FINAL REPORT

HITHESH SHANMUGAM

CSC 583

TABLE OF CONTENTS:

Development Environment:

I used Google Collab for this final project because it needed GPU to run my model and I used four different accounts for this project. I used python language and the packages I used can be seen in the snippet below:

```
import transformers
import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
from transformers import BertTokenizer, BertForSequenceClassification
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score, precision_score, recall_score
from collections import Counter
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
import shap
```

Figure1: Necessary Packages

Data and Dataset:

Here is the overview of the dataset. For extension I didn't use another dataset because this dataset was handful for me.

Total Instances: 7623

Class Distribution: The dataset contains multiple classes with varying frequencies. Here are some examples of the classes and their corresponding counts:

FALSE: 3140 instances

False: 3109 instances

Misleading: 463 instances

MISLEADING: 173 instances

Mostly false: 101 instances

Partly false: 99 instances

No evidence: 57 instances

... and several more classes with lower counts.

Class Percentages: The class percentages represent the proportion of each class in the dataset. For example:

FALSE: 41.19% of the dataset

False: 40.78% of the dataset

Misleading: 6.07% of the dataset

MISLEADING: 2.27% of the dataset

Mostly false: 1.32% of the dataset

Partly false: 1.30% of the dataset

No evidence: 0.75% of the dataset

... and so on.

***Average Text Length*:** The average text length in the dataset is approximately 6457 characters.

Pre-process:

Filtering for English entries: To ensure that only English entries are included in the analysis, the code filters the dataset by selecting only the rows where the language is set to 'en'. This step helps eliminate entries in other languages and focuses on English text data.

Mapping class labels to target categories: A function named map_labels() is defined to convert the original class labels into target categories. The function takes a label as input and assigns it to one of the following categories: 'TRUE', 'FALSE', 'MISLEADING', 'UNVERIFIED', or 'UNKNOWN'. At the beginning I started with only three labels 'TRUE', 'FALSE', 'UNKNOWN' and as for one extension I added another two labels as the above. That was my final choice.

Creating a new Data Frame: A new Data Frame, named new_df, is created by selecting specific columns ('content_text', 'title', 'source_title', 'ref_category_title', 'class') from the original dataset. These columns are my features. At beginning I used only two features ('content_text', 'class') and for another extension I used a total of five features.

Dropping rows with empty values: To ensure data quality and avoid potential issues during analysis, any rows in new_df that have empty values in the 'content_text' or 'class' columns are dropped. This step helps remove instances with missing or insufficient information.

Mapping class labels to target categories (continued): The 'class' column in new_df is transformed into a 'target' column by applying the map_labels() function to each entry in the 'class' column. This step facilitates working with categorical target labels that are consistent and standardized.

Train-Validation-Test Split:

Train Data: 2276 instances (used for training the model)

Validation Data: 569 instances (used for tuning hyperparameters and evaluating the model during training)

The hyperparameters for my model was:

Optimizer: The Adam optimizer is chosen for training the model. It is a popular optimization algorithm that adapts the learning rate for each parameter based on their gradients. The torch.optim.Adam() function is used to initialize the optimizer, and the model parameters are passed as the input.

Learning rate: The learning rate determines the step size at which the optimizer updates the model parameters during training. In this case, the learning rate is set to 1e-5 (0.00001). A lower learning rate generally leads to slower but more accurate convergence.

Learning rate scheduler: A learning rate scheduler adjusts the learning rate during training based on a predefined schedule. Here, a step-wise scheduler is used, represented by the torch.optim.lr_scheduler.StepLR() function. It decreases the learning rate by a factor of 0.1 after every 5 steps (epochs) of training. This step size and decay rate can be modified based on the specific requirements of the training process.

Loss function: The loss function measures the dissimilarity between the predicted outputs and the true labels, guiding the model to minimize this discrepancy during training. In this case, the cross-entropy loss function (nn.CrossEntropyLoss()) is selected. It is commonly used for multi-class classification problems, such as this sequence classification task with multiple target labels.

These hyperparameters collectively determine the optimization process during training. The optimizer, learning rate scheduler, and loss function work together to update the model's parameters, adjust the learning rate, and calculate the loss for each training iteration. By carefully selecting and tuning these hyperparameters, the model can effectively learn and generalize from the training data.

Model and Tokenizer:

In this project I used only one tokenizer and two models. I first experimented the two models with 2 features and 3 labels and chose to go forward with the best model from the results.

Let's explain the model and tokenizer I used:

Tokenizer:

Name: BERT Tokenizer

Description: The BERT tokenizer is used to preprocess and tokenize text data for input into the BERT model. It applies various text preprocessing techniques, such as word splitting and adding special tokens, to convert raw text into a format suitable for the BERT model.

