NLP PROJECT

Question Answering and CLASSIFICATION SYSTEM

DATA SET: IOS VS. ANDROID

By: Sarah Hussien
&
Hams Dorgham

Table of CONTENTS

INTRODUCTION

- NLP question answering and classification systems play a crucial role in enhancing how machines interpret and respond to human language.
- By enhancing the precision and speed of these systems, we can streamline information retrieval, enhance user experiences, and facilitate better human-machine communication.
- Understanding the workings of question classification and answering in NLP is essential for developers to create reliable models and improve system performance through informed decisions on model architectures and training methodologies.

MILESTONE 2

IMPLEMENTS OF A TEXT CLASSIFICATION MODEL USING LSTM (LONG SHORT-TERM MEMORY) NEURAL NETWORK ARCHITECTURE ON A DATASET.

1. **DATA PREPROCESSING:**

- THE CODE BEGINS BY IMPORTING NECESSARY LIBRARIES SUCH AS PANDAS, NUMPY, NLTK FOR NLP TASKS LIKE TOKENIZATION, LEMMATIZATION, ETC., AND TENSORFLOW/KERAS FOR BUILDING AND TRAINING THE NEURAL NETWORK.

- TEXT PREPROCESSING STEPS LIKE CONVERTING TEXT TO LOWERCASE, REMOVING PUNCTUATION, TOKENIZATION, REMOVING STOPWORDS, AND LEMMATIZATION ARE PERFORMED USING NLTK.

- THE PREPROCESSED TEXT IS THEN USED TO TRAIN A WORD2VEC MODEL TO CONVERT WORDS INTO DENSE VECTORS.

Import Libraries and Dataset

```
import pandas as pd
import numpy as np
import nltk
#used for tokenization
nltk.download('punkt')
from nltk.tokenize import word_tokenize
#used for lemmatization
nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer
#used for Stemming
from nltk.stem import PorterStemmer
#used for removing stopwords
nltk.download('stopwords')
from nltk.corpus import stopwords
#used for removing punctuations
import string
#used for POS Tagging
nltk.download('averaged_perceptron_tagger')
#used for visualizations
import seaborn as sns
import matplotlib.pyplot as plt
from wordcloud import WordCloud

import tensorflow as tf
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Sequential
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Embedding, SimpleRNN, Concatenate
```

MILESTONE 2

```python
def preprocess_text(text):
    # Standardizing text to Lowercase
    text = text.lower()
    # Removing the punctuation
    text = text.translate(str.maketrans('', '', string.punctuation))
    # Tokenizing the text
    tokens = word_tokenize(text)
    # Removing stopwords
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word not in stop_words]
    #Lemmatization
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(word) for word in tokens]
    return tokens
```

MILESTONE 2

2. WORD EMBEDDING WITH WORD2VEC:

- THE WORD2VEC MODEL IS TRAINED ON THE PREPROCESSED TEXT DATA TO CREATE WORD EMBEDDINGS.
- EACH WORD IS REPRESENTED AS A DENSE VECTOR OF A FIXED SIZE (100 DIMENSIONS).
- THESE WORD EMBEDDINGS CAPTURE SEMANTIC MEANING AND CONTEXT OF WORDS IN THE TEXT.

