NLP Report - Team 13

Omar Aboelazm
Mohammed Assem

May 19, 2024

Abstract

In the realm of Natural Language Processing (NLP), the ability to classify text accurately and effectively has become a fundamental task with widespread applications. From sentiment analysis to information retrieval, the capacity to decipher textual data plays a pivotal role in numerous domains. Our project endeavors to delve into this realm by addressing the task of question classification and answering system, utilizing real-world datasets sourced from two prominent StackExchange forums: Android Enthusiasts and Ask Different (Apple).

1 Introduction

Our selection of the "Android vs. iOS" dataset stemmed from a strategic consideration of relevance, accessibility, and potential for exploration. The dataset offers a rich repository of questions posted by users on both Android and iOS platforms, enabling us to conduct comprehensive analyses and develop sophisticated models. By leveraging this dataset, we aim to tackle a significant challenge in NLP—differentiating between questions pertaining to Android and iOS platforms and providing accurate responses.

1.1 Relevance and Significance:

The ubiquity of mobile technology, particularly the Android and iOS ecosystems, underscores the importance of understanding user queries and preferences within these domains. As mobile devices continue to shape various aspects of modern life, the ability to categorize and address user inquiries accurately holds immense practical value. Our project seeks to contribute to this endeavor by employing advanced NLP techniques to classify questions and build an effective question-answering system.

1.2 Challenges and Opportunities:

The task of question classification and answering system presents several challenges, including handling unstructured text data, mitigating class imbalances, and ensuring model robustness across diverse query types. However, these challenges also represent opportunities for innovation and learning. By navigating through these complexities, we aim to gain insights into the intricacies of text classification and contribute to the development of practical solutions that enhance user experiences in the mobile technology domain.

1.3 Project Objectives:

In the subsequent sections of this report, we will delve into a comprehensive literature review, exploring recent research relevant to our problem domain. Additionally, we will conduct a thorough analysis of the dataset at hand, uncovering insights and potential challenges that will inform our approach in the subsequent milestones.

2 Literature Review

Recent advancements in question answering (QA) systems have showcased the potential of natural language processing (NLP) techniques in automating information retrieval and knowledge dissemination. (Lavanya, 2021). Traditional approaches, primarily relying on keyword matching and syntactic search algorithms, have paved the way for more sophisticated methodologies that prioritize semantic understanding and context-based responses (Beta et. al, 2023) . However, challenges persist, particularly in low-resource languages and specialized domains such as mobile technology platforms like Android and iOS.

2.1 Addressing Challenges in Low-Resource Languages

Das and Saha (2022) address the challenge of building QA systems in low-resource languages like Bengali by employing supervised learning algorithms and leveraging machine-readable dictionaries such as WordNet. Their system achieves high accuracy in question classification and answer retrieval, demonstrating the feasibility of building comprehensive QA systems in languages with limited linguistic resources. The adoption of supervised learning methods and the use of a text corpus as the system's repository underscore the importance of utilizing available resources effectively in overcoming language-specific challenges.

2.2 Harnessing Deep Learning and NLP Techniques

Tzu-Hsuan Lin et al. (2022) leverage deep learning and NLP techniques, specifically the Bidirectional Encoder Representations from Transformers (BERT) model, to develop an intelligent question and answer system tailored to the construction industry's needs. By integrating BERT with a mobile chatbot interface, their system enables conversational machine understanding and facilitates user-friendly information searches in the context of building information modeling and artificial intelligence of things (BIM-AIOT). The utilization of machine learning models and NLP techniques enables accurate prediction and efficient information retrieval, empowering professionals in the Architecture, Engineering, and Construction (AEC) domain to make informed decisions swiftly.

2.3 Pipelines for QA and QC Models

In Natural Language Processing (NLP), several common pipelines and model architectures have emerged as standard approaches for various tasks. One prevalent pipeline involves preprocessing the text data by tokenizing, normalizing, and vectorizing the text before feeding it into a neural network-based model. For classification tasks, Convolutional Neural Networks (CNNs) and recurrent architectures like Long Short-Term Memory (LSTM) networks are widely used due to their ability to capture sequential information effectively.

Attention mechanisms have become a fundamental component in many state-of-the-art models across various fields of natural language processing (NLP). Originally introduced in the context of neural machine translation (Bahdanau et al., 2014), attention mechanisms allow models to focus on relevant parts of the input sequence when making predictions. The key idea is to dynamically weigh different parts of the input sequence, giving higher importance to the more relevant tokens. They have since been adopted in a wide range of NLP tasks, including text classification, language modeling, and question answering. They have also been extended and refined in various ways, such as self-attention mechanisms in transformer models (Vaswani et al., 2017), which enable capturing complex dependencies between tokens in the input sequence.

Another common practice in NLP research is the use of transfer learning and pre-trained language models. Researchers often leverage pre-trained models like BERT (Devlin et al., 2018), GPT (Radford et al., 2018), or RoBERTa (Liu et al., 2019) as starting points for their specific NLP tasks. These pre-trained models are fine-tuned on domain-specific data or downstream tasks to adapt them to the specific task at hand. This approach has proven effective in achieving state-of-the-art results across a wide range of NLP tasks, including text classification, named entity recognition, question answering, and machine translation.

Muhammad Zulqarnain et al. (2021) compared different neural networks architectures for QC, and among the tested layers which where CNN, LSTM and GRU, the combination of CNN layer followed by an GRU layer yielded the best results.

These pipelines and architectures, along with their variations and combinations, form the backbone of modern NLP research and applications.

2.4 Implications for Android vs. iOS Question Classification

While the aforementioned studies focus on diverse domains such as language-specific QA systems and BIM-AIOT integration, their methodologies and insights offer valuable lessons for the task of classifying questions related to Android and iOS support platforms. By adopting supervised learning algorithms, deep learning models, and NLP techniques, researchers can enhance the accuracy and robustness of question classification systems, enabling more effective differentiation between user queries in the mobile technology domain. Furthermore, the utilization of large text corpora and domain-specific knowledge repositories can facilitate the development of comprehensive QA systems tailored to the intricacies of Android and iOS platforms.

2.5 Conclusion

In conclusion, recent advancements in QA systems, exemplified by the works of Das and Saha (2022) and Tzu-Hsuan Lin et al. (2022), demonstrate the transformative potential of deep learning, NLP, and supervised learning techniques in automating information retrieval and knowledge dissemination across diverse domains.

By leveraging these methodologies and insights, researchers can tackle the challenges of question classification in specialized domains such as Android and iOS platforms, paving the way for more accurate and efficient question answering systems tailored to the needs of modern mobile technology users.

3   Data analysis and insights

For the following part, we will be discussing and analyzing all aspects of the dataset, while also extracting as many insights as possible.

3.1   Value analysis

Figure 1: Code Snippet

Figure 2: Code Output

Starting with the basics, we used simple pandas methods to view the dataset dimensions, the first few rows, the data types used and null occurrences.As shown, the dataset has 51370 entries and 7 columns. Luckily, this dataset is clean from null values.

The following is just a clearer sample of the dataset.

Figure 3: Data-frame Head

As seen, the seven columns are:

- Id: question id, each question has a unique value.

- Title: title of the question asked.

- Body: questions body related to the title.

- Score: score assigned to the the questions.

- ViewCount: number of views the question got.

- Label: classification label, whether the question is related to android or ios

- LabelNum: a one-hot encoding of the "Label" column, basically a binary classification indicator.

Also, no duplicates are present in the dataset.


```python
# Check for duplicates
duplicate_rows = data_train[data_train.duplicated()]
print("Number of duplicate rows:", len(duplicate_rows))
```
0.1s   Python

Number of duplicate rows: 0

Figure 4: Duplicate Check

So, for the primary analysis, it is clear that the dataset doesn't contain any troubling entries.

3.2 Exploratory Data Analysis (EDA)

The exploratory data analysis will include analysis of different columns and visualization to help drive all useful insights that can be captured from the dataset.

- **Label distribution**


```python
import matplotlib.pyplot as plt

# Distribution of labels
label_distribution = data_train['Label'].value_counts()
print("Distribution of labels:")
print(label_distribution)

# Pie chart of label distribution
plt.figure(figsize=(6, 6))
plt.pie(label_distribution, labels=label_distribution.index, autopct='%1.1f%%')
plt.title('Distribution of Labels')
plt.show()
```
0.0s   Python
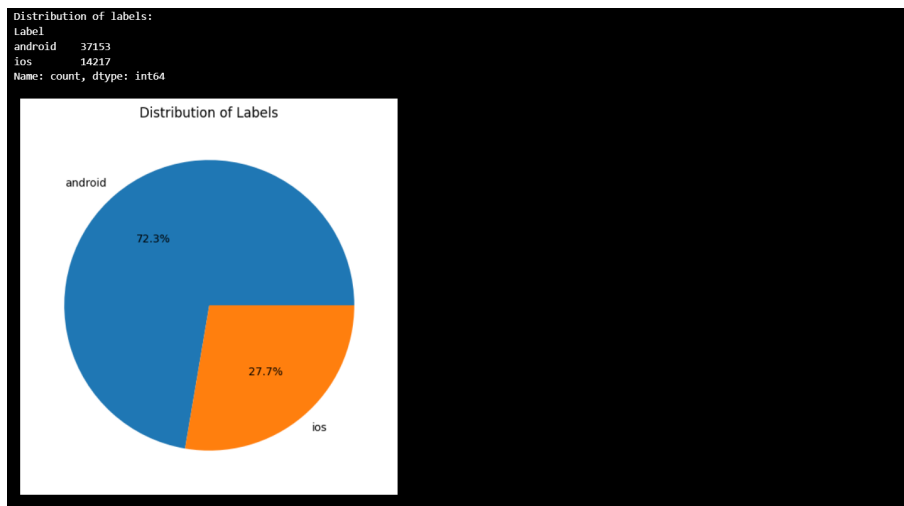
Figure 5: Code Snippet

Figure 6: Label Distribution

The dataset holds 37153 questions related to android (72.3%) and 14217 questions related to ios (27.7%) , which clearly shows that the dataset is biased towards android questions. This bias is logical because per the latest statistics provided by Statista , android market share is 70.11% while IOS is 29.19%.
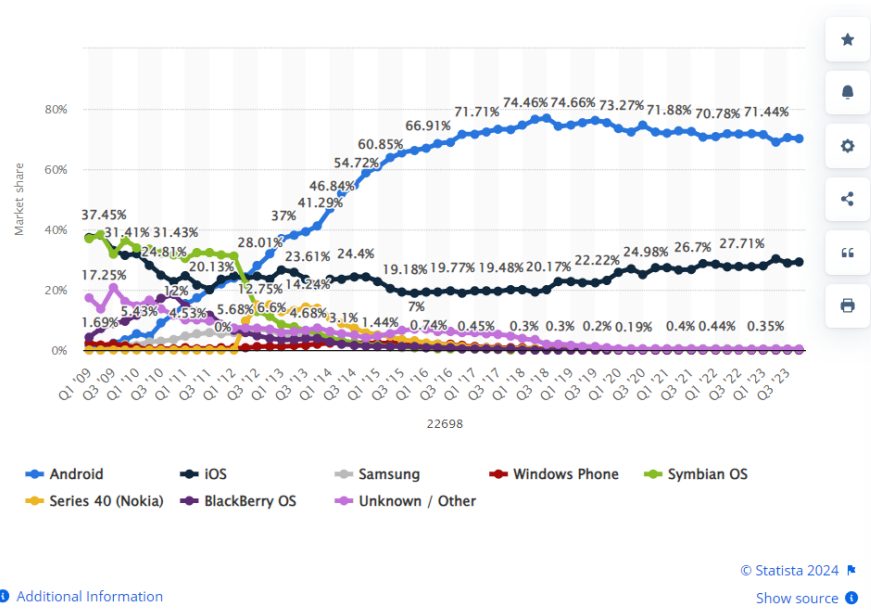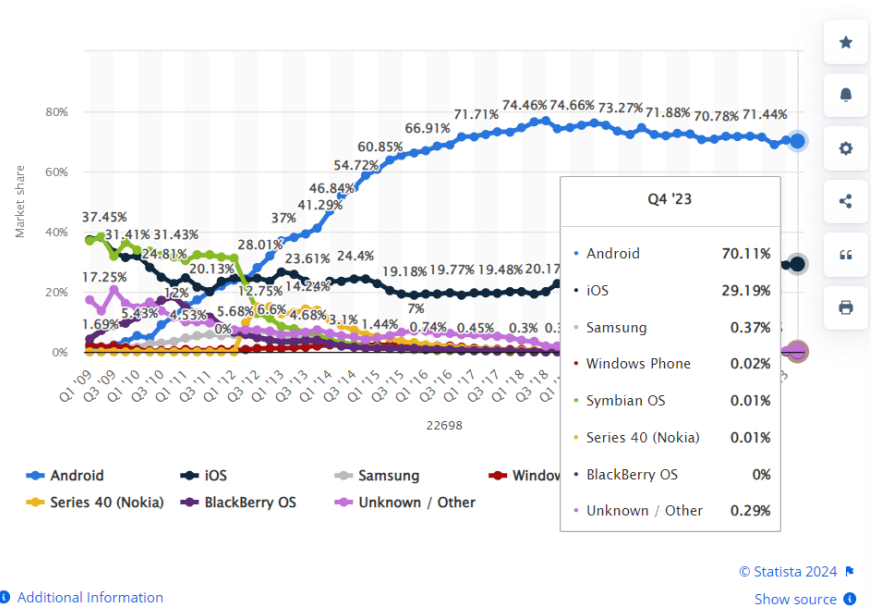
4

Figure 7: Statista Data

Figure 8: Statista Data

But, this bias is to be considered during the training of the model.

