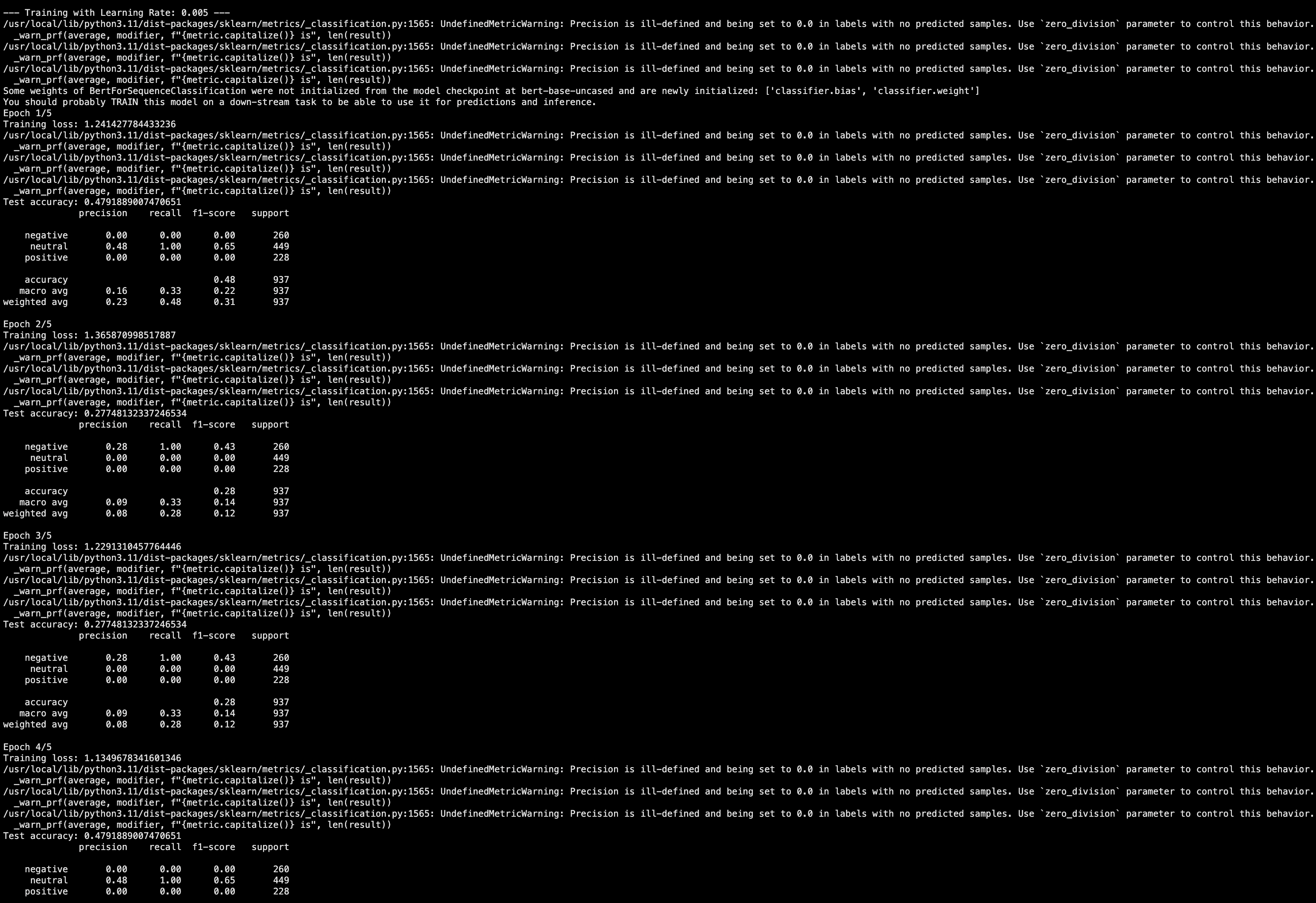


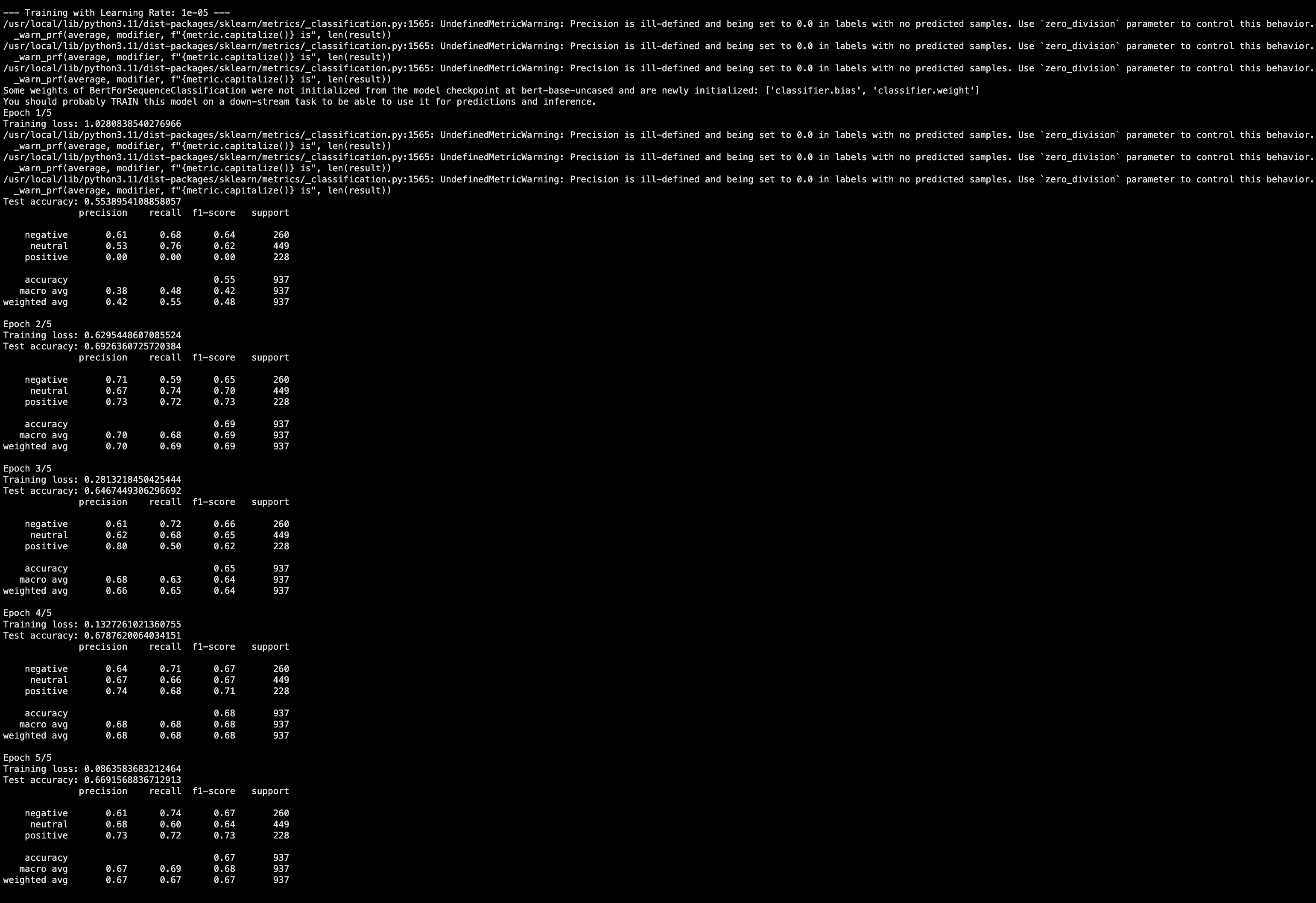
**Task 5:**i) First, the model returns logits (raw scores) for each class (Negative, Neutral, Positive). After that It applies torch.softmax, which converts logits into probabilities (values between 0 and at last we can select the highest probability as the predicted class (using torch.argmax). We are displaying the probability for each class.



ii) Since BERT was trained with a maximum sequence length of 512, we shouldn't alter max\_len. Max\_len can be set to a lower value, if necessary, but it cannot be increased past 512 without retraining the model, and lowering it could result in the loss of crucial data.