3. FEATURE ENGINEERING:

EACH SENTENCE IN THE DATASET IS CONVERTED INTO AN AVERAGE VECTOR REPRESENTATION BY TAKING THE MEAN OF THE WORD VECTORS OF ALL WORDS IN THE SENTENCE.

```python
import gensim
from gensim.models import Word2Vec
# Training a Word2Vec model with a vector size of 100
sentences = df['txt_Concatenated'].tolist()
word2vec_model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, workers=4)

# creating a Function to convert sentences to the average of their word vectors
def sentence_to_avg_vector(sentence):
    vectors = [word2vec_model.wv[word] for word in sentence if word in word2vec_model.wv]
    return np.mean(vectors, axis=0) if len(vectors) > 0 else np.zeros(100)

Resulting_Vectors = np.array([sentence_to_avg_vector(sentence) for sentence in sentences])
```

```python
Resulting_Vectors
```

```
rray([[ 1.0166194 ,  1.5930845 , -0.16051413, ..., -0.3220234 ,
        0.35284796,  0.72307634],
      [ 1.123724  ,  2.5170949 , -0.25199008, ..., -0.35289535,
        0.08950842, -0.38245296],
      [ 0.7088074 ,  2.403312  , -0.30885723, ..., -0.41211787,
       -0.4560729 , -0.6023427 ],
      ...,
      [ 0.7053586 ,  0.7364668 , -0.2808236 , ..., -0.22677502,
        0.3912683 ,  0.5631373 ],
      [ 1.2523965 ,  1.6723537 , -0.15821514, ...,  0.1642752 ,
        0.5111507 ,  0.40103722],
      [ 0.84566516,  1.4925274 , -0.63777786, ..., -0.41319183,
       -0.32444426,  0.25579965]], dtype=float32)
```

MILESTONE 2

4. SPLITTING DATA

5. MODEL ARCHITECTURE:

- A SEQUENTIAL LSTM-BASED NEURAL NETWORK MODEL IS BUILT USING TENSORFLOW/KERAS.
- THE INPUT SHAPE IS DEFINED BASED ON THE SHAPE OF THE FEATURE VECTORS OBTAINED AFTER FEATURE ENGINEERING.
- THE LSTM LAYER WITH 64 UNITS IS FOLLOWED BY DROPOUT REGULARIZATION TO PREVENT OVERFITTING.
- DENSE LAYERS WITH RELU ACTIVATION ARE ADDED FOR FURTHER PROCESSING, AND A FINAL DENSE LAYER WITH SIGMOID ACTIVATION IS ADDED FOR BINARY CLASSIFICATION.
- THE MODEL IS COMPILED WITH THE ADAM OPTIMIZER AND BINARY CROSS-ENTROPY LOSS FUNCTION.
-

6. MODEL TRAINING:

- THE MODEL IS TRAINED ON THE TRAINING DATA FOR 10 EPOCHS WITH A BATCH SIZE OF 32.
- VALIDATION DATA IS ALSO SPECIFIED FOR MONITORING THE MODEL'S PERFORMANCE DURING TRAINING.

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM


timesteps = 1  # Assuming each sample is one timestep
features = Resulting_Vectors.shape[1]

X_train_lstm = np.reshape(X_train, (X_train.shape[0], timesteps, features))
X_test_lstm = np.reshape(X_test, (X_test.shape[0], timesteps, features))

model = Sequential()
# LSTM layer with 64 units, input_shape is the shape of each sample in the dataset
model.add(LSTM(64, input_shape=(timesteps, features)))
model.add(Dropout(0.2))
# Dense layer for further processing
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.2))
# Sigmoid for binary classification
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(X_train_lstm, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

Python

```
Epoch 1/10
1468/1468 [==============================] - 8s 4ms/step - loss: -1715.3485 - accuracy: 0.2618 - val_loss: -4700.6548 - val_accuracy: 0.2688
Epoch 2/10
1468/1468 [==============================] - 7s 5ms/step - loss: -10606.1094 - accuracy: 0.2618 - val_loss: -16482.7520 - val_accuracy: 0.2689
Epoch 3/10
1468/1468 [==============================] - 5s 3ms/step - loss: -25590.5410 - accuracy: 0.2619 - val_loss: -33151.8047 - val_accuracy: 0.2691
Epoch 4/10
1468/1468 [==============================] - 6s 4ms/step - loss: -45775.1992 - accuracy: 0.2618 - val_loss: -54894.5195 - val_accuracy: 0.2688
Epoch 5/10
1468/1468 [==============================] - 6s 4ms/step - loss: -71246.3438 - accuracy: 0.2621 - val_loss: -81431.6797 - val_accuracy: 0.2688
Epoch 6/10
1468/1468 [==============================] - 5s 4ms/step - loss: -101449.4531 - accuracy: 0.2620 - val_loss: -112008.1719 - val_accuracy: 0.26
Epoch 7/10
```