- Score distribution

```
import seaborn as sns

# Define function to remove outliers based on IQR
def remove_outliers_iqr(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return data[(data[column] >= lower_bound) & (data[column] <= upper_bound)]

# Visualize distribution of 'Score' before removing outliers
plt.figure(figsize=(8, 6))
sns.boxplot(x=data_train['Score'])
plt.title('Distribution of Score (Before Removing Outliers)')
plt.xlabel('Score')
plt.show()

# Remove outliers from 'Score' column
data_no_outliers = remove_outliers_iqr(data_train, 'Score')

# Visualize distribution of 'Score' after removing outliers
plt.figure(figsize=(8, 6))
sns.boxplot(x=data_no_outliers['Score'])
plt.title('Distribution of Score (After Removing Outliers)')
plt.xlabel('Score')
plt.show()
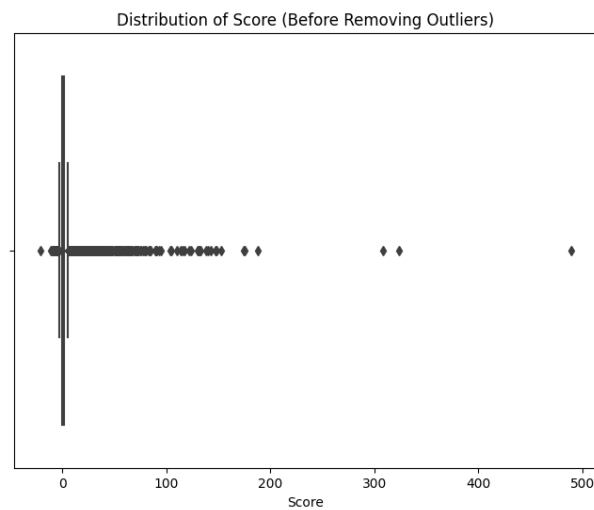```

Figure 9: Box-plot Distribution
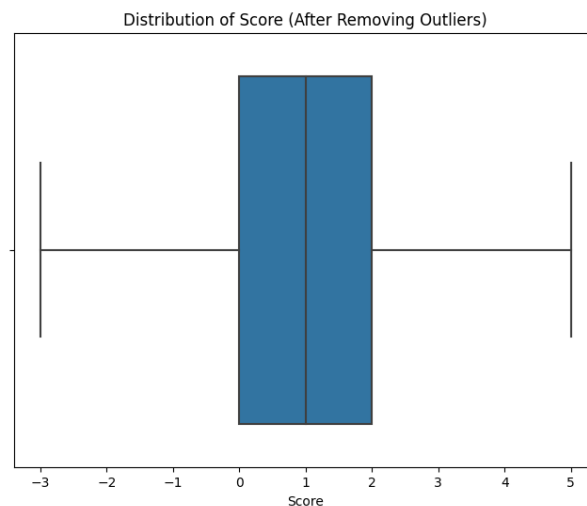


Figure 10: With Outliers



Figure 11: Without Outliers

The first box-plot shows the distribution of score before removing outliers, and the right box-plot shows the distribution of values after removing outliers. Outliers were removed using the Interquartile Range (IQR) method. It is a measure of statistical dispersion, or spread, which is used to identify the extent of spread in the middle 50% of a dataset. The IQR is calculated as the difference between the third quartile (Q3) and the first quartile (Q1) of the data.
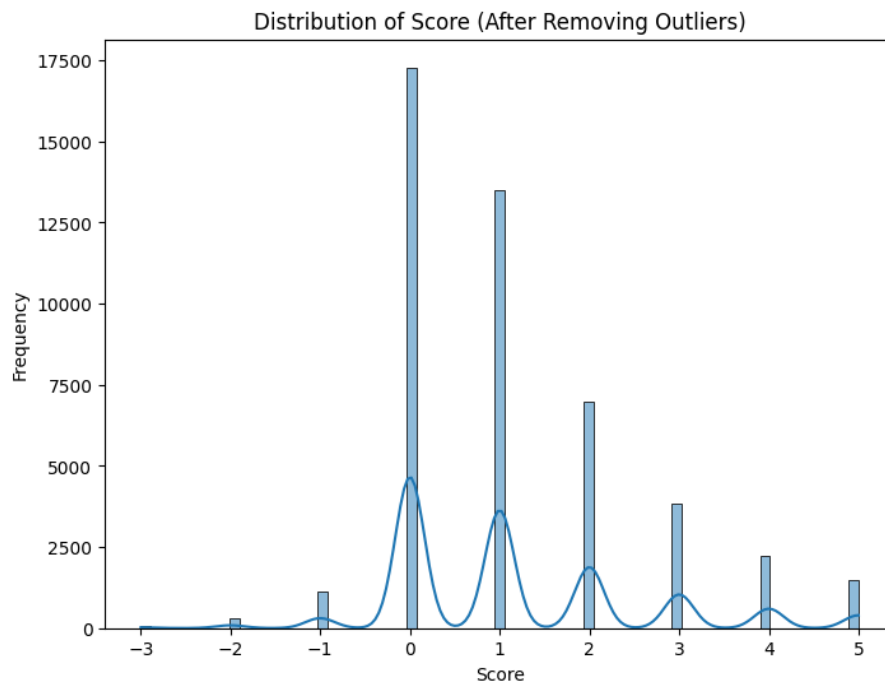
Figure 12: Histogram Without Outliers

The above histogram visualizes the distribution of scores after removing outliers.

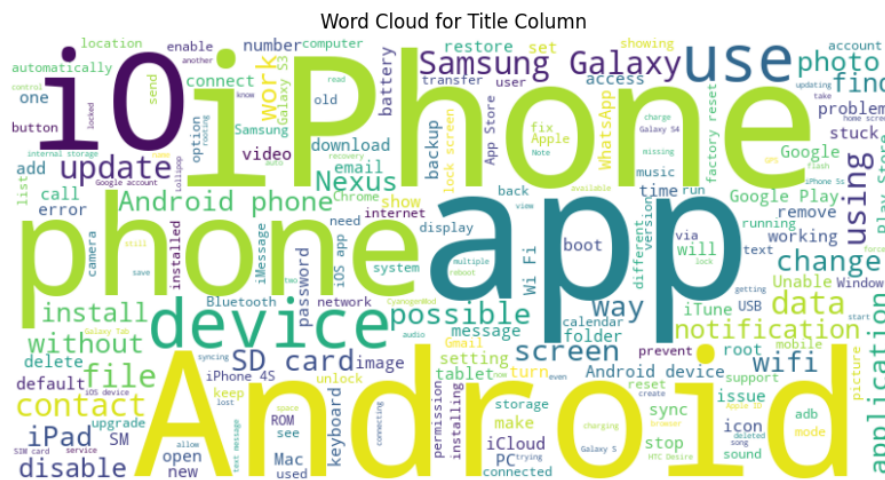- **Word-cloud**



Figure 13: General WordCloud



Figure 14: Specific WordCloud

Word-cloud was used in the first image to visualize the most used words in the "Title" column, and then re-used to visualize the words related to each OS class separately.

- **Sentiment polarity analysis**

```python
from textblob import TextBlob

# Compute sentiment polarity for each title
data_train['Title_Sentiment'] = data_train['Title'].apply(lambda x: TextBlob(x).sentiment.polarity)

# Plot distribution of sentiment polarity
plt.figure(figsize=(8, 6))
sns.histplot(data_train['Title_Sentiment'], bins=20)
plt.title('Distribution of Title Sentiment Polarity')
plt.xlabel('Sentiment Polarity')
plt.ylabel('Frequency')
plt.show()
```

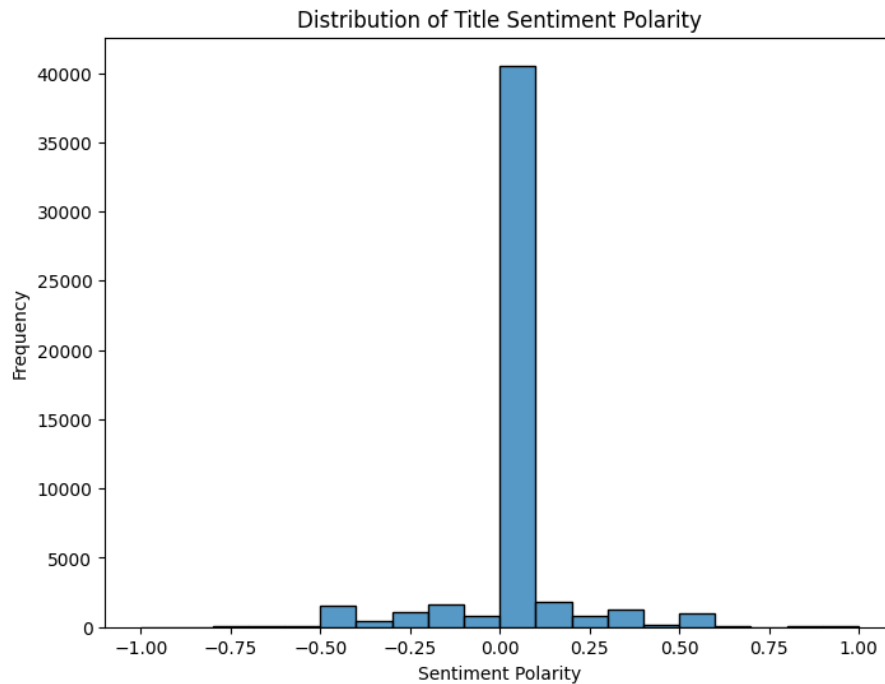Figure 15: Sentiment Analysis

Figure 16: Sentiment Distribution

The above plot shows that most questions have a sentiment polarity of 0, which is neutral. That is logical as these questions are mostly technical.

- **Topic classification**

```python
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.text import CountVectorizer

# Vectorize text data
vectorizer = CountVectorizer(max_features=1000, stop_words='english')
X = vectorizer.fit_transform(data_train['Title'])

