Report of Lexical Complexity Prediction (LCP)

By: Jian Hui Mai

Introduction

The purpose of the Lexical Complexity Prediction is to predict the lexical complexity given a sentence and a word in that sentence. In other words, given a sentence and a word in the sentence, the model attempts to predict how hard it is to understand that word in the given sentence context. For example, given the sentence: “Under basal conditions, the protein resides in subapical intracellular vesicles; however, under conditions requiring water retention AQP2 translocates to the apical membrane, permitting water reabsorption [7,8].”, the word “reabsorption”, the complexity is “0.6”. The word reabsorption is not too difficult to understand but in the given context it is.

A model predicting Lexical Complexity could have many uses. One can use it to predict the difficulty of understanding a certain text before presenting it to an audience. For example, if a subject matter expert wants to present a paper to a non-technical audience, they can use the model to see where the confusing parts may lie. Another could use the model to predict similar text that the reader would understand.

The many working parts make predicting Lexical Complexity challenging. For example, a common word can be difficult to understand given a certain context. We can see that in the example I mentioned above. With that in mind, we must also consider the sentence, the context of that sentence, and the role that the word plays in that sentence. Additionally, everyone views complexity differently. In other words, something difficult for one person to understand might be less challenging for someone else. For example, a subject matter expert in biochemistry would understand the meaning of that sentence better than I can.

Problem Formulation

The inputs of the model are the sentence and a word in that sentence, and it outputs the complexity. The sentences and words come from three corporas: bible, Biomed, and europarl. The training set contains 2,576 rows of Biomed text, 2,574 rows of Bible text, and 2,512 of europarl text with a total of 7,662 sentences. The testing set contains 345 europarl texts, 289 Biomed texts, and 283 Bible texts with a total of 917 sentences. The sentence and word are both in English while the complexity is a numerical value between 0 and 1 with 0 representing an easier understanding and 1 the hardest understanding. In the training set, the average complexity is 0.30 with a minimum of 0.0 and a maximum of 0.86. The testing set has an average complexity of 0.30, a minimum of 0.0, and a maximum of 0.78. The task type of this model is a regression which means the model outputs a numerical value.

Method

Given the importance of both the context and word in predicting the lexical complexity, I decided the BERT (Bidirectional Encoder Representations from Transformers) is the best approach. I believe BERT is the best approach because it can handle long-distance dependencies well. In other words, the BERT architecture does a good job in predicting the dependencies between words. Since I am using BERT for a regression task and not language modeling or next-sentence prediction, I could not leverage the pretraining and could only use the BERT architecture. In essence, the BERT models serve as a feature extractor for both the sentences and words. The feature extractor is then concatenated and passed onto a linear layer to produce a sigmoid output.

Data Processing

The raw data was loaded into a Pandas DataFrame. Due to the way that Pandas interprets ‘null’, the tokens with null were imported as nans so I reverted them to the string null. I also shuffled the dataset and kept everything while resetting the index. The shuffling ensures that the validation sets do not contain only europarl. After shuffling, I removed the columns ID and Corpus since those are not useful for our purposes. This left us with three columns: sentence, token, and complexity. I lowercased the sentence and word columns. Additionally, any punctuation and websites were removed from the sentence column. I did not apply any stemming or lemmatization to give BERT enough context information. The preprocessed dataset was separated into three lists: sentences, tokens, and complexities. The sentences and complexities list were used tokenized/encoded separately and the same goes for the tokens and complexities list. Tokenization/encoding was done using the AutoTokenizer.from_pretrained("bert-base-uncased") which created input_ids and attention_masks for the sentences + complexities and tokens + complexities. Padding, truncation, and max_length was all applied. The sentences + complexity encoding would create input ids and attention masks up to the max_length specified and anything beyond the max_length would be truncated. The same goes for the token + complexity encoding. The two inputs and masks for sentences and tokens were then passed on to a function called dataloaders to create validation splits, tensors, and dataloaders. The dataloaders would return the train_dataloader and val_dataloader which contains all the sentence inputs, labels, and masks.

Model Design

My model contains two BERT architectures. These BERT architectures serve the purpose of a feature extractor. The first BERT architecture uses encoded sentences to predict the complexity while the second BERT architecture uses encoded tokens to predict the same complexity score.

All the models use a dropout and a hidden_act parameter. The final models use a dropout in the hidden layer (hidden_dropout_prob), a dropout in the attention layer (attention_probs_dropout_prob), and a dropout in the classification head. This is done due to the overfitting that was occurring. Additionally, the encoder and pooler layer use a relu function to ensure that we retrieve only positive values. The outputs of both BERT models are then weighted with 0.20 for the sentence BERT and 0.80 for the token BERT. The weighted outputs are then passed into a linear layer with a Sigmoid activation function to produce the final prediction. In other words, my model is an ensemble of two BERT architectures that are weighted. The experiment section below will go into the details of the hyperparameters I used.