7. MODEL EVALUATION

```
loss, accuracy = model.evaluate(X_test_lstm, y_test)

print("Test Loss:", loss)
print("Test Accuracy:", accuracy)
```

```
459/459 [==============================] - 1s 2ms/step - loss: -300512.2812 - accuracy: 0.2549
Test Loss: -300512.28125
Test Accuracy: 0.2548712491989136
```

-TEST LOSS OF -300512.28125 IS EXTREMELY NEGATIVE. LOSS VALUES ARE TYPICALLY NON-NEGATIVE, REPRESENTING THE DISCREPANCY BETWEEN PREDICTED AND TRUE VALUES.

-SUCH A LARGE NEGATIVE LOSS VALUE IS HIGHLY UNUSUAL AND INDICATES A SIGNIFICANT PROBLEM WITH THE MODEL OR ITS EVALUATION. IT COULD BE AN INDICATION OF INCORRECT LOSS CALCULATION OR A MALFUNCTION IN THE MODEL'S TRAINING OR EVALUATION PROCESS.

- TEST ACCURACY OF APPROXIMATELY 25.49% IS RELATIVELY LOW.

-WHILE TEST ACCURACY VALUES VARY BASED ON THE SPECIFIC TASK AND DATASET, A TEST ACCURACY OF AROUND 25% SUGGESTS THAT THE MODEL'S PERFORMANCE IS NOT SATISFACTORY. IT MEANS THAT THE MODEL IS CORRECTLY CLASSIFYING ONLY ABOUT A QUARTER OF THE INSTANCES IN THE TEST SET.

MILESTONE 2

LIMITATIONS AND CONSIDERATIONS:

- A RELATIVELY SIMPLE LSTM ARCHITECTURE WAS USED, WHICH MIGHT NOT CAPTURE COMPLEX PATTERNS IN TEXT DATA EFFECTIVELY, ESPECIALLY FOR LARGE DATASETS WITH DIVERSE TEXT STRUCTURES.
- WORD2VEC EMBEDDINGS MIGHT NOT CAPTURE RARE OR OUT-OF-VOCABULARY WORDS WELL, IMPACTING THE MODEL'S PERFORMANCE.
- THE PREPROCESSING STEPS SUCH AS STOPWORD REMOVAL, LEMMATIZATION, ETC., MIGHT NOT BE OPTIMAL FOR ALL DATASETS AND COULD BE FURTHER REFINED BASED ON SPECIFIC REQUIREMENTS.
- THE CODE DOES NOT INCLUDE EXTENSIVE HYPERPARAMETER TUNING, WHICH COULD POTENTIALLY IMPROVE THE MODEL'S PERFORMANCE.

MILESTONE 2

EXAMPLE OF A USER INPUT AND MODEL'S PREDICTION

```
# This is an ios Question
question1 = ["How does iOS differ from other mobile operating systems?"]
processed_question = preprocess_text(question1[0])
# Converting question to vector
question_vector = np.array([sentence_to_avg_vector(processed_question)])
#predicted_label1 = model.predict(question_vector)
#print("Predicted Label:", map_output_to_label(predicted_label1[0][0]))
question_vector_lstm = np.reshape(question_vector, (question_vector.shape[0], 1, question_vector.shape[1]))

# Predict the label
predicted_label1 = model.predict(question_vector_lstm)

# Map the output to labels
predicted_label = map_output_to_label(predicted_label1[0][0])

print("Predicted Label:", predicted_label)
#(Correctly predicted by the Model)
```

```
1/1 [==============================] - 0s 386ms/step
Predicted Label: iOS
```

MILESTONE 2-REGRESSION MODEL

```
model_Reg = Sequential()
model_Reg.add(Dense(64, activation='relu', input_dim = 100))
model_Reg.add(Dropout(0.2))
model_Reg.add(Dense(1))  # Single output neuron for regression; no activation function

model_Reg.compile(optimizer='adam', loss='mean_squared_error')
model_Reg.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

