# Questions on the code

1. How to define stratifying labels in the train/test split function? We want to split train and testset evenly, so each set has the same amount of samples per category.

**Ans:** When performing a train/test split, it's often important to ensure that the distribution of the target variable (i.e., the labels) is the same in both the training and testing sets. This can be done using stratification.

To stratify the labels in the train/test split function, you can use the stratify parameter. This parameter takes an array or a list of the labels and ensures that each class is represented equally in both the training and testing sets.

Here's an example of how to use train\_test\_split function from the Scikit-learn library in Python to perform a stratified split:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, stratify=y)

1. How to change the loss function to NLLloss? We might want to use a different loss function: NLLloss.

**Ans:** To change the loss function to NLLLoss in a BERT model from Hugging Face, you first need to load the loss function then pass it in list of loss functions.

import torch.nn as nn

NLL\_Loss = nn.NLLLoss() # pass this NLL\_loss in the list of loss functions.

1. Somewhere seeds are missing, as different results are generated, please check!

**Ans:** If you've set the seed for both the train\_test\_split function and the PyTorch training loop using random\_state and torch.manual\_seed, respectively, then your results should be reproducible.

However, there are a few other factors that can affect the reproducibility of your results, such as GPU/CPU variability, multi-threading, and randomness in the data loading process. Here are some possible solutions to try:

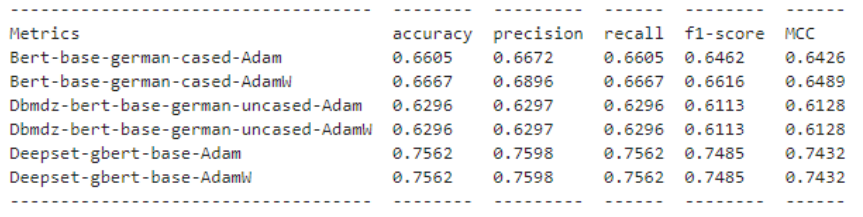
1. Set the seed for other libraries or functions that may have randomness. For example, you can set the seed for NumPy and other data loading libraries that you might be using.
2. Use deterministic operations in your code wherever possible. For example, use torch.mean instead of torch.sum to reduce the variability of your results.
3. Ensure that your model is always initialized in the same way. If you're using a pre-trained model, make sure to load it from the same checkpoint every time.
4. If you're using a GPU, try setting the CUDNN\_DETERMINISTIC and CUDNN\_BENCHMARK environment variables to True. This can help ensure deterministic behavior of the underlying libraries.
5. If you're using multi-threading in your data loading process, try setting num\_workers=0 in your data loader. This will load data in the main thread and can help ensure deterministic behavior.
6. Consider using a single worker for data loading if possible. This can help ensure that the order of data is always the same.

Now I am updating the code for setting seeds in the tokenizer and data loader also, which will display the reproducible results. I am updating the file in the training of the model without fine-tuning and I will attach this with this rebuttal.

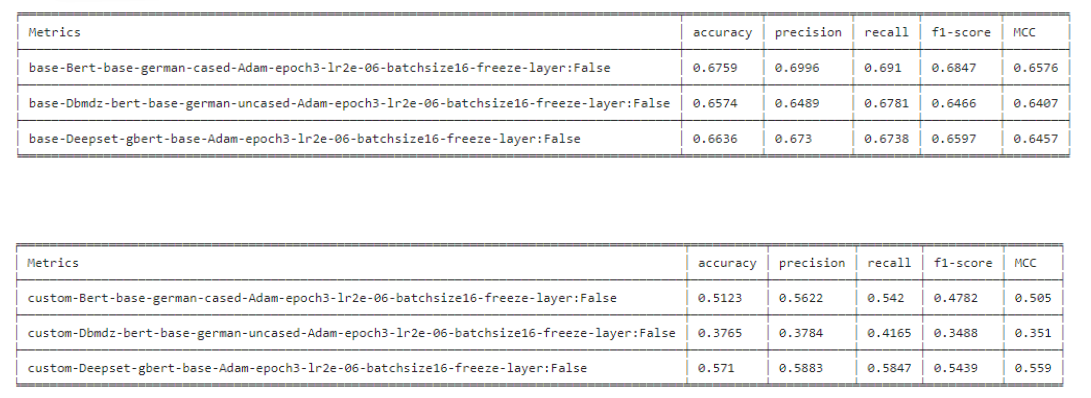
1. How to adapt pipeline without finetuning to generate same or better results as old model version? 🡪 results are worse than before:

“old code” results:

**Ans:** Last time you create two files during the online meeting. One for hyperparameter tuning and one for without hyperparameter tuning. In the file, without hyperparameter tuning, you just need to replace the model definition with the model definition in the first version of the code.



Hyperparameter pipeline code results:



1. The code with the attention weights throws an error:





**Ans:** this one is the old code. I also mentioned in the comment to ignore this one and use the previous one. The code in the previous cell runs perfectly and shows the results in your desired format.