# Apply LDA
lda = LatentDirichletAllocation(n_components=5, random_state=42)
lda.fit(X)

# Display top words for each topic
feature_names = vectorizer.get_feature_names_out()
for topic_idx, topic in enumerate(lda.components_):
    print(f"Topic {topic_idx+1}:")
    print(" ".join([feature_names[i] for i in topic.argsort()[:-10 - 1:-1]]))
    print()
```

Figure 17: Topic Classification

Figure 18: Topics

Figure 19: Topics

We utilized Latent Dirichlet Allocation (LDA) to uncover latent topics within text data. We first vectorized the text using CountVectorizer, then trained the LDA model to identify topics based on word distributions. Finally, we displayed the top words associated with each topic, providing insight into the main themes captured by the model.

3.3 Data pre-processing

Lowercasing

Figure 20: Lowercasing

The lowercasing is done to normalize the words for future tokenization and model training.

3.4 Tokenization

Figure 21: Tokenization Process

Figure 22: Tokenization Output

Tokenization is done to the dataset after lowercasing, it produces a list of all the words used in the dataset.

3.5 Removing punctuation
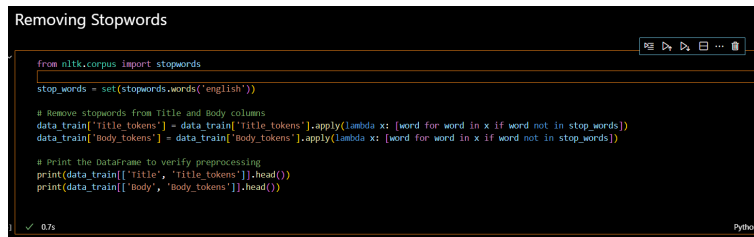
Figure 23: Removing punctuation

Cleaning the tokens by removing punctuations.

3.6 Removing Stopwords

Figure 24: Removing Stopwords

Cleaning the tokens by removing stop words that will not add much to the context.

3.7 Stemming / Lemmatization

Figure 25: Stemming / Lemmatization

Returning the tokens to their stem word.

3.8 Joining tokens

Figure 26: Joining tokens

Reconstructing the sentences by joining the final tokens.

3.9 Token Analysis
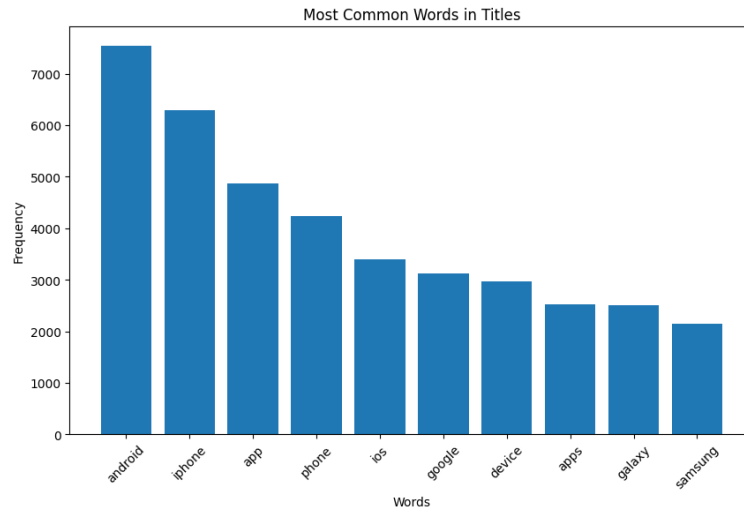
Figure 27: Title tokens
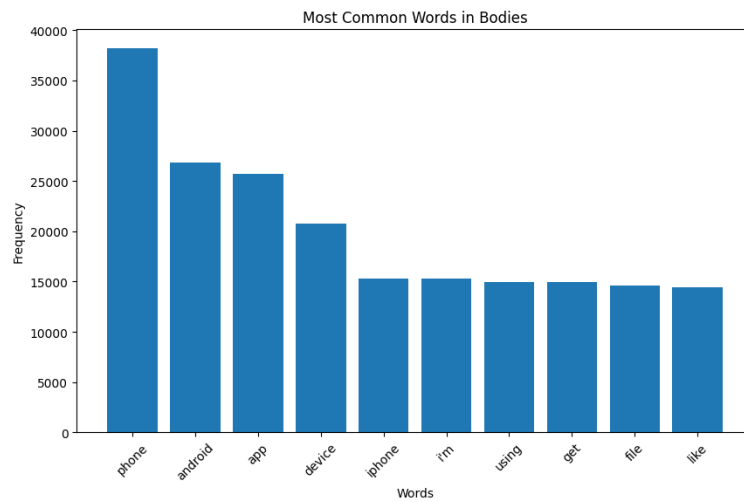
Figure 28: Body tokens

After applying all the cleaning methods for tokens, the above histogram visualizes the most frequent tokens in both the Body and Title columns. Which shows a successful tokenization as these words are mostly associated with technical questions related to android and ios.

3.10 Critical Insights

1. Imbalance in Android vs. iOS Content: The dataset heavily leans towards Android-related questions, reflecting the dominance of Android in the market compared to iOS. This skew raises concerns about potential differences in user interests or available information, highlighting the need for careful model training to maintain balance.

2. Effect of Extreme Values on Scores: Removing outliers significantly changes the distribution of scores, indicating that extreme values play a significant role in shaping overall trends. Understanding these outliers' nature and impact is crucial for accurate modeling and interpretation.

3. Digging Deeper into Word Frequency: While word clouds provide initial insights into common themes, a deeper analysis of word frequency could reveal underlying patterns and topics. Uncovering and understanding these subtleties can enhance analysis and model performance.

4. Exploring Neutral Sentiment: The predominance of neutral sentiment reflects the dataset's technical nature. However, examining sentiment fluctuations and their relationship with user engagement could uncover valuable insights for optimizing content and enhancing user satisfaction.

5. Ensuring Data Integrity through Pre-processing: Thorough pre-processing techniques, such as advanced tokenization to maintain sequence integrity and handling contractions, are essential for preserving data accuracy and minimizing information loss. Multiple problems were encountered as contraction tokenization and version tokenization (e.g. 4.4.2). Which needed special and advanced tokenization.

In summary, examining label biases, understanding outlier effects, conducting nuanced word frequency analysis, exploring sentiment trends, and employing meticulous pre-processing practices are crucial for extracting valuable insights and ensuring reliable model performance in real-world datasets.

4 Classification Model

Initially, the task at hand involved classifying text data into binary categories (1 = IOS & 0 = ANDROID) using a shallow neural network that was not pre-trained. The dataset consisted of textual information of questions related to both categories split into titles and bodies, each associated with a numerical label that identified the class. The goal was to predict these labels based on the textual content.

4.1 Data Preprocessing

Starting with data pre-processing, the textual data was tokenized using the Tokenizer module from Keras, to transform the string tokens into sequence of integers. Both title and body data were combined for each entry, creating a single sequence which served as a wider feature for learning the class. This combined sequence was then split into training and testing sets using a standard 80-20 split. Additionally, padding was applied to ensure uniform sequence length for model input, with a maximum sequence length of 100 tokens.


```
Raw data tokens:  ['twrp', 'recovery', 'android', 'doubt', 'twrp', 'recovery', 'tool', 'developed', 'smartphone', 'specifically', 'even', 'almost',
Numerically embedded data tokens:  [270, 89, 2, 2180, 270, 89, 383, 2458, 609, 903, 52, 586, 123, 2, 1]
Padded data tokens:  [ 270   89    2 2180  270   89  383 2458  609  903   52  586  123    2
    1    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0]
```

Figure 29: Tokenization and Embedding

Also the dataset at hand, has an imbalance between the classes as previously discussed. To address this imbalance, class weights were computed and incorporated into the model training process using the `compute_class_weight` function from scikit-learn. These class weights were utilized to provide higher importance to minority class samples during training, thus mitigating the imbalance issue.
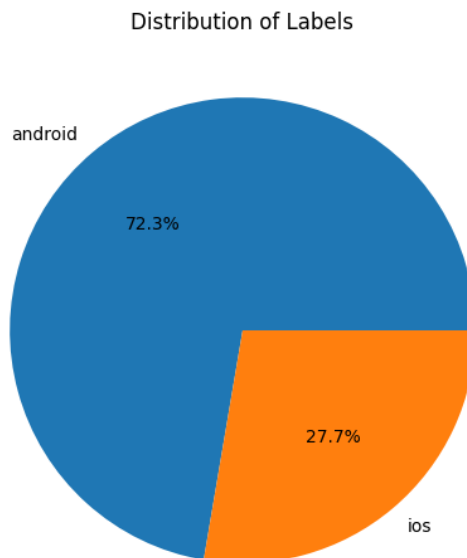
Figure 30: Label Distribution

4.2   Model Architecture

Subsequently, we chose an RNN model. First, a simple RNN model from Keras was used to limit the computational impact on our devices as we are dealing with a medium sized dataset. But, it didn't preform well and only reached an accuracy of 52%

Then, a Long Short-Term Memory (LSTM) network, which is more complex, was used. The model architecture included an embedding layer for word representation, followed by an LSTM layer for sequential processing, and a dense layer with sigmoid activation for binary classification.

Lastly, a more complicated and dynamic architecture was created with one embedding layer, three bidirectional LSTM layers and a dense layer. The input is embedding generated using Word2Vec embeddings to try and achieve a better accuracy.

The LSTM and BI-LSTM had the same accuracy of 98% but the BI-LSTM had slightly better precision, recall and F1-Score values. Which indicates it has lower false positives and false negatives.

So, after multiple model architectures we sticked with the BI-LSTM NN.

```python
# Word2Vec model for embeddings
word2vec_model = Word2Vec(sentences=X_combined, vector_size=embedding_dim, window=5, min_count=1, workers=4)

# Model architecture
class_model = Sequential()

# Embedding layer
class_model.add(Embedding(input_dim=len(class_tokenizer.word_index) + 1,
                          output_dim=embedding_dim,
                          weights=[np.vstack([np.zeros(embedding_dim), word2vec_model.wv.vectors])],
                          input_length=maxlen, trainable=False))

# Dynamic sized bidirectional LSTM layers
for _ in range(num_lstm_layers):
    class_model.add(Bidirectional(LSTM(units=lstm_units, dropout=0.2, recurrent_dropout=0.2, return_sequences=True)))

# Final bidirectional LSTM layer
class_model.add(Bidirectional(LSTM(units=lstm_units, dropout=0.2, recurrent_dropout=0.2)))

# Dense layer with sigmoid AF
class_model.add(Dense(1, activation='sigmoid'))

class_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Figure 31: Architecture 2

4.3   Model Training & Evaluation

The model was compiled with the Adam optimizer and binary cross-entropy loss function. Adam optimizer was used due its dynamic nature and robustness.

The BI-LSTM model was trained using a batch size of 32 and for a specified number of epochs (in this case, 10 epochs). During training, a validation split of 10% was utilized for monitoring model performance and preventing overfitting. Following training, the model was evaluated on the test set to assess its accuracy in predicting binary labels.

```python
# Hyperparameters
maxlen = 100
embedding_dim = 300   # Dimensionality of Word2Vec embeddings
lstm_units = 128
num_lstm_layers = 2   # Number of BI-LSTM layers
batch_size = 32
epochs = 10
```

Figure 32: Hyperparameters

Model performance was evaluated using various metrics, including accuracy and confusion matrix analysis. The confusion matrix provided insights into the model's classification performance by visualizing true positive, true negative, false positive, and false negative predictions.

Class	Precision	Recall	F1-Score	Support
0	0.98	0.99	0.99	7318
1	0.97	0.96	0.97	2956
Accuracy			0.98	10274
Macro Avg	0.98	0.98	0.98	10274
Weighted Avg	0.98	0.98	0.98	10274

Table 1: Classification report
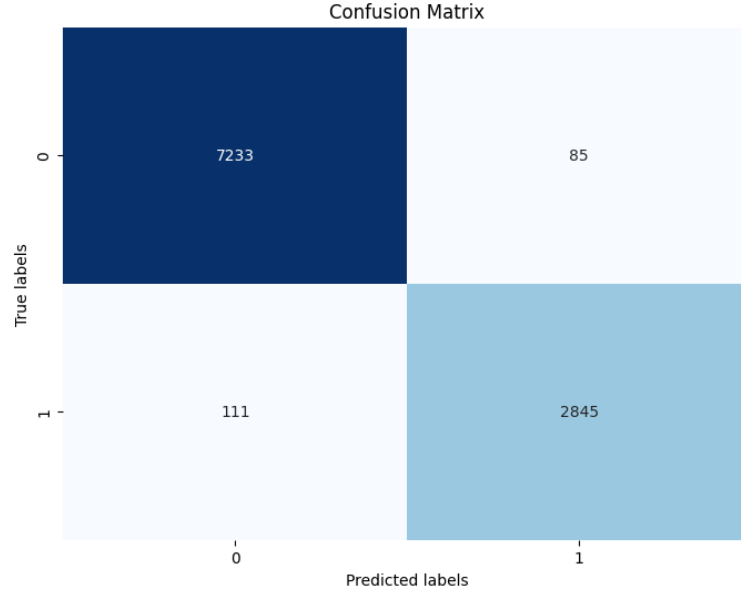
Figure 33: Confusion Matrix

Due to handling the imbalances and the appropriate size of the dataset, acceptable accuracies were achieved by the BI-LSTM Network. The model was then tested on custom user inputs to validate its usage.


```
user_input = input("Enter: ")

user_input_seq = class_tokenizer.texts_to_sequences([user_input])

user_input_pad = pad_sequences(user_input_seq, maxlen=100, padding='post', truncating='post')

print("User input: ", user_input)

predicted_label = class_model.predict(user_input_pad)

predicted_label_binary = 1 if predicted_label > 0.5 else 0
predicted_label_st = "Android" if predicted_label_binary == 0 else "IOS"


print("Predicted Label: ", predicted_label_st)
✓ 6.3s
User input:  I love samsung but have an iphone
1/1 [==============================] - 0s 37ms/step
Predicted Label:  IOS
```

Figure 34: User Input

Figure 35: User Input

4.4 Conclusion

Summing up the work, we successfully created and trained a classification model able to classify a given question to either relating to IOS or Android. The features where the title and body of questions asked on Stack Exchange and the target feature was the label given to these questions by users on the same platform.