```
Epoch 1/10
1468/1468 [==============================] - 4s 2ms/step - loss: 29.8742 - val_loss: 29.7661
Epoch 2/10
1468/1468 [==============================] - 3s 2ms/step - loss: 29.5621 - val_loss: 29.8538
Epoch 3/10
1468/1468 [==============================] - 4s 3ms/step - loss: 29.4603 - val_loss: 29.5387
Epoch 4/10
1468/1468 [==============================] - 3s 2ms/step - loss: 29.4425 - val_loss: 29.4439
Epoch 5/10
1468/1468 [==============================] - 3s 2ms/step - loss: 29.3192 - val_loss: 29.6010
Epoch 6/10
1468/1468 [==============================] - 3s 2ms/step - loss: 29.2887 - val_loss: 29.4526
Epoch 7/10
1468/1468 [==============================] - 5s 3ms/step - loss: 29.2132 - val_loss: 29.4055
Epoch 8/10
1468/1468 [==============================] - 3s 2ms/step - loss: 29.1447 - val_loss: 29.5240
Epoch 9/10
1468/1468 [==============================] - 3s 2ms/step - loss: 29.0847 - val_loss: 29.4393
Epoch 10/10
1468/1468 [==============================] - 3s 2ms/step - loss: 29.0683 - val_loss: 29.3383

<keras.src.callbacks.History at 0x7fd45f6c1ba0>
```

MILESTONE 2-REGRESSION MODEL

Using Mean Squared Error to Calculate the loss in the regression model

[31]

```
mse = model_Reg.evaluate(X_test, y_test)
print('Test MSE:', mse)
```

```
459/459 [==============================] - 1s 1ms/step - loss: 43.4585
Test MSE: 43.458465576171875
```

Prediction Testing (Trial) Real Life Questions as input to the regression model

[36]

```
question4 = ["How does iOS differ from other mobile operating systems?"]
processed_question = preprocess_text(question4[0])
# Converting question to vector
question_vector = np.array([sentence_to_avg_vector(processed_question)])
predicted_score = model_Reg.predict(question_vector)
print("Predicted Score:", np.round(predicted_score[0][0]))
```

```
1/1 [==============================] - 0s 19ms/step
Predicted Score: 2.0
```

Milestone 3

Using predefined BERT model on Binary Classification task

1. Data Preparation

```
#This CustomDataset class allows encapsulating the data and handle tokenization,
#encoding, padding, and truncation within the dataset itself, making it easier to work with BERT models in PyTorch.

class CustomDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

    #This method returns the total number of samples in the dataset. In this case, it returns the length of the texts array.
    def __len__(self):
        return len(self.texts)
```

- CustomDataset class is a tailored class used to prepare data for training machine learning models, particularly those based on transformers like BERT by handling tokenization, encoding, padding, and truncation of the text data

Milestone 3

```python
def __getitem__(self, idx):
    text = str(self.texts[idx])
    label = self.labels[idx]

    encoding = self.tokenizer.encode_plus(    #The text is tokenized and encoded using the provided tokenizer (self.tokenizer.encode_plus()).
                                              #This method converts the text into input IDs and attention mask, which are necessary inputs for BERT.
        text,
        add_special_tokens=True,
        max_length=self.max_len,
        return_token_type_ids=False,
        pad_to_max_length=True,        #--->  #The tokenizer's encode_plus method is called with pad_to_max_length=True and truncation=True.
                                              #This ensures that all input sequences have the same length (max_len) by padding shorter sequences and truncating longer sequences.
        return_attention_mask=True,
        return_tensors='pt',
        truncation=True
    )

    return {
        'text': text,
        'input_ids': encoding['input_ids'].flatten(),
        'attention_mask': encoding['attention_mask'].flatten(),
        'label': torch.tensor(label, dtype=torch.long)
    }
```


- “CustomDataset” class includes “getitem” method that performs the following:

1. allows indexing to access individual samples from the dataset.
2. tokenizes and encodes the input text using BertTokenizer
3. uses ”encodeplus” method to converts the input text into input IDs and attention mask
4. returns a dictionary containing the original text, input IDs, attention mask, and label to be used as inputs for transformer-based models like BERT.

Milestone 3

2. Model Loading