Loss Function Design

I was tasked to use the Mean Absolute Error as the loss function (L1 Loss). The Mean Absolute Error is calculated by the average of the summation of the absolute value of all predicted values minus true values.

$$\mathbf{MAE} = \frac{1}{n}\sum_{i=1}^{n} |\, \boldsymbol{x}_i - \boldsymbol{x} \,|$$

Pytorch contains a built-in function that would allow us to calculate the MAE loss, but I decided to implement the formula from scratch. The function is called mae_loss with input parameters of outputs and labels. The outputs refer to the predicted complexity values and the labels refer to the true complexity value.

Training

As previously stated, I created many different models to find the model that yielded me the lowest test mean absolute error score. Each model had a different BERT configuration, batch

size, max_position_embedding, dropout rate, and weights for the BERT output. Despite the differences, all models are trained with the AdamW optimized with 5 epochs using the train function. In each epoch, there is a second loop that goes through each batch to predict the complexity score and update the parameters. At each 100^{th} step, it prints out the current loss using the MAE_loss function that I created above. After training all the data at that specific epoch, the model then calculates the validation loss with the parameters provided. This calculation is done with the evaluate function. It works like the train function where it goes through each batch of the validation dataset and computes the MAE loss. The MAE loss is also printed at every 100^{th} step. Additionally, the function prints out the average training loss and validation loss which I used to create the graphs in the experiment section.

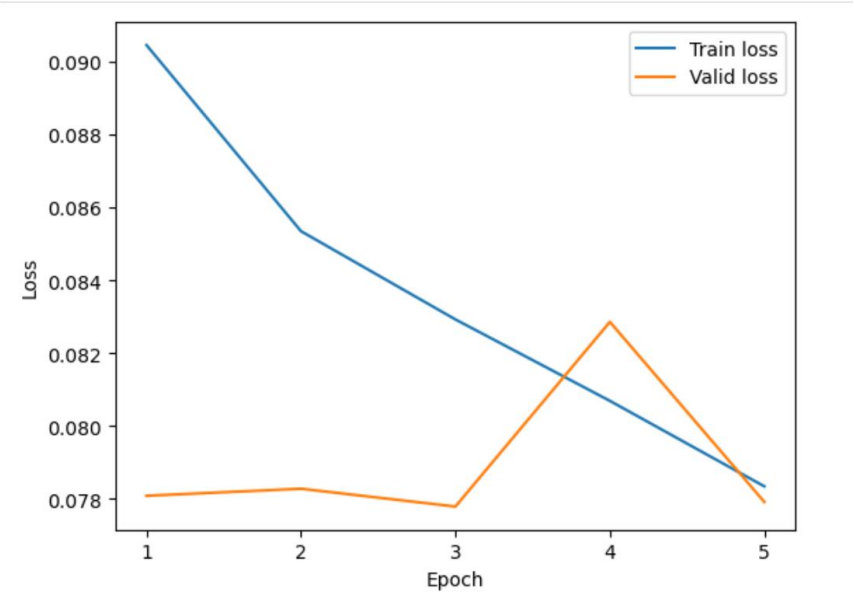
Inference

The inference/prediction function exists in the BERT_Model class namely under the forward function. The forward function takes four parameters: sent_id, sent_mask, token_id, and token_mask. As previously stated, my model contains two BERT models, one for sentence and complexity and another for token and complexity. Both BERT architectures work the same way but are weighted differently. A sentence and complexity are passed into the BERT architecture to produce a prediction given the current configuration. The same goes for the token and complexity BERT architecture. Both use the same configuration namely: 85 max_position_embeddings, relu hidden_act, 0.20 hidden_dropout_prob, 0.20 attention_probs_dropout_prob, and 0.30 classifier_dropout. I would retrieve the last hidden layer from each BERT architecture. Weights were then applied to these last hidden states namely 0.80 for the token model and 0.20 for the sentence model. These outputs were then concatenated with a linear layer applied and a sigmoid activation function to generate the final prediction output.

For any inference on new data, we must use the run_prediction.py which loads the trained model I created called “model.pth”. The loaded model will then use the dataset and forward function described above to predict the complexity scores.

