Team Kayak

Karlin Dye and Kai Pak
CS 410 Text Information Systems, Fall 2020
December 13, 2020

BERT For 100% Accuracy (Sarcasm)

13th December, 2020

Introduction

In this paper, we discuss the implementation of a pre-trained BERT NN model to a binary classification problem based on NLP characteristics to detect sarcasm in a dataset consisting of a series of Twitter messages (tweets). We find that the BERT model performs exceedingly well with minimal data processing and hyperparameter tuning. Through repeated experiments, the team consistently scored .71-.75 on F1 with a final F1 score on the leaderboard of 0.745 and beating the baseline. The following sections provide some information on the BERT language model, the infrastructure, language, and libraries for this project, the code, a discussion about the training process and results, and instructions on how to reproduce our final output.

Introduction to BERT

Language models are key components for applications ranging from speech recognition to information retrieval. In the realm of information retrieval, often a unigram model or bag of words representation is employed. However, there are much more advanced language models that employ deep neural networks that can take word context into account and output embeddings of words in a continuous space. BERT, or Bidirectional Encoder Representations from Transformers, is one of these deep learning language models that is able to include context from both the left and right sides of words. BERT was developed by researchers at Google AI Language. More specifically the BERT language model is a pre-trained neural network that through transfer learning can be refined for a particular corpus of new documents. Since the output of word embeddings from the BERT language model includes contextual understanding the embeddings perform well in the realm of text classification.

The BERT neural network architecture consists of 24 layers of transformer blocks, 16 attention heads, and 340 million parameters. The BERT model employs a preprocessing step for input text that can take into account a pair of sequential sentences and includes information about the word positions, sentence positions, and the words themselves. This preprocessing step for text input is an important component of the model training process.

Once input is properly preprocessed the BERT model uses some novel techniques as part of the training process. One challenge involved is how to include both left and right context in a final output embedding. The BERT researchers solved this problem by utilizing a technique called Masked LM (MLM) which masks 15% of input tokens at random and then uses a classification layer to predict the masked token.

An additional strategy that the BERT model uses to help encode context is a technique called Next Sentence Prediction (NSP). The process involves an additional classification layer that is given pairs of sentences and is trained to predict whether one of the sentences in the pair is subsequent from the first. This technique is reliant on the preprocessing of text input that provides information about sentence positions. During training a random sentence from the corpus is paired with an input sentence 50% of the time and the other 50% of the time is paired with the actual subsequent sentence pair from the input.

BERT has been pre-trained on an extremely large corpus of text and is often fine-tuned for a specific application. BERT can be used in a wide range of applications including sentiment analysis, next sentence classification, question answering tasks, named entity recognition (NER), and many more. The use of BERT for many of these tasks have achieved state-of-the-art results.

There are many options for using BERT for text classification tasks. One common approach is the use of the PyTorch Transformers library from HuggingFace. This library contains pre-trained BERT models of different sizes and for different languages and use-cases. The full documentation for how to use the library is available at:

https://huggingface.co/transformers/model_doc/bert.html

Model Code

Two classes were written to handle data processing and the model. Utilizing huggingface package with Python and Pytorch as deep learning framework, we wrote the following classes:

`DataPrep`

The `DataPrep` class is used to process the train.jsonl and test.jsonl files and write out three CSV files that are used for training the classification model and one CSV file that will be used for getting predictions for submission. The *response_only* argument can be used when instantiating the class to either write out the tweet response by itself or both the response and context tweets concatenated together. The *train_test_split* method is used to split the train.jsonl data into a training, validation, and test set to be used for model training. When training the neural network the validation set is used to monitor progress. After training is complete the test set is used to do a final evaluation of the models performance. The *write_data* method writes out three CSV files for the training, validation, and testing along with a file that will be used to generate the predictions for submission.