```
[ ] tokenizer = BertTokenizer.from_pretrained('prajjwal1/bert-tiny')
    model = BertForSequenceClassification.from_pretrained('prajjwal1/bert-tiny', num_labels=2)
```

*Train-Test Splitting *

```
[ ] # Splitting data into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(df['txt_Concatenated'], df['LabelNum'], test_size=0.2, random_state=42)


    # Defining training and testing datasets
    train_dataset = CustomDataset(X_train.values, y_train.values, tokenizer, max_len=128)
    test_dataset = CustomDataset(X_test.values, y_test.values, tokenizer, max_len=128)


    # Defining data loaders
    train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)          #This randomization helps prevent the model from learning the order of
                                                                                   #the data and can improve the training process by introducing variability
                                                                                   #in the batches presented to the model during each training iteration.
                                                                                   #----> reducing the risk of overfitting to the training data



    test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)           #This consistency allows for fair and reproducible
                                                                                   #evaluation of the model's performance on the test set.
```

- The library used: BertForSequenceClassification.
- The model name: ’prajjwal1/bert-tiny’ model

Milestone 3

3. Training the Model

```
# Specifying Training parameters
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5)
criterion = torch.nn.CrossEntropyLoss()

# Defining the number of epochs and Training loop
epochs = 3
for epoch in range(epochs):
    model.train()
    total_loss = 0
    for batch in tqdm(train_loader, desc=f'Epoch {epoch + 1}/{epochs}', unit='batch'):
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['label'].to(device)

        optimizer.zero_grad()
        outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs.loss
        total_loss += loss.item()

        loss.backward()
        optimizer.step()

    average_train_loss = total_loss / len(train_loader)
    print(f"Average training loss: {average_train_loss}")
```

```
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at prajjwal1/bert-tiny an
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Epoch 1/3:   0%|          | 0/2569 [00:00<?, ?batch/s]/usr/local/lib/python3.10/dist-packages/transformers/tokenizatio
  warnings.warn(
Epoch 1/3: 100%|██████████| 2569/2569 [14:35<00:00,  2.93batch/s]
Average training loss: 0.15394362207623935
Epoch 2/3: 100%|██████████| 2569/2569 [15:44<00:00,  2.72batch/s]
Average training loss: 0.0669902520298301
Epoch 3/3: 100%|██████████| 2569/2569 [13:59<00:00,  3.06batch/s]Average training loss: 0.05311315695979223
```

1. Decided on number of epochs (3 in this case).

2. Within each epoch, the model undergoes training iterations over batches of data retrieved from the training loader.

3. During each iteration, the model's predictions are compared with the ground truth labels, and a loss value is computed using the specified loss function (CrossEntropyLoss).

4. Backpropagation is then applied to compute the gradients of the loss with respect to the model's parameters, followed by an optimization step to update the model's weights.

Milestone 3

4. Evaluating the Model

```
model.eval()
total_correct = 0
total_samples = 0
with torch.no_grad():
    for batch in tqdm(test_loader, desc='Evaluating', unit='batch'):
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['label'].to(device)

        outputs = model(input_ids, attention_mask=attention_mask)
        _, predicted = torch.max(outputs.logits, 1)
        total_correct += (predicted == labels).sum().item()
        total_samples += labels.size(0)

accuracy = total_correct / total_samples
print(f"Accuracy: {accuracy}")
```

```
Evaluating: 100%|██████████| 643/643 [01:21<00:00,  7.91batch/s]Accuracy: 0.980533385244306
```

- Our model achieved an accuracy of 0.98, meaning it correctly classified 98% of the inputs.
- This high level of accuracy indicates robust performance in distinguishing between the two classes.

Milestone 3

Using predefined BERT model on Regression task

1. **Data Preparation**
2. **Model Loading**

```
tokenizer = BertTokenizer.from_pretrained('prajjwal1/bert-tiny')
model = BertModel.from_pretrained('prajjwal1/bert-tiny')