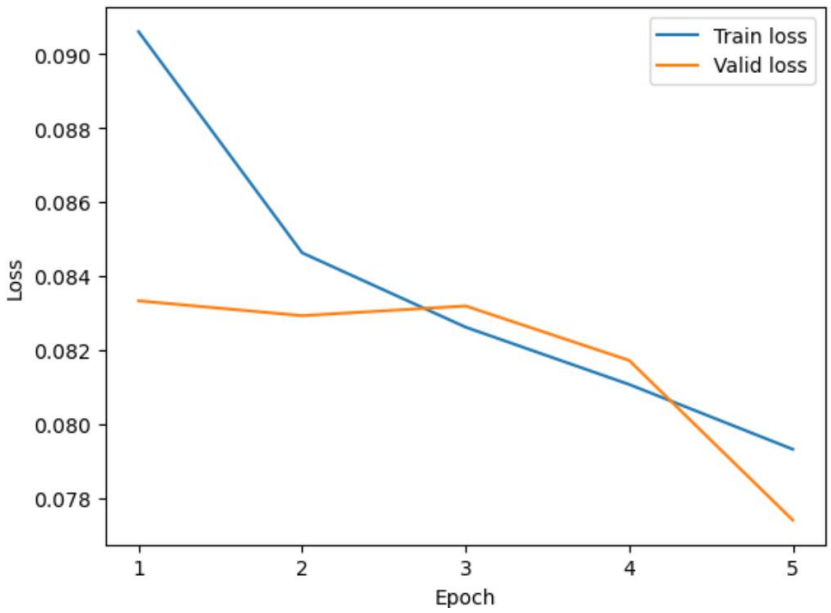
Experiments

Due to the implementation of an ensemble model, most of my experiments involved finding the best weights for the sentence BERT model and token BERT model. I created two initial models. One with a max_position_embedding of 50, hidden_act of relu, and classifier_dropout of 0.20. The weight of this initial model is 0.50 sentence BERT model and 0.50 token BERT model.



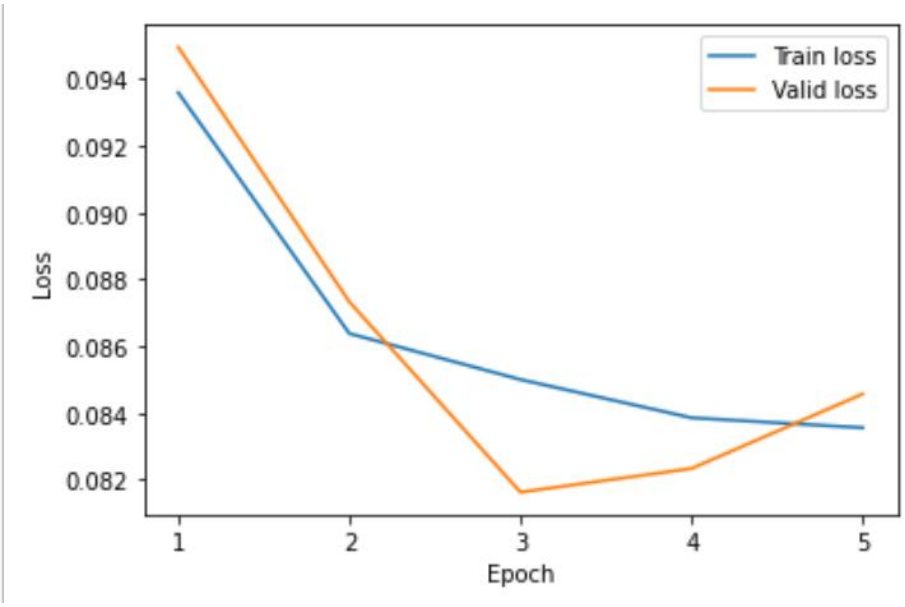
At the last epoch, the average training loss is 0.0783 while the validation loss is 0.0779. For the final total test loss, it was 4.9259. From the graph above, it appears the model is underfitting. Additionally, given the high test loss, I believe we can do better.

For the next model, I changed the weights to 0.20 for the Sentence BERT and 0.80 for the Token BERT. I was able to achieve the loss curve below.



At the last epoch, the average training loss is 0.0793, and the final step validation loss is 0.0774 which was a little lower than previously. I checked the final testing loss which was 4.8627. It is a slight improvement over the previous model but the model still underfits.

For the final model, I increased the (max_position_embeddings to 85. Due to this increase, I had to increase a couple dropouts to prevent overfitting. The default hidden layer dropout was 0.10 and attention layer dropout was 0.10. Now, they are as follows: hidden_dropout_prob = 0.15, attention_probs_dropout_prob = 0.15, classifier_dropout = 0.25. This final model was able to achieve an average training step loss of 0.0836 and final validation step loss of 0.0846.



The final testing loss is 2.6320 which a whole two points lower than the previous model.

Conclusion

I created three models to predict the complexity given the sentence and token. Unsurprisingly, the final model that contained more embeddings produced a better prediction than both the previous models. However, this model is more prone to overfitting which forced me to increase the dropout rate for a couple of layers namely the hidden layer, attention layer and classifier output. Despite the final test MSE of 2.6320, I believe we can still do better. However, given the memory requirements of training two BERT models, I could not increase the max_position_embeddings more. I believe that if I had access to more powerful cloud machines like AWS Sagemaker, I could produce an even more accurate model.