After testing multiple architectures, it was clear that the BI-LSTM was fit for the task at hand. The only drawbacks were that the model was not pre-trained and didnt have any data labeled as "Irrelevant". This will cause the model to ultimately predict a label of "IOS" or "Android" for any input even if it was completely irrelevant to both subjects. Yet, the model had a satisfactory performance with both labels.

Future work will include using a pre-trained model and trying more complex architectures and further tune the hyper-parmeters to achieve better performance.

5 Regression Model

In this project, our objective was to develop a regression model capable of accurately predicting scores based on the textual content from the questions of IOS Vs Android dataset. The dataset used for this task contains various features such as the title, body, view count, and score of each question. Our goal was to leverage natural language processing (NLP) techniques and machine learning algorithms to accurately predict the score of a question based on its textual content. This report provides a detailed overview of the data preprocessing steps, model architecture, training process, evaluation metrics, and rationale behind key design choices.

5.1 Data Preprocessing

The initial phase of the project involved thorough data preprocessing to ensure the quality and reliability of the dataset. We performed an exploratory analysis to identify any inconsistencies or anomalies within the data after and including some of that was held in milestone 1. One crucial concern we had was the presence of entries with a view count of 0 but a positive score, which contradicted typical behavior. To maintain data integrity, we deemed these entries as invalid and removed them from the dataset to prevent potential noise during model training. Fortuantely, when we checked the dataset we found no such entries.

Figure 36: Invalid Entries

16

5.2 Outlier Removal

Addressing outliers in the score column was another essential preprocessing step that we witnseed during EDA in milestone 1. Outliers can significantly impact the performance of regression models, leading to biased predictions. The original dataset before doing EDA and preprocessing in milestone 1 had a range of scores [-21.0 , 489.0] which would negatively affect the performance of our model. To mitigate this issue, we employed the Interquartile Range (IQR) method to identify and filter out outliers from the score column. By calculating quartiles and defining upper and lower bounds, we ensured that the model learned from a more representative and reliable dataset, enhancing prediction accuracy. The new scores range became [-3,5]

Now the model can perform much better being more balanced.

5.3 Model Architecture

For the regression model architecture, we opted for a deep learning approach, specifically utilizing a Bidirectional Long Short-Term Memory (BiLSTM) network. The choice of BiLSTM was motivated by its ability to capture long-range dependencies and sequential patterns in textual data. Bidirectional LSTMs process sequences in both forward and backward directions, enabling the model to effectively capture contextual information from surrounding words.

5.3.1 Word Embeddings

To represent textual data numerically, we utilized word embeddings generated using the Word2Vec algorithm. Word embeddings encode semantic relationships between words by mapping them to dense vector representations in a continuous vector space. We chose Word2Vec over other methods like BERT due to its simplicity, efficiency, and effectiveness in capturing semantic information from large text corpora. The decision to use 100 as the embedding dimension was based several trials. A higher-dimensional embedding space allows for a richer representation of semantic nuances and improves the model's ability to learn complex patterns in the data.

```python
from sklearn.utils.class_weight import compute_class_weight

# Tokenize and pad sequences
X_title = reg_data['Title_tokens'].apply(lambda x: eval(x)).values
X_body = reg_data['Body_tokens'].apply(lambda x: eval(x)).values
X_combined = [title + body for title, body in zip(X_title, X_body)]

X_train, X_test, y_train, y_test = train_test_split(X_combined, reg_data['Score'].values, test_size=0.2, random_state=42)

reg_tokenizer = Tokenizer()
reg_tokenizer.fit_on_texts(X_combined)

X_train_seq = reg_tokenizer.texts_to_sequences(X_train)
X_test_seq = reg_tokenizer.texts_to_sequences(X_test)

maxlen = 100
X_train_pad = pad_sequences(X_train_seq, maxlen=maxlen, padding='post', truncating='post')
X_test_pad = pad_sequences(X_test_seq, maxlen=maxlen, padding='post', truncating='post')

# Word2Vec model for embeddings
embedding_dim = 100  # Dimensionality of Word2Vec embeddings
word2vec_model = Word2Vec(sentences=X_combined, vector_size=embedding_dim, window=5, min_count=1, workers=4)
```

Figure 37: Word Embedding for the regression task

5.3.2 Building Architecture and Training model

The model training process involved tokenizing and padding the combined title and body text data to a fixed length. We split the dataset into training and testing sets to evaluate the model's performance objectively. During training, we optimized the model using the Adam optimizer with mean squared error (MSE) as the loss function. We opted for the Adam optimizer coupled with mean squared error (MSE) as the loss function for several reasons. Adam's adaptive learning rates provide efficient convergence by dynamically adjusting learning rates for each parameter. This optimizer's versatility and robustness across different datasets and model architectures make it a reliable choice. Additionally, MSE is a natural fit for regression tasks, offering smooth differentiability and sensitivity to errors, crucial for optimizing neural networks to predict continuous numeric values accurately. These choices were made to ensure efficient optimization and effective performance in training our regression model. The training process spanned multiple epochs, with the model's performance monitored on a validation set to prevent overfitting and ensure generalization. We used 1 LSTM layer with 64 LSTM units. We opted for one LSTM layer and 64 LSTM units for several reasons. Firstly, using multiple LSTM layers caused the model to learn hierarchical representations of the input data, capturing both local

and global dependencies in the text, however the complexity reduced the performance of the model. Secondly, choosing 64 LSTM units strikes a balance between model complexity and computational efficiency. With 64 units, the model can capture a diverse range of patterns and features in the input sequences without overly increasing the computational burden. This choice aims to achieve a good trade-off between model capacity and training efficiency, ensuring effective learning while managing computational resources effectively.

We tried adding an activation function in the dense layer. The function is linear since our task is regression. In regression tasks, where the goal is to predict continuous numeric values, the linear activation function is commonly used in the output layer. The main reason for using the linear activation function is that it allows the model to output any real number within the given range without constraining the output values to a specific range like [0, 1] in the case of sigmoid activation or [0, 1] in the case of softmax activation. However, upon several trials, we found out the model performs worse bu outputting only 1 value for all the tests. So we decided to move without it.

We were concerned about overfitting due to the distribution of our scores which were mostly 1, thus we mitigated overfitting by adding dropout layers to the model architecture. Dropout layers were added to the LSTM layers in the model architecture. For each LSTM layer, the dropout parameter was set to 0.2, meaning that during training, 20% of the input units to the LSTM layers were randomly set to zero.

```python
# Model architecture
reg_model = Sequential()

# Embedding layer
embedding_layer = Embedding(input_dim=len(reg_tokenizer.word_index) + 1,
                            output_dim=embedding_dim,
                            input_length=maxlen, trainable=False)

embedding_layer.build((None,))
embedding_layer.set_weights([np.vstack([np.zeros(embedding_dim), word2vec_model.wv.vectors])])

reg_model.add(embedding_layer)

# Dynamic sized bidirectional LSTM layers
num_lstm_layers = 0  # Number of LSTM layers
lstm_units = 64
for _ in range(num_lstm_layers):
    reg_model.add(Bidirectional(LSTM(units=lstm_units, dropout=0.2, recurrent_dropout=0.2, return_sequences=True)))

# Final LSTM layer
reg_model.add(Bidirectional(LSTM(units=lstm_units, dropout=0.2, recurrent_dropout=0.2, return_sequences=False)))

# Dense layer for regression
reg_model.add(Dense(1))

# Compute class weights
class_weights = compute_class_weight('balanced', classes=np.unique(y_train), y=y_train)
class_weights_dict = {i: w for i, w in enumerate(class_weights)}

reg_model.compile(optimizer='adam', loss='mean_squared_error', metrics=['mean_squared_error'])

# Train the model
batch_size = 32
epochs = 3
reg_model.fit(X_train_pad, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)

# Evaluate the model
loss, mse = reg_model.evaluate(X_test_pad, y_test)
print("Mean Squared Error:", mse)
```

Figure 38: Regression Model Architecture

5.3.3 Model Evaluation

Upon training and evaluation, we assessed the model's performance using a range of performance metrics, including mean squared error (MSE), mean absolute error (MAE), root mean squared error (RMSE), R-squared (R^2) coefficient, and Pearson correlation coefficient. The evaluation results demonstrated a significant improvement in model performance after outlier removal, with reduced errors and enhanced correlations between predicted and actual scores.

Figure 39: Regression Evaluation


```python
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import math

# Make predictions
y_pred = reg_model.predict(X_test_pad)

# Calculate Mean Absolute Error (MAE)
mae = mean_absolute_error(y_test, y_pred)
print("Mean Absolute Error:", mae)

# Calculate Root Mean Squared Error (RMSE)
rmse = math.sqrt(mean_squared_error(y_test, y_pred))
print("Root Mean Squared Error:", rmse)

# Calculate R-squared (R^2)
r_squared = r2_score(y_test, y_pred)
print("R-squared (R^2):", r_squared)

# Calculate Pearson correlation coefficient
correlation_coefficient = np.corrcoef(y_test, y_pred.squeeze())[0, 1]
print("Pearson Correlation Coefficient:", correlation_coefficient)
```

Figure 40: Regression Evaluation 2

Figure 41: Regression Evaluation 3

This training process shows a gradual decrease in the mean squared error (MSE) across the epochs, indicating that the model is learning and improving over time. The initial MSE of approximately 1.94 decreases to around 1.81 by the end of the third epoch. However, when evaluating the model on the validation data, the MSE slightly increases to approximately 1.88, suggesting that the model may be slightly overfitting to the training data. While the overall trend indicates that the model is learning and improving, further analysis may be required to address potential overfitting issues and optimize the model's performance.

To mitigate overfitting more, we already had a dropout of 0.2, but we decided to apply Early Stopping also.

```
# Train the model
batch_size = 32
epochs = 3
# Define EarlyStopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

# Train the model with EarlyStopping callback
reg_model.fit(X_train_pad, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1, callbacks=[early_stopping])
```

Figure 42: Early Stopping

We implemented early stopping during model training to monitor the validation loss and stop training when it starts to increase, indicating overfitting.

```
  warnings.warn(
Epoch 1/3
1053/1053 ——————————— 234s 214ms/step - loss: 1.9463 - mean_squared_error: 1.9463 - val_loss: 1.7996 - val_mean_squared_error: 1.7996
Epoch 2/3
1053/1053 ——————————— 195s 185ms/step - loss: 1.8421 - mean_squared_error: 1.8421 - val_loss: 1.7925 - val_mean_squared_error: 1.7925
Epoch 3/3
1053/1053 ——————————— 180s 171ms/step - loss: 1.7987 - mean_squared_error: 1.7987 - val_loss: 1.7843 - val_mean_squared_error: 1.7843
293/293 ——————————— 14s 47ms/step - loss: 1.9092 - mean_squared_error: 1.9092
Mean Squared Error: 1.9010637998580933
```

Figure 43: Early Stopping Result

However, The addition of early stopping has resulted in a slight increase in the mean squared error (MSE) compared to the previous result without early stopping. While early stopping can prevent overfitting by stopping the training process when the model starts to overfit to the training data, it can also terminate training before the model reaches its optimal performance. In this case, early stopping may have stopped the training process prematurely, leading to a slightly higher MSE. Adjusting the patience parameter of the early stopping callback or trying different combinations of hyperparameters may help in achieving better results. Additionally, evaluating the model's performance on a separate validation set can provide insights into its generalization capabilities and help in fine-tuning the model architecture and training process.

Thus, we decided to continue without it.