Pretrained Model: "bert-base-uncased"

Pretraining Dataset: The tokenizer is pretrained on a large corpus of English text, such as the BooksCorpus and English Wikipedia, using an unsupervised language modeling objective. The "uncased" variant means that the tokenizer does not differentiate between uppercase and lowercase letters.

Model1:

Name: BERT for Sequence Classification

Description: The BERT model is a pre-trained transformer-based model developed by Google AI for various natural language processing (NLP) tasks. In your code, you are using the BERT model specifically for sequence classification.

Pretrained Model: "bert-base-uncased"

Architecture:

Transformer Layers: The BERT model consists of a stack of transformer encoder layers. Each layer contains multiple self-attention heads and feed-forward neural networks to capture the contextual information of the input sequence.

Hidden Layer Size: The hidden layer size of the BERT model is determined by the architecture and pretrained model variant. For "bert-base-uncased", it typically has a hidden layer size of 768.

Number of Labels: The BERT model for sequence classification is configured to handle a specific number of labels or classes. In your code, you have set num_labels=3, indicating that the model is designed for a classification task with three possible labels.

Pretraining Dataset: The BERT model is pretrained on a large corpus of text data, such as BooksCorpus and English Wikipedia, using masked language modeling and next sentence prediction objectives. This pretraining enables the model to learn rich representations of text.

The combination of the BERT tokenizer and BERT model allows you to preprocess text data and then classify sequences into one of three classes using the power of pretraining and transfer learning.

Model2:

Name: BERT for Sequence Classification

Description: The BERT model is a pre-trained transformer-based model developed by Google AI for various natural language processing (NLP) tasks. In your code, you are using the BERT model specifically for sequence classification.

Pretrained Model: "yiyanghkust/finbert-tone"

Architecture:

Transformer Layers: The "BertForSequenceClassification" model is built on top of the BERT (Bidirectional Encoder Representations from Transformers) architecture. It consists of a stack of transformer encoder layers, which capture the contextual information of the input sequence through self-attention mechanisms and feed-forward neural networks.

Hidden Layer Size: The hidden layer size of the "BertForSequenceClassification" model is determined by the architecture and the pretrained model variant you are using. Since you are using the "yiyanghkust/finbert-tone" variant, the hidden layer size would typically be the same as the original BERT model, which is 768.

Number of Labels: The "BertForSequenceClassification" model is designed for sequence classification tasks and is configured to handle a specific number of labels or classes. In your code, you have set num_labels=3, indicating that the model is designed for a classification task with three possible labels.

Pretraining Dataset: The "yiyanghkust/finbert-tone" variant of the BERT model is likely pretrained on a specific dataset related to financial sentiment analysis or tone classification. The details of the specific dataset used for pretraining are not mentioned in the code snippet provided. Pretraining allows the model to learn general language representations and can be fine-tuned for the specific sequence classification task at hand.

By using the "yiyanghkust/finbert-tone" pretrained model, you can leverage the contextual understanding and domain-specific knowledge captured during pretraining to perform sequence classification specifically in the financial domain.

Results of Both Models: (Number of labels = 3; Number of features = 2)

```
Epoch 10:
  Train Loss: 0.1054
  Validation Loss: 0.3888
  Validation F1 (Macro): 0.5754
  Validation F1 (Micro): 0.8752
  Validation Precision: 0.8234
  Validation Recall: 0.5354
```

Figure 2: "bert-base-uncased"

```
Epoch 10:
  Train Loss: 0.1298
  Validation Loss: 0.4689
  Validation F1 (Macro): 0.6006
  Validation F1 (Micro): 0.8330
  Validation Precision: 0.6024
  Validation Recall: 0.6051
```

"yiyanghkust/finbert-tone"

From the results I chose to go with *"bert-base-uncased"* model. This was my first extension. I have results of the shapley in the code but here I will be using the shapley of my best model.

Now that I have chosen the model for my next extension, I added two additional labels as you would have seen above.

Results:

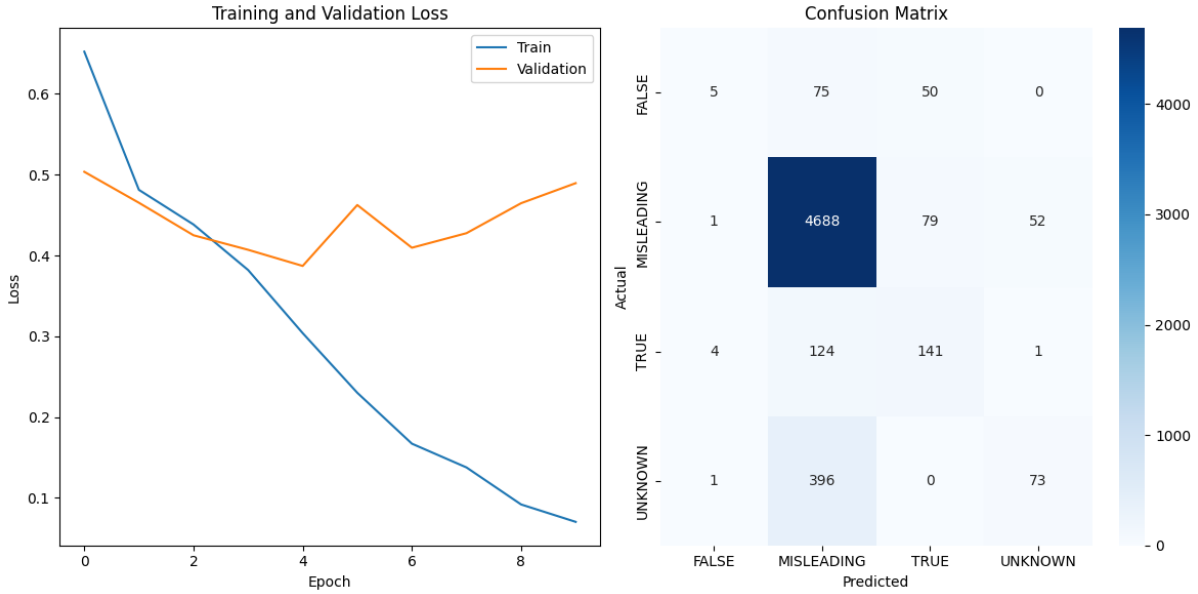

Figure 3: Results of Five label and two class

I was able to run the model only for 10 epochs because by this point it has already taken one of my accounts GPU for the hyperparameter tuning. You could have seen the hyperparameters above those were my best choice for 10 epochs. The score was better than the previous and you can see that below:

```
Train Loss: 0.0703
Validation Loss: 0.4895
Validation F1 (Macro): 0.4415
Validation F1 (Micro): 0.8624
Validation Precision: 0.6109
Validation Recall: 0.4222
```

Now for my last extension I added three additional features in the previous model and this turned out to be better than the previous ones.

The final model has five labels and five features and those were as follows:

Labels: 'TRUE', 'FALSE', 'MISLEADING', 'UNVERIFIED', and 'UNKNOWN'.

Features: 'content_text', 'title', 'source_title', 'ref_category_title', and 'class'.

Results:

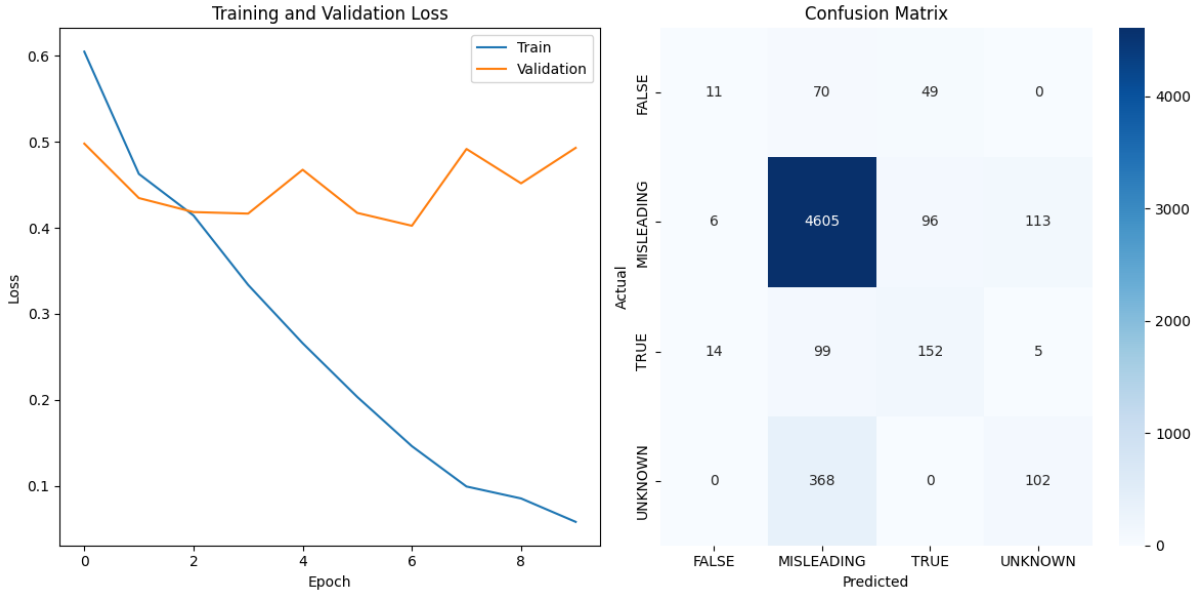

Figure 4: Results of Five label and five class

The score was better than the previous and you can see that below:

```
Train Loss: 0.0584
Validation Loss: 0.4928
Validation F1 (Macro): 0.4732
Validation F1 (Micro): 0.8559
Validation Precision: 0.5565
Validation Recall: 0.4550
```

I could tell more if I could have run the model for more epochs. But I was satisfied with the results for the extensions.