**Task 6:**





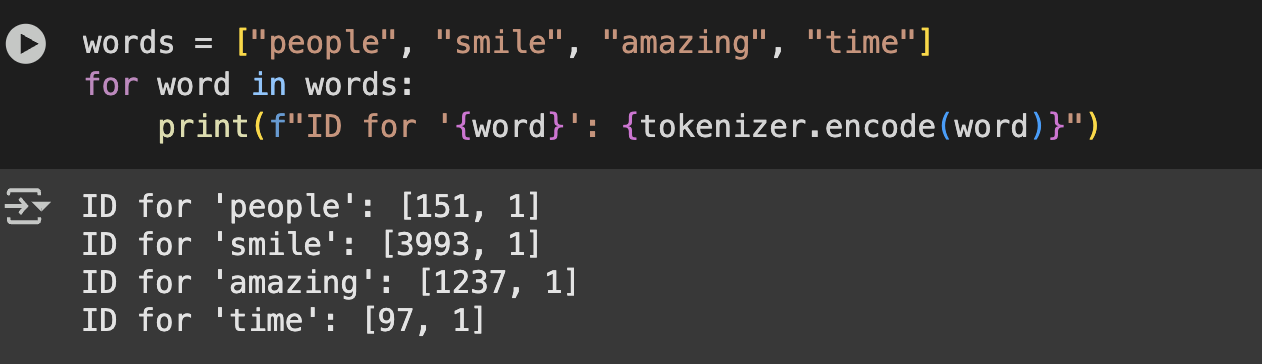
Performance improved massively when using a lower learning rate (0.0001) compared to higher ones (0.001, 0.005, 0.0005). Higher learning rates (0.001, 0.005, 0.0005) made the model unstable or unable to converge properly which might be due to the model overshooting the optimal weights — never properly minimizing loss or parameters oscillate or diverge due to gradient updates being large. Lower learning rate (0.0001) allowed gradual, smoother updates, so the model could learn the subtle differences between the three classes.

**Task 7:**

**i)**



**ii)** The model adds a special token, usually the [EOS] (end-of-sequence) token, at the end of each sequence to indicate where the input sequence ends. Just like BERT has [SEP].  
**iii) BERT**: Unlike FLAN-T5, BERT does not differentiate between input and output text during tokenization, and the labels are usually the target tokens. It is not naturally suited for sequence-to-sequence tasks and typically only operates with input sequences (sentence pairs).  
**FLAN-T5**: This model, which is sequence-to-sequence, handles input and output as sequences.  
The tokenized version of the target text. In our case, the "Intensity Class" of the tweets—that is supplied as labels for training is matched by the encoded labels in FLAN-T5. Usually involving a mechanism to teach the model the desired output given the input, these labels are specific to the output sequence.

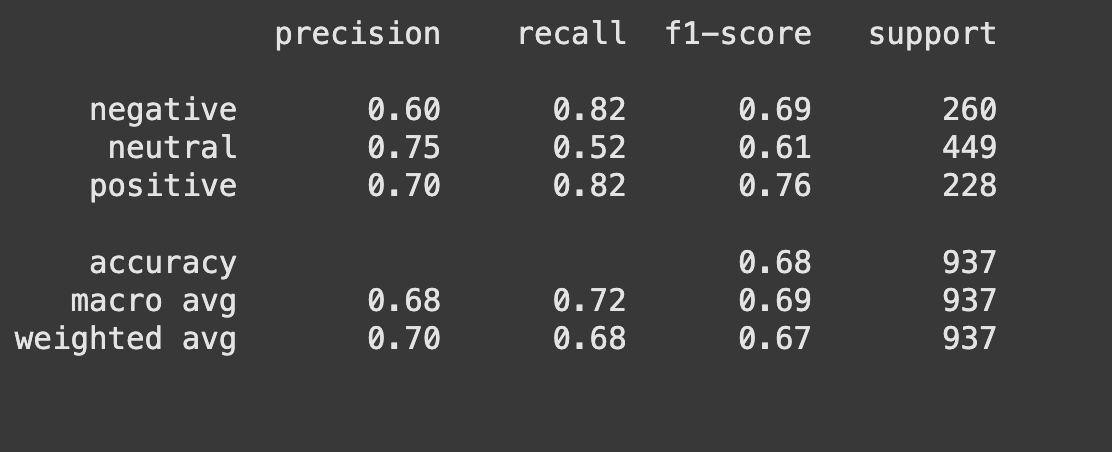
iii) 