# Define model architecture for regression
class RegressionBert(torch.nn.Module):
    def __init__(self, bert):
        super(RegressionBert, self).__init__()
        self.bert = bert
        self.linear = torch.nn.Linear(128, 1)  # Assuming TinyBERT has hidden size of 128

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = outputs.last_hidden_state.mean(dim=1)
        score = self.linear(pooled_output)
        return score
```

```
vocab.txt: 100% 232k/232k [00:00<00:00, 2.60MB/s]
config.json: 100% 285/285 [00:00<00:00, 14.7kB/s]
pytorch_model.bin: 100% 17.8M/17.8M [00:00<00:00, 32.1MB/s]
```

- The model architecture used: RegressionBert
- The model name: ’prajjwal1/bert-tiny’ model

Milestone 3

3. Training the Model

```
for epoch in range(5):  # Train for 5 epochs
    model_regression.train()
    total_loss = 0
    for batch in train_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        target = batch['score'].unsqueeze(1).to(device)

        optimizer.zero_grad()
        output = model_regression(input_ids, attention_mask)
        loss = criterion(output, target)
        total_loss += loss.item()

        loss.backward()
        optimizer.step()

    print(f'Epoch {epoch+1}, Loss: {total_loss}')
```

```
Epoch 1, Loss: 94529.64983081818
Epoch 2, Loss: 92121.84314161539
Epoch 3, Loss: 90424.09765303135
Epoch 4, Loss: 88517.39712738991
Epoch 5, Loss: 86728.92257028818
```

1. A training loop for a regression task over five epochs is executed.

2. The training data is iterated over in batches using the trainloader, which contains preprocessed input sequences (inputids) and their corresponding target scores.

3. The model's optimizer gradients are reset, and the model is passed the input sequences to generate predictions.

4. These predictions are compared with the target scores using a specified loss function (MSE), and the loss is calculated.

5. The gradients of the loss with respect to the model's parameters are computed via backpropagation, and the optimizer (AdamW) adjusts the model's parameters to minimize the loss.

Milestone 3

4. Evaluating the Model

```
[ ]  model_regression.eval()
     test_dataset = CustomDataset(test_texts, test_scores, tokenizer, max_len=128)
     test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)

     predictions = []
     targets = []

     with torch.no_grad():
         for batch in test_loader:
             input_ids = batch['input_ids'].to(device)
             attention_mask = batch['attention_mask'].to(device)
             target = batch['score'].unsqueeze(1).to(device)

             output = model_regression(input_ids, attention_mask)

             predictions.extend(output.cpu().numpy())
             targets.extend(target.cpu().numpy())

     predictions = np.array(predictions).flatten()
     targets = np.array(targets).flatten()

     mse = mean_squared_error(targets, predictions)
     print(f'Mean Squared Error: {mse}')

     Mean Squared Error: 25.6873836517334
```

- Our model has a MSE of 25.6 which means the average squared difference between the predicted and actual scores is 25.6.

Conclusion

In the Binary Classification Task:

- Our predefined BERT model achieved an accuracy of 0.98 in classifying questions as related to Android or iOS.
- The LSTM model achieved an accuracy of 0.25 which did not match the performance of BERT.
- The pretrained BERT model used in Milestone 3 outperforms LSTM model used in Milestone 2 as it considers the entire context of a sentence, both from the left and the right, which enables it to understand the full meaning of each word in context while LSTM model captures dependencies by processing the input sequentially (one word at a time)

Conclusion

In the Regression Task:

- Our predefined BERT model achieved a MSE of 25.68 in predicting the scores of the questions
- The LSTM model achieved a MSE of 43.45
- This significant difference in performance is due the architectural advantages of BERT over LSTM. BERT's transformer architecture processes text bidirectionally, meaning it considers the full context of each word within a sentence from both directions, which allows it to capture intricate relationships and dependencies in the text more effectively while LSTMs process text sequentially and may struggle with capturing the full context as effectively as BERT.
- This sequential processing can lead to less effective handling of long-range dependencies and contextual information, resulting in higher prediction errors and thus a higher MSE in classification tasks.

THANK YOU