TWITTER SENTIMENT CLASSIFICATION USING BERT

Github Link: https://github.com/fryohatfield/LLM-Project

Name: Friday Odeh Ovbiroro

ID: 22034665

1.1. INTRODUCTION

Online platforms such as Twitter has actually affected the way people represent their views and experiences on different perspectives in this present age of technology. This particular social media platform is very essential for analysing and categorizing tweets, rather than manual method that are nearly impossible for sentiment analysis.

In Natural Language Processing (NLP), sentiment analysis is a method of classifying tweets into positive, negative and neutral opinions, and this is powered by the automated capabilities of the NLP. This approach facilitates better decision making process in industries because it saves time and helps to give better representation of customers' opinion in line with satisfaction. Using the BERT (Bidirectional Encoder Representations for Transformers) model for sentiment analysis enhances the understanding of tweet content and sentiment . (Pota et al., 2020).

The accuracy of BERT for opinion mining lies on its strength to capture contextual data from text or words. The specific objective of this task is to develop a model that can accurately predict customers' emotions expressed in tweets, allowing for a better insight of public mood. This is achieved by training BERT on a dataset of Twitter messages.

1.2. METHODOLOGY

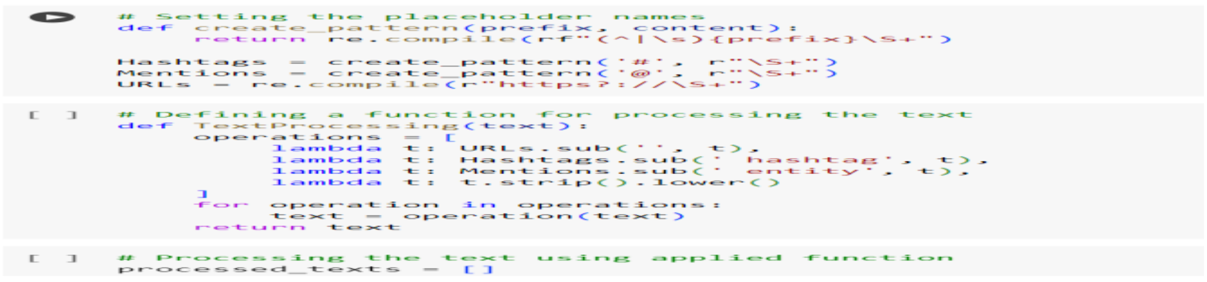
1.2.1. BERT Language Model

This approach involves a large language model, which is pre-trained on a large corpus of text and code. This is a transformer based model that has a unique innovative ability for its bidirectional training, to understand the context of words more effectively by considering text in both directions. Employing the TFBert method from the Hugging Face library, used particularly for categorizing text within a predefined order.

1.3. Dataset and Pre-Processing

The sentiment analysis dataset contains tweets, which are categorised into positive or negative. Before building the model, I first perform pre-processing, to enable the data more suitable for the analysis. The string variables will be converted into numerical values, and this is called label encoding. For the model to be able to understand and process the data, tweets is being transformed into numeric tokens that the model can interpret. This process includes adding special tokens, adjusting the tweets length by padding or truncating sequences and creating attention masks to indicate the essential part of the text. (Ayman Samir, Saleh Mesbah Elkaffas and Madbouly. 2021).

Figure 1: Data Pre-processing



```
# Setting the placeholder names
def create_pattern(prefix, content):
    return re.compile(rf"(^|\s){prefix}\S+")

Hashtags = create_pattern('#', r"\S+")
Mentions = create_pattern('@', r"\S+")
URLs = re.compile(r"https?://\S+")
```

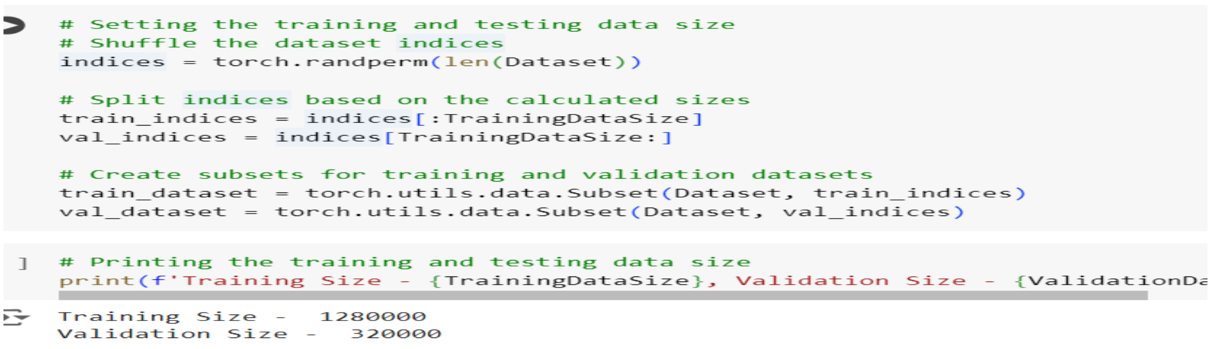
```
# Defining a function for processing the text
def TextProcessing(text):
    operations = [
        lambda t: URLs.sub('', t),
        lambda t: Hashtags.sub(' hashtag', t),
        lambda t: Mentions.sub(' entity', t),
        lambda t: t.strip().lower()
    ]
    for operation in operations:
        text = operation(text)
    return text
```

```
# Processing the text using applied function
processed_texts = []
```