**Task 8:**i) DataCollatorForSeq2Seq is a utility that helps prepare batches of data. It performs several functions, including:  
a) Similar to BERT, DataCollatorForSeq2Seq pads sequences to ensure that all sequences in a batch are of the same length.  
b) Ensure that padding tokens are ignored during the loss calculation  
c) Handling the labels for the sequence-to-sequence task, which may also require padding.

ii) padding=True tells the tokenizer to pad all sequences in the batch to the length of the longest sequence (or to a length that is divisible by a given value, which is handled by pad\_to\_multiple\_of, which is 8 in our case). Essentially, like we saw in BERT, it ensures that all sequences in a batch are the same length before being fed into the model. The padding is done by adding special padding tokens to the end of the sequences so that they all have the same length.

iii) batched=True means that the function processes multiple samples at once to reduce computation overhead rather than processing them one by one.

iv) Setting label\_pad\_token\_id to -100 tells the loss function to ignore these padding tokens. This helps the model avoid learning from the padding tokens and focuses only on the meaningful parts of the sequence.

**Task 9:**i) The loss calculation is same as BERT with one exception, which is the Padding Tokens Handling, where padding tokens are ignored during loss calculation.  
ii) 

From the classification report, the F1 scores for each class are:  
Negative class: F1 score = 0.69  
Neutral class: F1 score = 0.61  
Positive class: F1 score = 0.76

And macro f1 score is 0.68

**Task 10:**i) The trainer.predict() function is used to make predictions on a dataset using a trained model. It generates the predicted outputs for the test. This method computes the model’s predictions and compares them with the true labels, providing an evaluation that includes metrics such as accuracy, loss, and other model-specific outputs.  
ii) a. predictions.predictions: The raw model outputs (logits or generated token IDs) for each input in the test set.  
b. predictions.label\_ids: The true labels from the test dataset, aligned with the inputs.  
iii) The predicted token IDs are cleaned up by this line. It substitutes the tokenizer's pad token ID for the prediction at that location whenever the true label is -100, indicating padding or ignored tokens. -100 indicates areas (such as padding) that were ignored during training and testing. Padding tokens are used in place of random predictions at padding positions when decoding predictions back to text.  
iv) skip\_special\_tokens=True tells the tokenizer to remove special tokens like <pad>, and <eos>.

**Task 11:**i) Input: A single text string (a sentence) passed as input\_text.  
Output: A predicted text generated by the model based on the input, printed to the console.  
ii) First we tokenize the input text. Then, we generates prediction using the trained model. At last we decode the prediction, and print the predicted class.

**Task 12:**It was taking a lot of time to run everything at once particularly for epoch 10. So in the code file I segregated it and then my GPU credits got over. So my interpretation will be restricted to epochs 1 and 5. I am certain that the code for epoch 10 will work.  
A screenshot of a computer

AI-generated content may be incorrect.  
A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

There was no improvement in performance across all learning rates and all epochs. Test accuracy stood at ~48% for every setting, which was even lower for our learning rate = 3e-4 and epoch = 5. Loss is decreasing slightly, but the model is not learning useful patterns for classes other than Neutral. This may be due to overfitting to the Neutral dominant class.