`SarcasmDetector`

The `SarcasmDetector` class is used to instantiate, train, evaluate, and perform predictions with the BERT based neural network model. For the sake of brevity not all methods and attributes will be described in this paper. A sample notebook in the repository will provide additional detail on how to properly use this class. The primary usage of the class for model training, evaluation, and prediction uses the following sequence of methods:

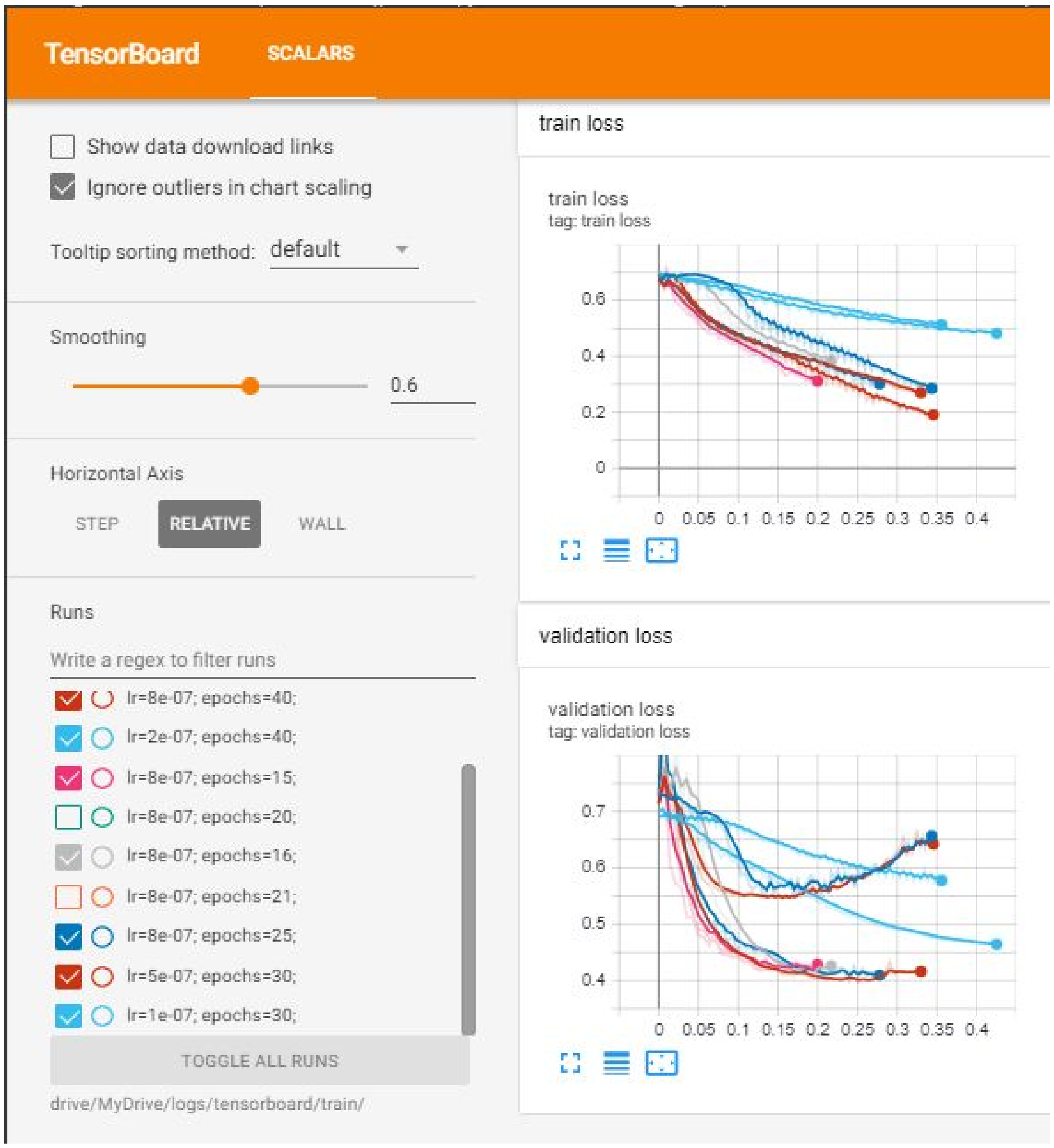
1. Instantiate the class with arguments indicating the directories where the CSV data resides, the model weights should be saved, and the training tensorboard logs should be saved.
2. Use the *tokenize_data* method to tokenize the training, validation, and test text data appropriately for use in the BERT pre-trained model. The names of each file will need to be passed along with an appropriate batch size for training (if you are running out of memory when attempting to train reduce this value).
3. The *tune* method is then utilized to perform a gridsearch for the optimal learning rate hyperparameter. A list of learning rates and a list of the number of epochs to train for must be supplied as arguments when using this method.
4. The *evaluate* method can then be used to provide precision, recall, and f1 metrics for the trained model using the test set.
5. Lastly, the *predict* method uses the trained model to create predictions on the tweets supplied for submission. The output of this method is a dataframe that can then be written to disk for submission.