5.3.4   Results and Performance Metrics

Metric	Value
Mean Squared Error	1.9077458381652832
Mean Absolute Error	1.0839211270658935
Root Mean Squared Error	1.3812117203683794
R-squared (R^2)	0.04285148909237679
Pearson Correlation Coefficient	0.21123497273298147

Table 2: Evaluation Metrics

The Mean Squared Error (MSE) of approximately 1.91 in our regression task signifies the average squared deviation between the predicted scores and the actual scores in our dataset. While there isn't a universally defined threshold for what constitutes a "good" MSE value, its interpretation hinges on various factors. Comparatively analyzing our MSE against alternative models or baseline performance metrics is crucial to ascertain the effectiveness of our model. Additionally, considering the scale of our target variable, which in our case pertains to scores, aids in contextualizing the significance of the MSE value. We should also assess whether this MSE aligns with our application's requirements for prediction accuracy. It's important to supplement our analysis with other evaluation metrics like Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R-squared (R^2) to gain a comprehensive understanding of our model's performance. Ultimately, while MSE offers valuable insights into prediction errors, its interpretation should be nuanced and accompanied by a holistic assessment of our model's efficacy.

Given that the range of score values in our dataset spans from -3 to 5, a Mean Squared Error (MSE) of approximately 1.91 indicates that, on average, the squared difference between our model's predictions and the actual scores falls within this range. While the MSE value is an important indicator of prediction accuracy, its interpretation should be contextualized within the scale of the target variable. In our case, the MSE suggests that the model's predictions are relatively close to the actual scores, considering the range of possible values. However, further analysis using other evaluation metrics such as Mean Absolute Error (MAE) and Root
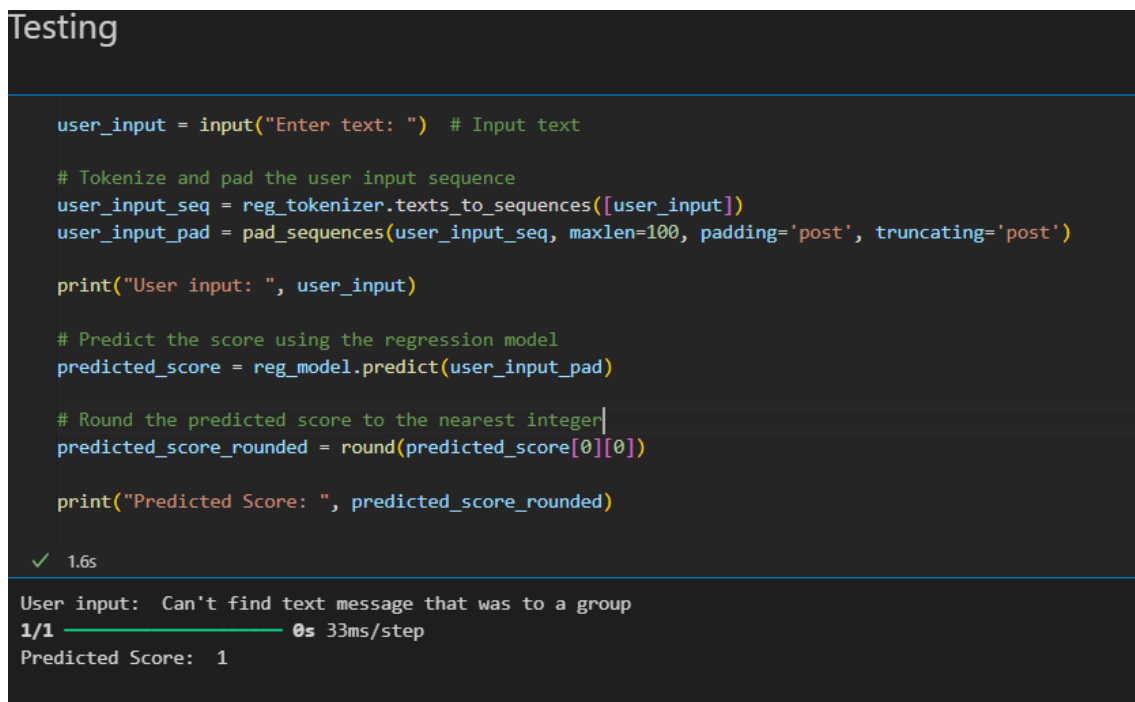
Mean Squared Error (RMSE) would provide additional insights into the model's performance, particularly in understanding the magnitude of prediction errors across the score range.

The MAE and RMSE values are relatively low, indicating that the model's predictions are close to the actual scores on average. The R-squared value of approximately 0.042 indicates that the model explains a small portion of the variance in the data, suggesting that there may still be room for improvement. However, the Pearson correlation coefficient of approximately 0.211 indicates a moderate positive correlation between the predicted and actual scores.

Overall, while the model shows promising results with reduced errors and improved correlations, further analysis and potential enhancements could be beneficial to improve its performance further. Therefore, the current model could be used as a reference point since as mentioned we care more about the architecture than the performance in our milestone.

5.3.5 Testing the Model

To validate the model's functionality, we implemented a testing mechanism that allowed users to input text representing a question. The model then predicted the corresponding score based on the provided text. Predicted scores were rounded to the nearest integer to align with the discrete nature of scores, ensuring practical usability and interpretability.


```
Testing

user_input = input("Enter text: ")  # Input text

# Tokenize and pad the user input sequence
user_input_seq = reg_tokenizer.texts_to_sequences([user_input])
user_input_pad = pad_sequences(user_input_seq, maxlen=100, padding='post', truncating='post')

print("User input: ", user_input)

# Predict the score using the regression model
predicted_score = reg_model.predict(user_input_pad)

# Round the predicted score to the nearest integer
predicted_score_rounded = round(predicted_score[0][0])

print("Predicted Score: ", predicted_score_rounded)

✓  1.6s

User input:  Can't find text message that was to a group
1/1 ─────────────── 0s 33ms/step
Predicted Score:  1
```

Figure 44: Regression Task Testing

The actual score for this question was 1. Note : many results will be 1 due to the imbalanced data since most questions had this score. However, we removed outliers and we mitigated overfitting using the previously mentioned strategies.

5.4 Conclusion and Future Work

In conclusion, we successfully developed a regression model capable of predicting scores based on textual data with improved accuracy and reliability. Through meticulous data preprocessing, thoughtful model architecture design, and comprehensive evaluation, we addressed key challenges associated with score prediction tasks in online forums. Future work could explore alternative architectures, incorporate additional features, and investigate advanced NLP techniques to further enhance predictive performance and model interpretability. Overall, the developed regression model represents a valuable tool for analyzing and understanding question scores in online community platforms.

6 Classification Task Using Pre-trained Model

For the classification task using pre-trained models, we'll leverage the power of transformers, specifically DistilBERT and RoBERTa, for text classification. As before, our task is to classify text as related to IOS or ANDROID topic, but this time we will use a pre-trained transformer and use transfer learning to make the model familiar with our dataset.

These models have been pre-trained on large corpora and have shown excellent performance in various natural language processing tasks.

We have settled with DistilBERT and RoBERT as most of the BERT transformers yielded similar values but differed greatly in training time due to their complexity.

6.1 DistilBERT Model

DistilBERT is a distilled version of BERT, resulting in a smaller model size and faster inference times compared to BERT. DistilBERT achieves this by using a smaller architecture and fewer parameters while maintaining relatively good performance.

As we have already cleaned and pre-processed the data, we begin by loading the data and tokenizing it using the DistilBERT tokenizer. The dataset consists of textual information of questions split into titles and bodies, each associated with a numerical label indicating the class (1 for IOS and 0 for ANDROID). We combine the title and body for each entry to create a single sequence, which serves as the input for the model. The dataset is then split into training and testing sets with an 80-20 split.

```
# Load data
data_distilbert = pd.read_csv("training_dataset.csv")

# Tokenize and pad sequences using DistilBERT tokenizer
tokenizer_distilbert = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
maxlen_distilbert = 300

X_title_distilbert = data_distilbert['Title_tokens'].apply(lambda x: eval(x)).values
X_body_distilbert = data_distilbert['Body_tokens'].apply(lambda x: eval(x)).values
X_combined_distilbert = [' '.join(title + body) for title, body in zip(X_title_distilbert, X_body_distilbert)]

X_train_distilbert, X_test_distilbert, y_train_distilbert, y_test_distilbert = train_test_split(X_combined_distilbert, data_distilbert['LabelNum']

# Tokenize and encode sequences
train_encodings_distilbert = tokenizer_distilbert(X_train_distilbert, truncation=True, padding=True, max_length=maxlen_distilbert)
test_encodings_distilbert = tokenizer_distilbert(X_test_distilbert, truncation=True, padding=True, max_length=maxlen_distilbert)
```

Figure 45: DistilBERT Model

6.2 Model Architecture and Training

We load the pre-trained DistilBERT model for sequence classification. This model includes the DistilBERT architecture with a classification head on top. We compile the model with the Adam optimizer and sparse categorical cross-entropy loss function. To prevent overfitting, early stopping is applied during training. The model is trained for a specified number of epochs on the training dataset, and its performance is evaluated on the testing dataset.

```
# Load pre-trained DistilBERT model for sequence classification
model_distilbert = TFDistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased', num_labels=2)

# Compile the model
optimizer_distilbert = tf.keras.optimizers.Adam(learning_rate=5e-5)
loss_distilbert = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model_distilbert.compile(optimizer=optimizer_distilbert, loss=loss_distilbert, metrics=['accuracy'])

# Add early stopping to mitigate overfitting
early_stopping_distilbert = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', patience=3, restore_best_weights=True
)

# Fit the model
history_distilbert = model_distilbert.fit(
    train_dataset_distilbert,
    epochs=3,
    validation_data=test_dataset_distilbert,
    callbacks=[early_stopping_distilbert]
)

# Evaluate the model
loss_distilbert, accuracy_distilbert = model_distilbert.evaluate(test_dataset_distilbert)
print("Test Accuracy (DistilBERT):", accuracy_distilbert)
```

Figure 46: DistilBERT Architecture

Due to the computational expenses of the transformer, it was only trained for 3 epochs, but due to its complexity and pre-trained nature, this was enough to reach satisfactory accuracy rates.

6.3 Model Evaluation and Prediction

We evaluate the trained model on the testing dataset to assess its accuracy in predicting binary labels (IOS or ANDROID). Which yielded a training accuracy of 98.56%, loss of 0.0519 and test accuracy of 98.55%.

```
Some weights of the PyTorch model were not used when initializing the TF 2.0 model TFDistilBertForSequenceClassification: ['vocab_layer_norm.b
- This IS expected if you are initializing TFDistilBertForSequenceClassification from a PyTorch model trained on another task or with another
- This IS NOT expected if you are initializing TFDistilBertForSequenceClassification from a PyTorch model that you expect to be exactly identi
Some weights or buffers of the TF 2.0 model TFDistilBertForSequenceClassification were not initialized from the PyTorch model and are newly in
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Epoch 1/3
2340/2340 [==============================] - 14438s 6s/step - loss: 0.0658 - accuracy: 0.9788 - val_loss: 0.0600 - val_accuracy: 0.9804
Epoch 2/3
2340/2340 [==============================] - 14581s 6s/step - loss: 0.0370 - accuracy: 0.9877 - val_loss: 0.0468 - val_accuracy: 0.9845
Epoch 3/3
2340/2340 [==============================] - 14343s 6s/step - loss: 0.0244 - accuracy: 0.9924 - val_loss: 0.0519 - val_accuracy: 0.9856
585/585 [==============================] - 1002s 2s/step - loss: 0.0519 - accuracy: 0.9856
Test Accuracy (DistilBERT): 0.9855753779411316
```

Figure 47: DistilBERT Results

Additionally, we provide an option for users to input custom text, which is pre-processed and passed to the model for prediction. The model predicts the class label (ANDROID or IOS) for the input text based on its learned features.

```
1/1 [==============================] - 0s 60ms/step
User Input:  app store
-3.645697
3.4873729
Predicted class: IOS
```