1.4.   Splitting the Dataset

At this point, the dataset was divided into two sets, called the training and testing sets. The training set is used to optimize the data for better performance, while the testing set is used to assess its performance. The “torch” library is used for efficient data handling, with the “torch.utils.data” module which helps to load, batch and shuffle data during training.

Figure 2: Splitting of the dataset



```
# Setting the training and testing data size
# Shuffle the dataset indices
indices = torch.randperm(len(Dataset))

# Split indices based on the calculated sizes
train_indices = indices[:TrainingDataSize]
val_indices = indices[TrainingDataSize:]

# Create subsets for training and validation datasets
train_dataset = torch.utils.data.Subset(Dataset, train_indices)
val_dataset = torch.utils.data.Subset(Dataset, val_indices)
```

```
# Printing the training and testing data size
print(f'Training Size - {TrainingDataSize}, Validation Size - {ValidationDa
```

```
Training Size -  1280000
Validation Size -  320000
```

2. TRAINING, MONITORING, FINE-TUNING AND DEPLOYMENT

2.1 Training

Training of the data undergone through several rounds, called epochs. Each epoch has to do with passing the data through the network, updating the model’s weights, and applying backpropagation. For the purpose of reducing overfitting or preventing model from becoming too specialised to the training data, dropout and other approaches were used. (Kharde and Sonawane, 2016).

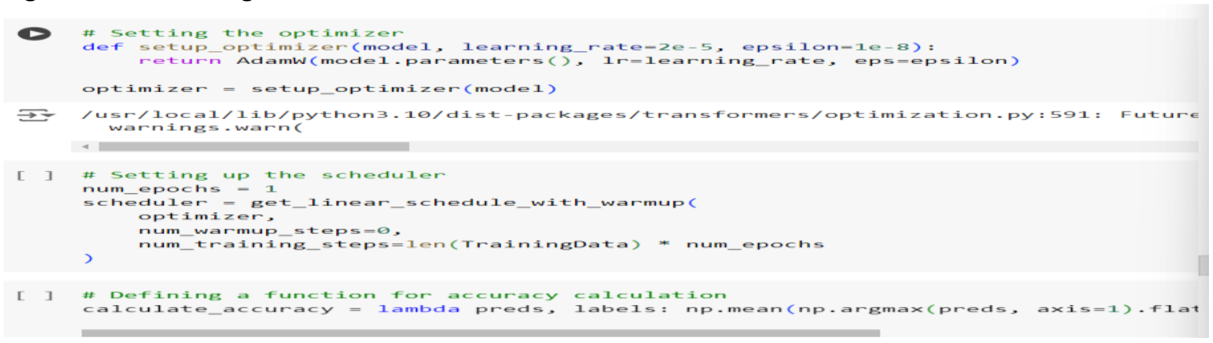
2.2. Monitoring Training Progress

To track the model’s training outcome and prevent overfitting, I assessed the model’s performance on the test set after each round of training (epoch). Key metrics like accuracy and loss function were used to evaluate how well the model performed on the new data. During training, validation metrics were closely observed, to implement early stop if the validation loss started to rise, to avoid overfitting.
The PyTorch allowed a log detailed training data, and tools like TensorBoard and matplotlib were used to visualise progress and giving meaningful insights into the model learning behaviour.

2.3. **Fine-Tuning BERT**

In Natural Language Processing, PyTorch is used extensively for developing language models like BERT and other language understanding systems. This is done by optimizing the pre-trained BERT model on Twitter dataset of the training set. A well suited BERT optimizer called the AdamW, was utilized to improve the model's performance, after it was constructed with a cross entropy loss function and trained by feeding it batches of tokenized tweets along with their labels. To reduce prediction errors, the model's weight was continuously adjusted.