XAI - SHAPLEY

An XAI (Explainable Artificial Intelligence) technique is implemented using the SHAP (Shapley Additive explanations) library. The algorithm aims to provide explanations for the predictions made by the model. The pipeline object is built using the transformers library, which allows making predictions on text classification tasks. The pred object serves as the predictor using the trained model and tokenizer. The explainer object is created using SHAP's Explainer class, which takes the predictor as input. Finally, SHAP values are computed for a subset of examples from the 'ref_category_title' column of the Data Frame, and a visualization of the SHAP values is generated using shap.plots.text().

The shapley was able to do good for some features and not for the other features. I used different features of the shapley for the last model which I had the best results.

The good shapley results was on the feature 'ref_category_title'. The reason this is good because we could see the labels in this particular feature. And the worst was on 'content_text' because I tried to run for the whole and the kernel kept dying and I was able to produce only for a single letter in this. By the time I was debugging this the GPU of my last account ended and I couldn't finish this.

There is a total of five different labels shapley results. And I am attaching only one of those in the below pictures.

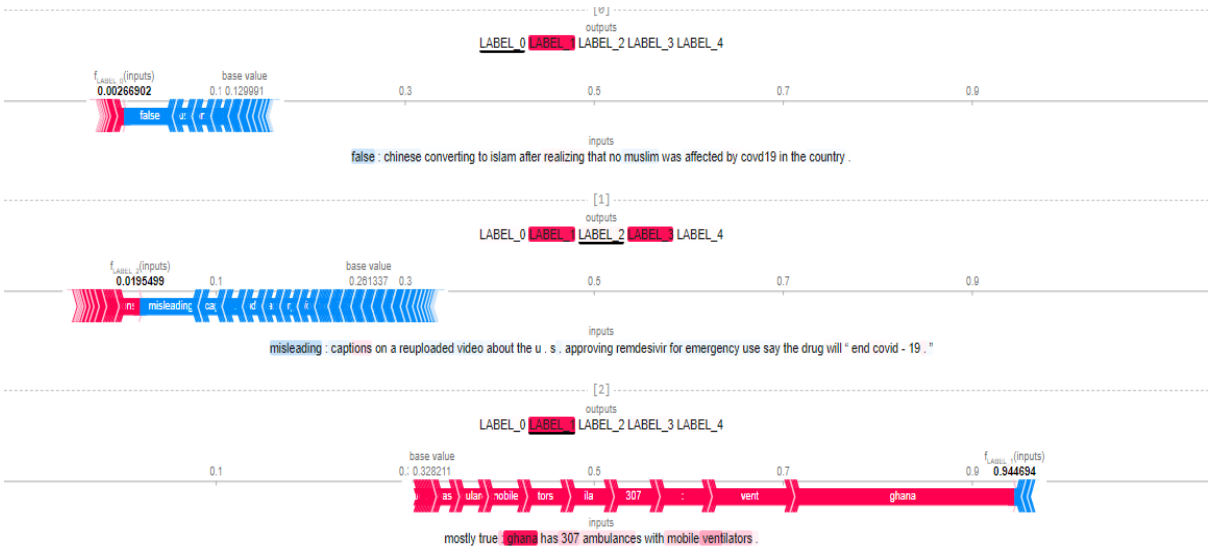

Figure 5: Shapley on good results

Figure 6: Shapley on bad results

General Reflections:

Completing this project was the most challenging task I encountered throughout the course. One of the major difficulties I faced was the shortage of available GPU resources, which limited the amount of work I could accomplish. As a result, I was unable to explore additional extensions than the above or work with other datasets within the given time constraints.

Another crucial aspect that I would focus on in future iterations is addressing class imbalances in the dataset. Class imbalances can significantly impact the performance and accuracy of a classification model. To mitigate this issue, I plan to explore different strategies such as oversampling minority classes, undersampling majority classes, or incorporating class weights during the training process. By implementing these techniques, I can ensure that the model is trained more effectively on all classes, leading to better overall performance.

In conclusion, despite the challenges posed by limited GPU resources and time constraints, I have gained valuable experience in preprocessing text data and developing a text classification model using BERT. Moving forward, I will continue to improve my skills in data handling, address class imbalances, and explore advanced techniques to further enhance the performance of my models.