Figure 48: DistilBERT IOS Test

Figure 49: DistilBERT ANDROID Test

6.4   Results Analysis

Finally, we analyze the model's performance using metrics such as accuracy, confusion matrix, and classification report. These metrics provide insights into the model's classification performance, including true positive, true negative, false positive, and false negative predictions.
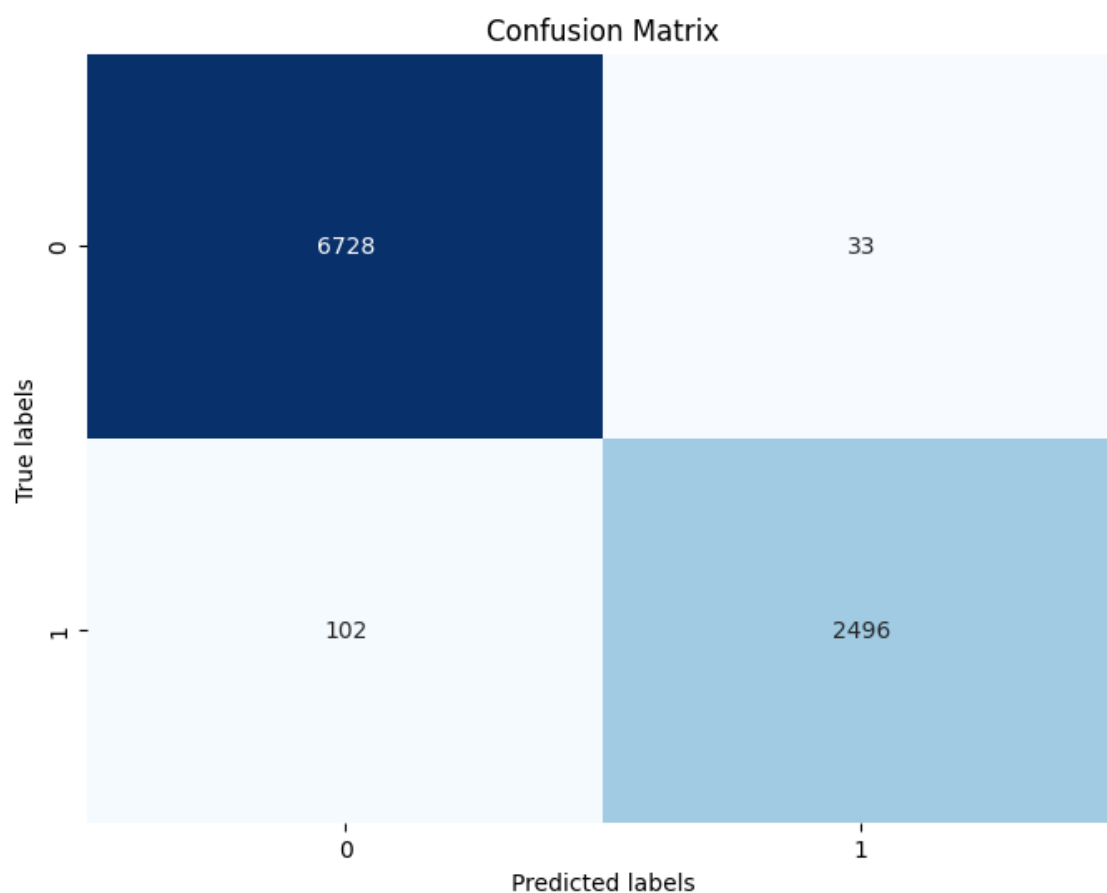
Figure 50: DistilBERT Confusion Matrix

Table 3: Classification Report

Class	Precision	Recall	F1-Score	Support
0	0.99	1.00	0.99	6761
1	0.99	0.96	0.97	2598
Accuracy	0.99			
Macro Avg	0.99			
Weighted Avg	0.99			

24

6.5 RoBERTa Model

RoBERTa (Robustly optimized BERT approach) represents an advanced transformer architecture renowned for its exceptional performance in various natural language processing tasks. Built upon the BERT architecture with refined training techniques, RoBERTa offers superior capabilities in understanding and processing textual data.

```python
# Load data
data_roberta = pd.read_csv("training_dataset.csv")

# Tokenize and pad sequences using RoBERTa tokenizer
tokenizer_roberta = RobertaTokenizer.from_pretrained('roberta-base')
maxlen_roberta = 300

X_title_roberta = data_roberta['Title_tokens'].apply(lambda x: eval(x)).values
X_body_roberta = data_roberta['Body_tokens'].apply(lambda x: eval(x)).values
X_combined_roberta = [' '.join(title + body) for title, body in zip(X_title_roberta, X_body_roberta)]

X_train_roberta, X_test_roberta, y_train_roberta, y_test_roberta = train_test_split(X_combined_roberta, data_roberta['LabelNum'].values, test_size

# Tokenize and encode sequences
train_encodings_roberta = tokenizer_roberta(X_train_roberta, truncation=True, padding=True, max_length=maxlen_roberta)
test_encodings_roberta = tokenizer_roberta(X_test_roberta, truncation=True, padding=True, max_length=maxlen_roberta)
```

Figure 51: RoBERT Model

6.6 Model Architecture and Training

We begin by loading the dataset and tokenizing it using the RoBERTa tokenizer. The dataset comprises textual information of questions categorized into titles and bodies, with numerical labels indicating the class (1 for IOS and 0 for ANDROID). By concatenating the title and body for each entry, we form input sequences for the RoBERTa model. The dataset is then split into training and testing sets with an 80-20 split.

```python
# Load pre-trained RoBERTa model for sequence classification
model_roberta = TFRobertaForSequenceClassification.from_pretrained('roberta-base', num_labels=2)

# Compile the model
optimizer_roberta = tf.keras.optimizers.Adam(learning_rate=5e-5)
loss_roberta = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model_roberta.compile(optimizer=optimizer_roberta, loss=loss_roberta, metrics=['accuracy'])

# Add early stopping to mitigate overfitting
early_stopping_roberta = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', patience=3, restore_best_weights=True
)

# Fit the model
history_roberta = model_roberta.fit(
    train_dataset_roberta,
    epochs=1,
    validation_data=test_dataset_roberta,
    callbacks=[early_stopping_roberta]
)

# Evaluate the model
loss_roberta, accuracy_roberta = model_roberta.evaluate(test_dataset_roberta)
print("Test Accuracy (RoBERTa):", accuracy_roberta)
```

Figure 52: RoBERT Architecture

We loaded the pre-trained RoBERTa model for sequence classification. This model architecture integrates RoBERTa's advanced features with a classification head. To facilitate model training, we compile it using the Adam optimizer and sparse categorical cross-entropy loss function. Furthermore, early stopping is employed to prevent overfitting during training. The model undergoes training for a specified number of epochs on the training dataset, followed by evaluation on the testing dataset.

6.7 Model Evaluation and Prediction

Evaluation metrics such as training accuracy, loss, and test accuracy are obtained to assess the model's performance. Despite the computational resources required by RoBERTa, it achieves satisfactory accuracy rates (Train Accuracy = 98.37% , loss = 0.0564 and Test Accuracy = 98.36%) within the specified constraints (Epochs = 1).

Some weights of the PyTorch model were not used when initializing the TF 2.0 model TFRobertaForSequenceClassification: ['roberta.embeddings.posit
- This IS expected if you are initializing TFRobertaForSequenceClassification from a PyTorch model trained on another task or with another archite
- This IS NOT expected if you are initializing TFRobertaForSequenceClassification from a PyTorch model that you expect to be exactly identical (e
Some weights or buffers of the TF 2.0 model TFRobertaForSequenceClassification were not initialized from the PyTorch model and are newly initiali
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
2340/2340 [==============================] - 29951s 13s/step - loss: 0.0867 - accuracy: 0.9709 - val_loss: 0.0564 - val_accuracy: 0.9837
585/585 [==============================] - 2127s 4s/step - loss: 0.0564 - accuracy: 0.9837
Test Accuracy (RoBERTa): 0.9836521148681641

Figure 53: RoBERT Results

Additionally, the model provides the flexibility for users to input custom text for prediction. Based on the learned features, the model predicts the class label (ANDROID or IOS) for the input text.

```
1/1 [==============================] - 0s 80ms/step
User Input:  app store
0.013657359
0.074780345
Predicted class: IOS
```

Figure 54: RoBERT IOS Test

```
1/1 [==============================] - 0s 64ms/step
User Input:  play store
1.0664793
-0.9198637
Predicted class: Android
```

Figure 55: RoBERT ANDROID Test

6.8 Results Analysis

Finally, a comprehensive analysis of the RoBERTa model's performance is conducted using various metrics, including accuracy, confusion matrix, and classification report. These metrics offer valuable insights into the model's classification capabilities and its effectiveness in distinguishing between IOS and ANDROID topics.

The difference between the training results and the confusion matrix mainly comes from the small training period for the RoBERT which results in lower accuracy and precision rates than DistilBERT. This can easily be mitigated by training it for more epochs.
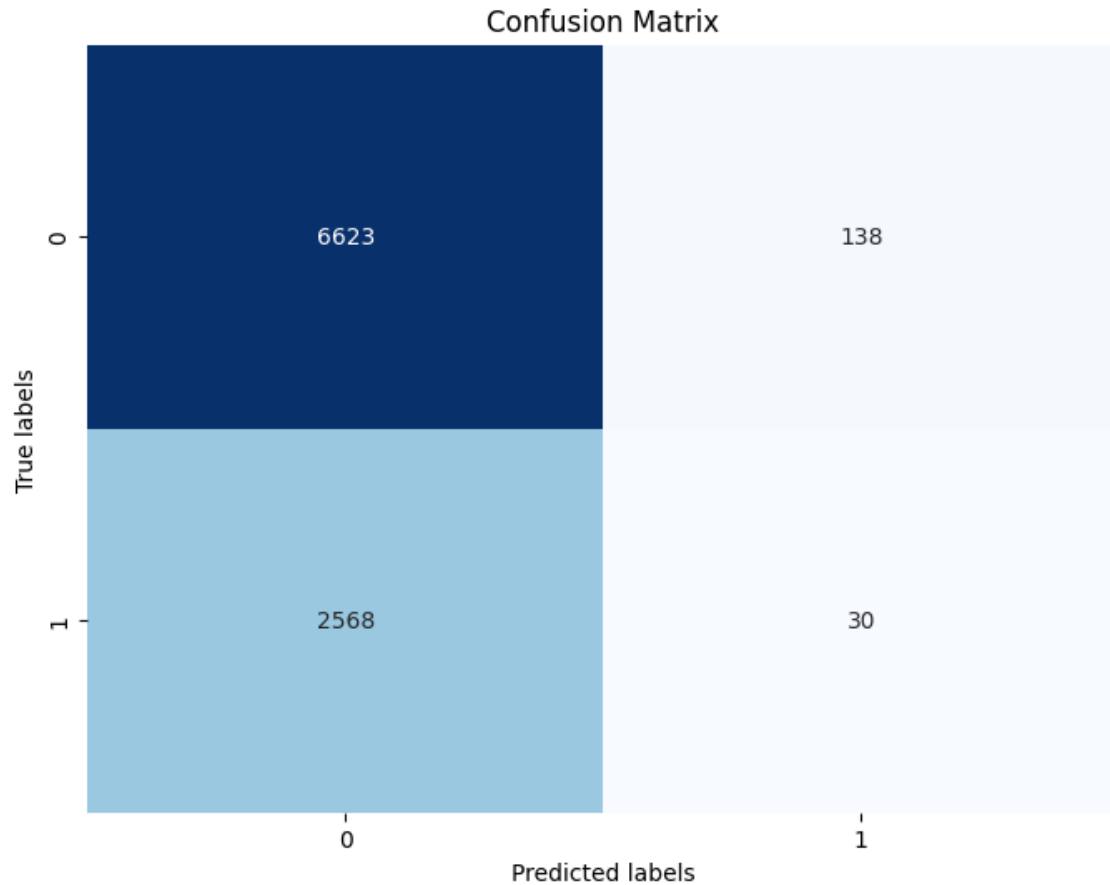
Figure 56: RoBERT Confusion Matrix

Table 4: Classification Report

Class	Precision	Recall	F1-Score	Support
0	0.72	0.98	0.83	6761
1	0.18	0.01	0.02	2598
Accuracy		0.71		
Macro Avg		0.45	0.50	0.43
Weighted Avg		0.57	0.71	0.61

6.9 Conclusion

In conclusion, we demonstrate the process of text classification using pre-trained transformer models (Distil-BERT and RoBERTa). These models where chosen due to their satisfactory trade-off between computational complexity and performance.

Both models showed very good results, even though RoBERT was trained for only 1 epoch, it reached a very similar accuracy level as DistilBERT which was trained for 3 epochs (98.37% VS. 98.56%).