Figure 3: Fine-Tuning section of the model.



```
# Setting the optimizer
def setup_optimizer(model, learning_rate=2e-5, epsilon=1e-8):
    return AdamW(model.parameters(), lr=learning_rate, eps=epsilon)

optimizer = setup_optimizer(model)
```

```
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:591: Future
  warnings.warn(
```

```
# Setting up the scheduler
num_epochs = 1
scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=0,
    num_training_steps=len(TrainingData) * num_epochs
)
```

```
# Defining a function for accuracy calculation
calculate_accuracy = lambda preds, labels: np.mean(np.argmax(preds, axis=1).flat
```

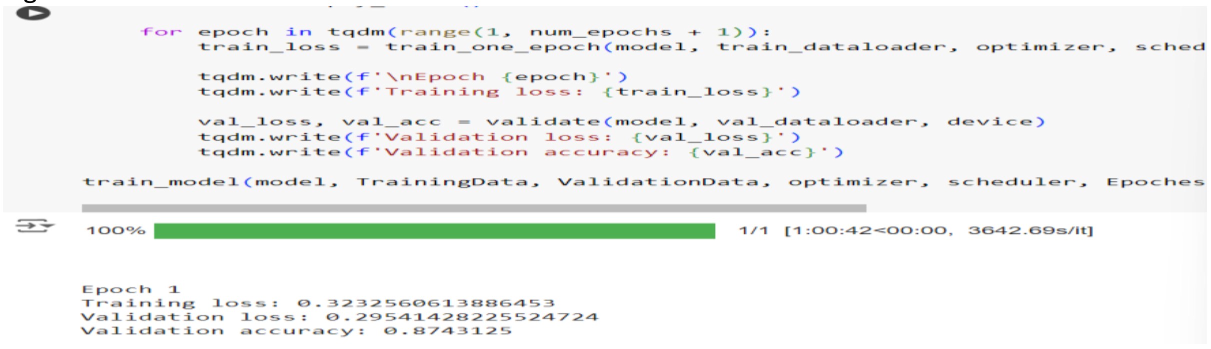
2.3. Deployment

Immediately after training and evaluation, the model was tested with real-world situations by making it available to users who were able to quickly classify the sentiment of the tweets they entered. The model displayed a high level of accuracy in determining tweet sentiment. The model created the log data and was processed by using the SoftMax function to ascertain whether a tweet was positive or negative. Deployment was streamlined by using PyTorch's "torch.save" to export trained model for integration into other applications. (Bello, Ng and Leung, 2023).

3. RESULT

The BERT model trained on the Sentiment140 dataset performed well, achieving a training loss of 0.3233 and a validation loss of 0.2954. This indicated that it learned effectively and generalized well to new data. The model correctly classified the sentiment of 87.43% of tweets in the validation set, showing a strong accuracy after just one epoch of training. These results demonstrated that the model was well-tuned and capable of reliably identifying the sentiment of tweets with minimal errors.
Demonstrating the accuracy of BERT training technique in understanding complicated pattern of language. (Bello, Sin Chun Ng and Leung, 2023).

Figure 4: Result Section



```
    for epoch in tqdm(range(1, num_epochs + 1)):
        train_loss = train_one_epoch(model, train_dataloader, optimizer, sched

        tqdm.write(f'\nEpoch {epoch}')
        tqdm.write(f'Training loss: {train_loss}')

        val_loss, val_acc = validate(model, val_dataloader, device)
        tqdm.write(f'Validation loss: {val_loss}')
        tqdm.write(f'Validation accuracy: {val_acc}')

train_model(model, TrainingData, ValidationData, optimizer, scheduler, Epoches
```

```
100% ██████████████████████████████ 1/1 [1:00:42<00:00, 3642.69s/it]


Epoch 1
Training loss: 0.3232560613886453
Validation loss: 0.29541428225524724
Validation accuracy: 0.8743125
```

3.1. CONCLUSION

For this task using BERT model, suggests a strong choice for sentiment analysis for this project, especially for handling social media text. It shows how to optimize a pre-trained BERT model for sentiment analysis on Twitter messages using PyTorch. However, regardless of fine-tuning challenges encountered with large language models like BERT, there are many advantages of

using BERT for sentiment analysis. It is a powerful tool for social media data analysis due to its precision and contextual understanding.
Based on future research, the model's performance could improve better, with further training and adjustments.

References

Bello, A., Leung, M.-F. and Ng, S.C. (2023). *A BERT Framework to Sentiment Analysis of Tweets*. [online] ResearchGate. Available at: https://www.researchgate.net/publication/366811050_A_BERT_to_Sentiment_Analysis_of_Tweets [Accessed 8 Aug. 2024]

Bello, A., Ng, S.-C. and Leung, M.-F.(2023). A BERT Framework to Sentiment Analysis of Tweets. *Sensors,* [online] 23(1), pp.506-506. doi:https://doi.org/10.3390/s23010506.

Kharde, V. and Sonawane, S. (2016). Sentiment Analysis of Twitter Data: A Survey of Techniques. *International Journal of Computer Applications*, [online] 139(11), pp.975-8887. Available at: https://arxiv.org/pdf/1601.06971.

Pota, M., Ventura, M., Catelli, R. and Esposito, M. (2020). An Effective BERT-Based Pipeline for Twitter Sentiment Analysis: A Case Study in Italian. *Sensors*, [online] 21(1), pp.133-133. doi:https://doi.org/10.3390/s21010133.