Infrastructure

We trained our models on the Collab Google environment which is a virtualized jupyter notebook utilizing Tesla GPUs for training.

Results

TensorBoard was used to log training and validation loss during the hyperparameter tuning process. The screenshot below shows the results from a number of training runs utilizing different learning rates and number of epochs.



☐ Show data download links
✓ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis: STEP [✓ RELATIVE] WALL

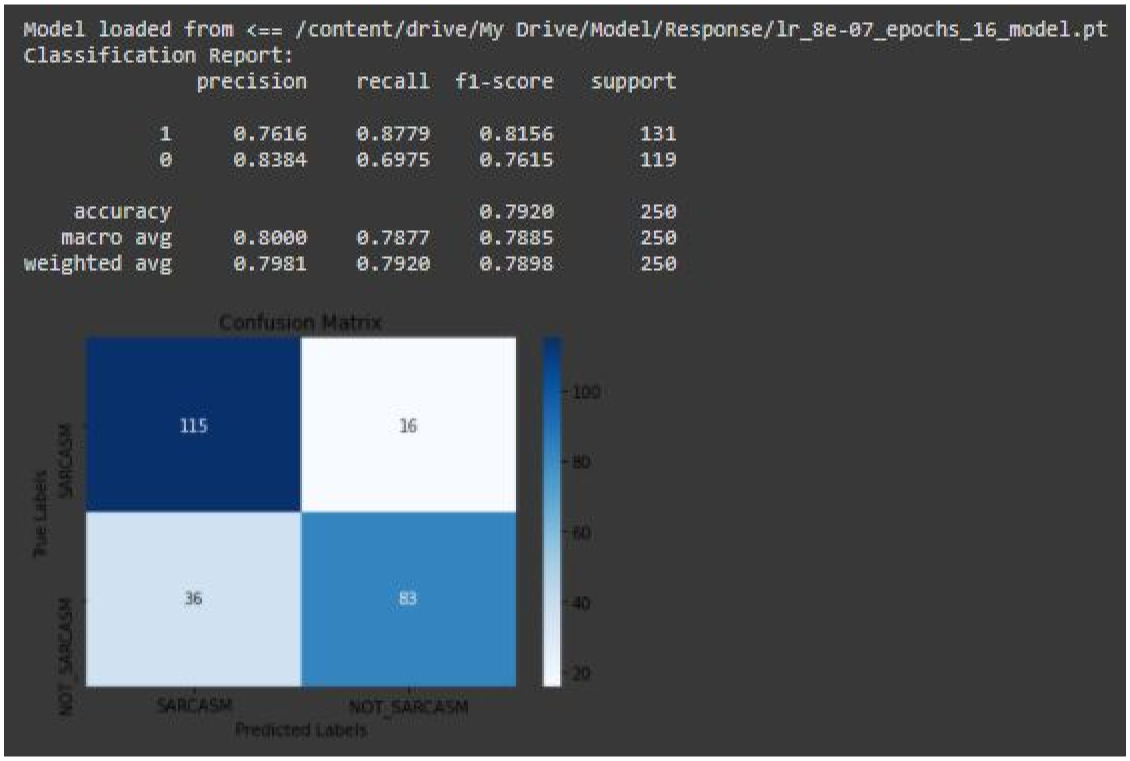
Runs

Write a regex to filter runs

✓ lr=8e-07; epochs=40;
✓ lr=2e-07; epochs=40;
✓ lr=8e-07; epochs=15;
☐ lr=8e-07; epochs=20;
✓ lr=8e-07; epochs=16;
☐ lr=8e-07; epochs=21;
✓ lr=8e-07; epochs=25;
✓ lr=5e-07; epochs=30;
✓ lr=1e-07; epochs=30;

drive/MyDrive/logs/tensorboard/train/

It turned out that using the tweet responses concatenated with context and a learning rate of 8e-7 was the best performing model. Below are the precision, recall, and f1 scores for this model on the held out test data.



```
Model loaded from <== /content/drive/My Drive/Model/Response/lr_8e-07_epochs_16_model.pt
Classification Report:
              precision    recall  f1-score   support

           1     0.7616    0.8779    0.8156       131
           0     0.8384    0.6975    0.7615       119

    accuracy                         0.7920       250
   macro avg     0.8000    0.7877    0.7885       250
weighted avg     0.7981    0.7920    0.7898       250
```