This shows that both are effecient for text classification and their usage depends on the complexity of the task and the available resources to be used on their training.

Neither RoBERT model nor the DistilBERT model needed any editing in the architecture of the Neural Network as they both are suitable for classification tasks.

Finally, this concludes our classification task journey that ends with powerful results and a notebook full of useful code that starts with raw data and transforms it to meaningful output that can be further developed to multiple NLP related tasks.

7 Regression Task Using Pre-trained Model

This section provides an in-depth analysis of implementing a regression task using the DistilBERT model. It discusses the selection process of various models, explains how DistilBERT works, and details the necessary modifications for regression tasks. Additionally, it covers the theoretical underpinnings of self-attention, transformer architecture, fine-tuning, and the data preprocessing steps involved.

7.1 Model Selection: Comparison and Choice

In our project, we initially considered several models: ALBERT, RoBERTa, XLNet, and DistilBERT. Each model has unique advantages and is tailored for specific use cases in natural language processing (NLP).

- **ALBERT**: Known for its parameter-sharing mechanism and efficient size, ALBERT is designed to reduce memory consumption and improve scalability. However, it often requires more training time and computational resources to achieve optimal performance.

- **RoBERTa**: An optimized version of BERT, RoBERTa excels in performance due to its extensive training on more data and modifications in training strategies. Despite its superior performance, it is computationally intensive.

- **XLNet**: This model outperforms many in capturing bidirectional context and achieving state-of-the-art results on various benchmarks. Its complexity and higher computational requirements, however, pose significant challenges.

- **DistilBERT**: A distilled version of BERT, DistilBERT is designed to be smaller, faster, and lighter while retaining 97% of BERT's performance. This balance of efficiency and effectiveness makes it an ideal choice for our regression task, where computational resources and speed are critical factors.

Given these considerations, we chose DistilBERT for its efficient architecture and strong performance in understanding language tasks. The following image shows the results using each of the mentioned models
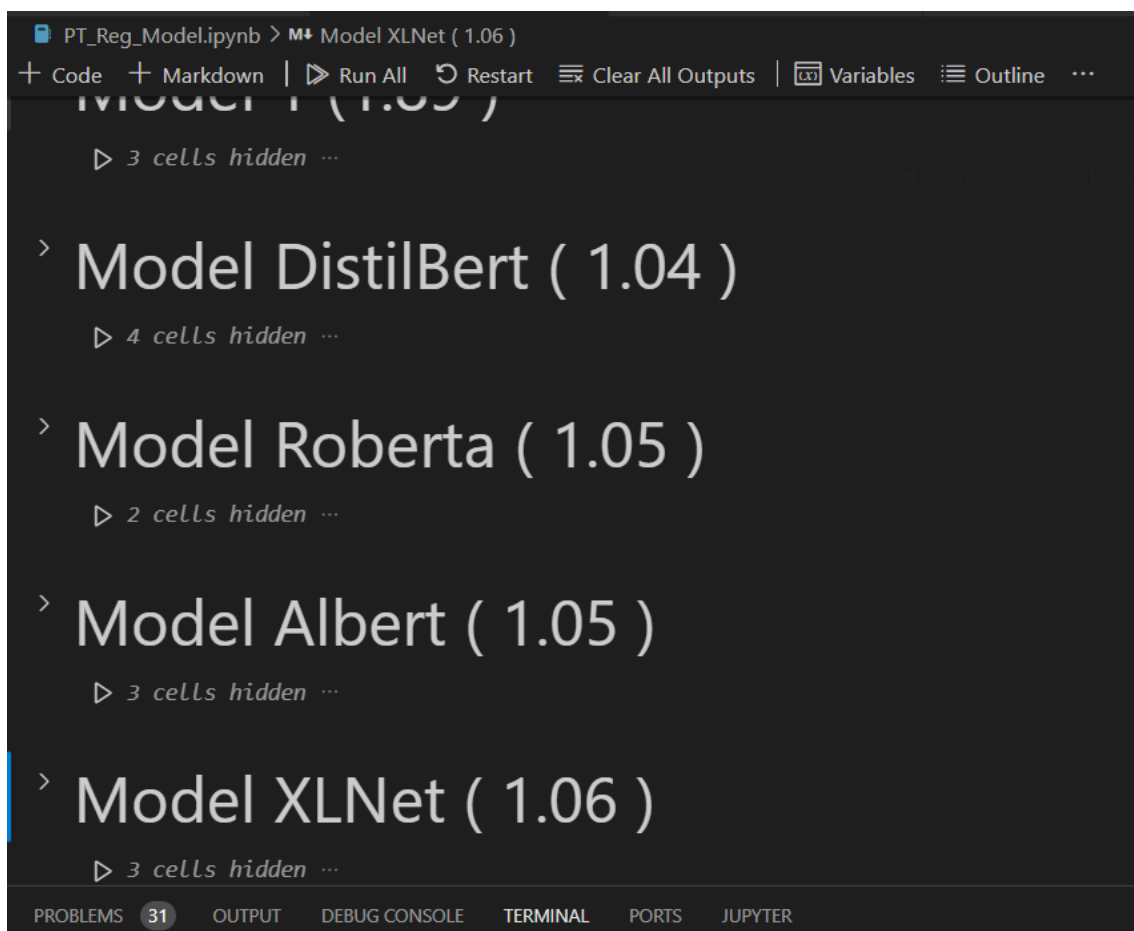

Figure 57: Pretrained Regression Models

7.2 Understanding DistilBERT and Transformer Architecture

7.2.1 DistilBERT Architecture

DistilBERT is a compact version of BERT, achieved through a technique called knowledge distillation. It reduces the number of layers from 12 (in BERT base) to 6, which significantly decreases model size and inference time. Despite the reduction, it maintains high language understanding by learning to mimic the output distributions of the larger BERT model during training.

7.2.2 Transformer Architecture

The underlying architecture of DistilBERT is the transformer, which uses self-attention mechanisms to process sequences of text. Here's a breakdown of how transformers work:

- **Input Embeddings**: Words in the input sequence are converted into dense vectors (embeddings).

- **Self-Attention**: For each word, the model computes Query (Q), Key (K), and Value (V) vectors. The self-attention mechanism then calculates attention scores by taking the dot product of Q and K, followed by a softmax operation to obtain attention weights. These weights are used to compute a weighted sum of the V vectors, capturing the importance of each word in the context of others.

- **Multi-Head Attention**: Instead of relying on a single set of attention scores, transformers use multiple attention heads to capture various aspects of word relationships. The outputs of these heads are concatenated and linearly transformed.

- **Positional Encoding**: Transformers do not have inherent order information, so positional encodings are added to input embeddings to retain word order information.

- **Feed-Forward Network**: Each word representation is passed through a feed-forward neural network after the self-attention layer.

- **Layer Normalization and Residual Connections**: These techniques help stabilize and accelerate training by normalizing outputs and adding the original input to the output of each sub-layer.

7.3 Self-Attention Mechanism

Self-attention is a core feature of transformers, allowing the model to weigh the relevance of different words in a sequence. In our implementation, the self-attention mechanism calculates attention scores for each word pair within the input, providing a dynamic context-aware representation of each word. This mechanism replaces the need for attention masks, as it inherently captures the relationships and importance of each word relative to others.

7.4 Data Preprocessing: Tokenization and Dataset Creation

To prepare our data for the model, we use the DistilBERT tokenizer. This tokenizer converts text into token IDs, ensuring uniform length through padding and truncation.

Listing 1: Tokenization and Dataset Creation

```
from transformers import DistilBertTokenizer
tokenizer3 = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
maxlen3 = 100

X_title3 = data3['Title_tokens'].apply(lambda x: eval(x)).values
X_body3 = data3['Body_tokens'].apply(lambda x: eval(x)).values
X_combined3 = [' '.join(title + body) for title, body in zip(X_title3, X_body3)]
```

```
# Tokenize and pad sequences using BERT tokenizer
tokenizer3 = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
maxlen3 = 100


X_title3 = data3['Title_tokens'].apply(lambda x: eval(x)).values
X_body3 = data3['Body_tokens'].apply(lambda x: eval(x)).values
X_combined3 = [' '.join(title + body) for title, body in zip(X_title3, X_body3)]
```

Figure 58: Tokenization and Dataset Creation

After tokenization, we split the data into training and testing sets:

Listing 2: Splitting Data

```
from sklearn.model_selection import train_test_split
X_train3, X_test3, y_train3, y_test3 = train_test_split(X_combined3, data3['Score'].values
```

```
X_train3, X_test3, y_train3, y_test3 = train_test_split(X_combined3, data3['Score'].values, test_size=0.2, random_state=42)

train_encodings3 = tokenizer3(X_train3, truncation=True, padding=True, max_length=maxlen3)
test_encodings3 = tokenizer3(X_test3, truncation=True, padding=True, max_length=maxlen3)
```

Figure 59: Splitting Data

The tokenized sequences are then converted into TensorFlow datasets. These datasets are shuffled and batched to optimize the training process, enhancing model performance and generalization.

Listing 3: TensorFlow Datasets

```
import tensorflow as tf
import numpy as np

train_encodings3 = tokenizer3(X_train3, truncation=True, padding=True, max_length=maxlen3)
test_encodings3 = tokenizer3(X_test3, truncation=True, padding=True, max_length=maxlen3)

train_dataset3 = tf.data.Dataset.from_tensor_slices((
    {key: np.array(value) for key, value in train_encodings3.items()},
    y_train3
)).shuffle(len(X_train3)).batch(16)

test_dataset3 = tf.data.Dataset.from_tensor_slices((
    {key: np.array(value) for key, value in test_encodings3.items()},
    y_test3
)).batch(16)
```

```
# Convert lists to TensorFlow Dataset
train_dataset3 = tf.data.Dataset.from_tensor_slices((
    {key: np.array(value) for key, value in train_encodings3.items()},
    y_train3
)).shuffle(len(X_train3)).batch(16)

test_dataset3 = tf.data.Dataset.from_tensor_slices((
    {key: np.array(value) for key, value in test_encodings3.items()},
    y_test3
)).batch(16)
```

Figure 60: TensorFlow Datasets

7.5 Model Fine-Tuning and Training

We load the pre-trained DistilBERT model, tailored for sequence classification, and modify it for our regression task. The final layer's activation function is changed to linear, allowing the model to output continuous values necessary for predicting scores.

Listing 4: Model Fine-Tuning

```
from transformers import TFDistilBertForSequenceClassification
model3 = TFDistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased',
model3.layers[-1].activation = tf.keras.activations.linear
```

```python
# Load pre-trained DistilBERT model for sequence classification
model3 = TFDistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased', num_labels=1)

# Modify the model for regression
model3.layers[-1].activation = tf.keras.activations.linear
```

Figure 61: Model Fine-Tuning

7.5.1 Importance of Linear Activation

For regression tasks, it is crucial to use a linear activation function in the final layer. Unlike non-linear activations such as softmax or sigmoid, which constrain the output to a specific range, linear activation allows the model to output any real value. This flexibility is essential for predicting continuous scores accurately.

We compile the model using the Adam optimizer, known for its efficiency and adaptive learning rate, which helps handle sparse gradients and improves convergence. The Mean Absolute Error (MAE) is chosen as the loss function and evaluation metric, suitable for regression tasks as it measures the average magnitude of errors in predictions.

Listing 5: Model Compilation

```
optimizer3 = 'adam'
loss3 = tf.keras.losses.MeanAbsoluteError()
model3.compile(optimizer=optimizer3, loss=loss3, metrics=['mean_absolute_error'])
```

```python
# Compile the model with Mean Absolute Error (MAE) as the loss function and evaluation metric
optimizer3 = 'adam'  # Use string identifier
loss3 = tf.keras.losses.MeanAbsoluteError()  # Change to MAE
model3.compile(optimizer=optimizer3, loss=loss3, metrics=['mean_absolute_error'])  # Using MAE as a metric for monitoring
```

Figure 62: Model Compilation

The model is then fine-tuned on our training data for 3 epochs. Fine-tuning involves training a pre-trained model on a specific dataset to adapt it to the task at hand. We chose 3 epochs to balance training time and performance, minimizing the risk of overfitting.

Listing 6: Fine-Tuning the Model

```
model3.fit(train_dataset3, epochs=3, validation_data=test_dataset3)
```

```python
# Fine-tune the model
model3.fit(train_dataset3, epochs=3, validation_data=test_dataset3)

# Evaluate the model
```

Figure 63: Fine-Tuning the Model

7.6 Model Evaluation

After training, we evaluate the model on the test dataset to measure its performance. The evaluation metric used is Mean Absolute Error, which provides an intuitive measure of prediction accuracy by averaging the absolute differences between predicted and actual values.

Listing 7: Model Evaluation

```
loss3, mae3 = model3.evaluate(test_dataset3)
print("Mean-Absolute-Error-3:", mae3)
```
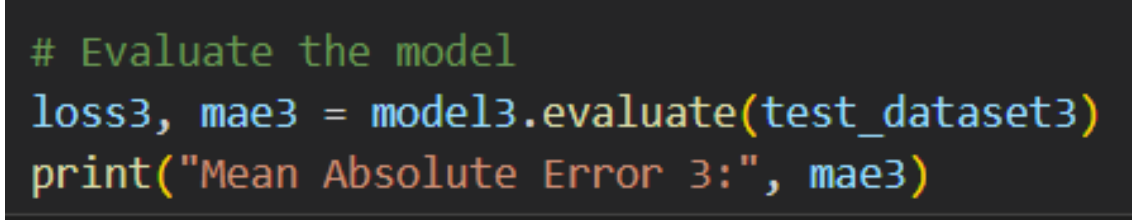

Figure 64: Model Evaluation

7.7 Model Training Results and Analysis

After implementing the regression task using DistilBERT, we observed several key results and warnings during the training process. These observations are crucial for understanding the model's performance and ensuring its proper usage.

7.7.1 Training Results

The training process spanned three epochs, and the following results were recorded:

- **Epoch 1**:

  ```
  loss: 1.0526, mean_absolute_error: 1.0526,
  val_loss: 1.0527, val_mean_absolute_error: 1.0527
  ```

- **Epoch 2**:

  ```
  loss: 1.0438, mean_absolute_error: 1.0438,
  val_loss: 1.0644, val_mean_absolute_error: 1.0644
  ```

- **Epoch 3**:

  ```
  loss: 1.0369, mean_absolute_error: 1.0369,
  val_loss: 1.0493, val_mean_absolute_error: 1.0493
  ```

The final evaluation on the test dataset resulted in a Mean Absolute Error (MAE) of 1.0493.
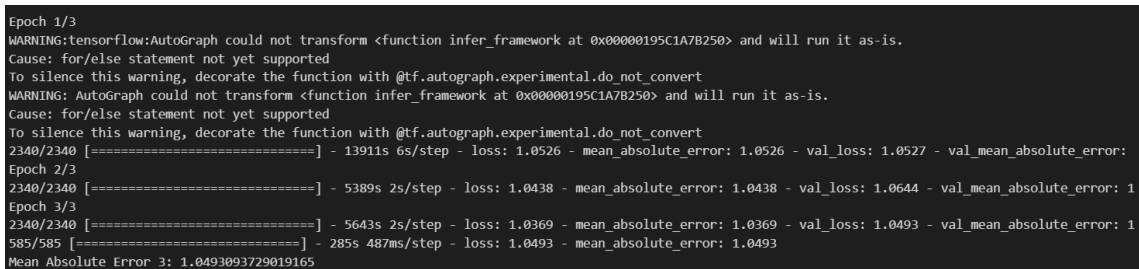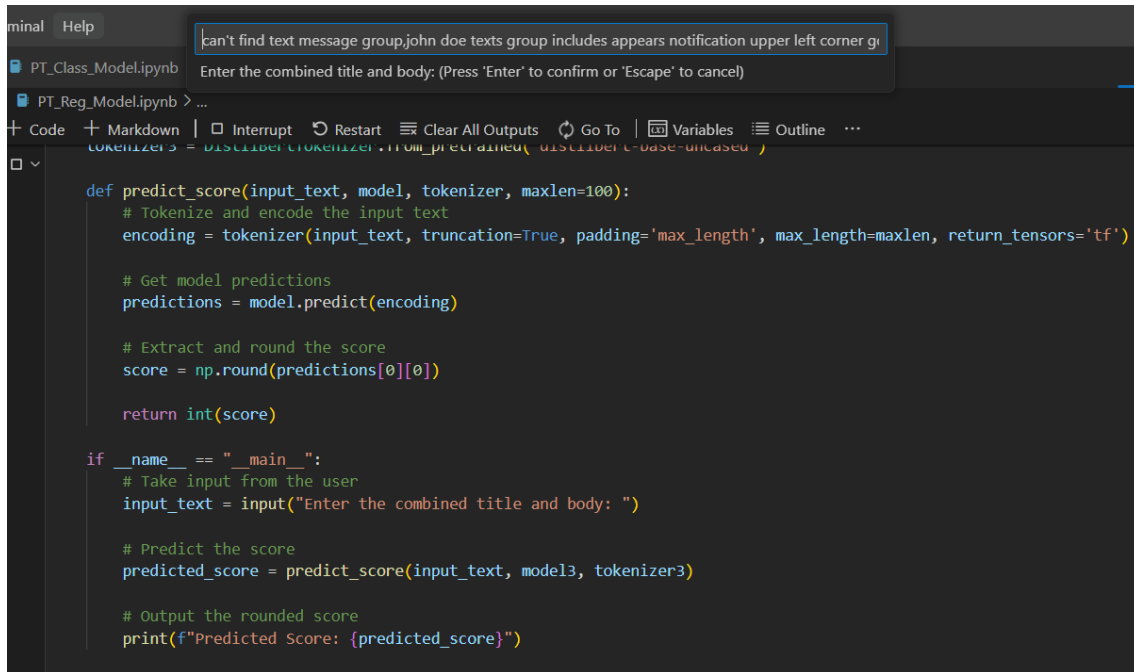

Figure 65: Results

7.7.2 Analysis of Results

The training results show a gradual decrease in both training and validation loss and MAE, indicating that the model is learning effectively. However, the slight increase in validation loss and MAE during the second epoch suggests that the model might have started overfitting slightly. This is mitigated by the subsequent decrease in the third epoch.

The following code was used to test the model against an input from the dataset that has score 1. After testing using our model, the predicted score is also 1. Input : *can't find text message group,john doe texts group includes appears notification upper left corner go text message conversations john doe there.*
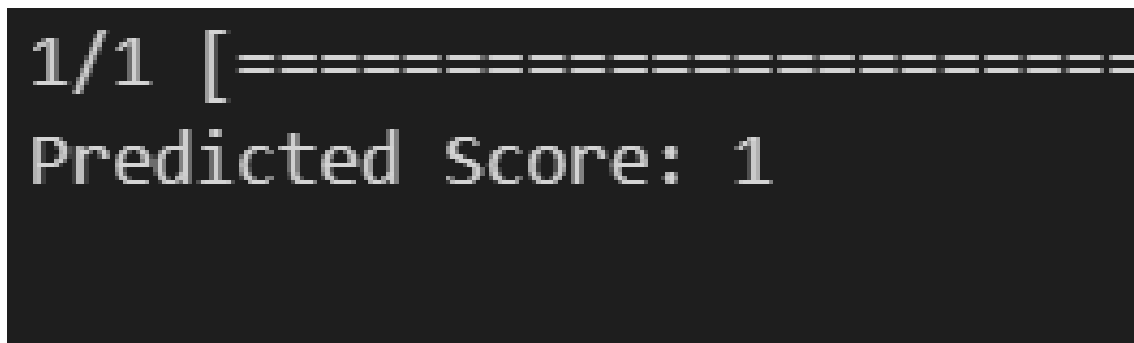

Figure 66: Results

Figure 67: Results

7.7.3 Holistic Overview of Model Performance

The model performance can be assessed based on the Mean Absolute Error (MAE), which measures the average magnitude of errors in predictions, without considering their direction (positive or negative). A lower MAE indicates better predictive accuracy.

- **Model Accuracy**: The final MAE of 1.0493 suggests that, on average, the model's predictions deviate from the actual scores by approximately 1.05 points. For many regression tasks, this level of accuracy is considered reasonable, especially when dealing with complex and nuanced text data.

- **Learning Trends**: The progressive reduction in both training and validation loss and MAE across the epochs indicates that the model effectively learned from the training data. The initial increase in validation MAE during the second epoch, followed by a decrease, suggests a slight overfitting tendency that was corrected in subsequent training.

- **Generalization Capability**: The relatively close values of training MAE and validation MAE indicate good generalization, meaning the model performs consistently well on unseen data. This balance is crucial for ensuring that the model's predictions are reliable in real-world applications.

7.8   Comparison of Custom Model and Pretrained Model for Regression Task

In this subsection, we provide a detailed comparison between the custom-configured model and the pretrained model used for the regression task. This comparison is essential to understand the differences in performance and the underlying reasons for these differences. The aim is to highlight the benefits and limitations of both approaches and explain the rationale behind their usage in the context of our project milestones.

7.8.1   Custom Model Configuration

In milestone 2, we configured and trained a custom LSTM-based model from scratch. Here is a summary of the key steps and components of this model:

1. **Data Preparation**:
   - The dataset was loaded and tokenized using `Tokenizer` from `tensorflow.keras.preprocessing.text`.
   - Token sequences were padded to ensure uniform input length.

2. **Word Embedding**:
   - We trained a `Word2Vec` model on the tokenized text data to generate word embeddings.
   - An embedding matrix was created to map each word in our vocabulary to its corresponding vector from the `Word2Vec` model.

3. **Model Architecture**:
   - An `Embedding` layer was initialized with the pretrained `Word2Vec` embeddings, set to be non-trainable.
   - This was followed by an LSTM layer with 64 units, incorporating dropout for regularization.
   - The final layer was a `Dense` layer with a linear activation function to output a single value for regression.

4. **Compilation and Training**:
   - The model was compiled using the Adam optimizer.
   - Training was conducted once over 10 epochs with with MSA loss function and once over 5 epochs with MAE function with a batch size of 32.

5. **Evaluation**:
   - The model was evaluated on a test set, resulting in a mean absolute error (MAE) of 1.049.

7.8.2   Pretrained Model Configuration

In a subsequent phase, we utilized a pretrained DistilBERT model fine-tuned for the regression task. Here is a summary of the key steps and components of this approach:

1. **Data Preparation**:
   - The dataset was tokenized using the `DistilBERT` tokenizer.
   - Token sequences were padded/truncated to a maximum length of 100.

2. **Pretrained Model**:
   - We used the `TFDistilBertForSequenceClassification` model from the `transformers` library, modified for regression by changing the output layer to have a linear activation function.

3. **Model Training**:
   - The model was compiled using the Adam optimizer and mean absolute error (MAE) as the loss function.
   - Fine-tuning was conducted over 3 epochs with a batch size of 16.

4. **Evaluation**:
   - The pretrained model was evaluated on the same test set, resulting in a mean absolute error (MAE) of 1.04.

7.8.3 Comparison and Analysis

Differences in Model Configuration:

- **Embedding Initialization**:
  - The custom model used `Word2Vec` embeddings trained on the specific dataset, while the pretrained model used embeddings learned during DistilBERT's pretraining on a large corpus.

- **Model Architecture**:
  - The custom model had an LSTM layer to handle sequential data, whereas the pretrained model utilized transformer layers inherent to DistilBERT, which can capture long-range dependencies more effectively.

Differences in Training and Fine-Tuning:

- The custom model required training from scratch, which involved initializing and learning embeddings and model weights specific to the dataset.

- The pretrained model leveraged transfer learning, where the DistilBERT model already had rich, contextual embeddings and required only fine-tuning on the specific regression task.

Performance Comparison:

- The pretrained model achieved a slightly better MAE (1.04) compared to the custom model (1.05). This minor difference is expected given the advanced capabilities of transformer models like DistilBERT in capturing complex language patterns and dependencies.

- The marginal improvement demonstrates the efficacy of pretrained models, which benefit from extensive pretraining on diverse datasets, leading to better generalization and performance on specific tasks with minimal fine-tuning.

Figure 68: Non Pre-Trained Model Mean Absolute Error

Figure 69: Pre-Trained Model Mean Absolute Error

Rationale Behind Results:

- The pretrained model's superior performance can be attributed to its extensive pretraining on vast amounts of text data, providing it with a strong language understanding capability.

- The custom model, while performing slightly worse, still achieved competitive results, highlighting the effectiveness of traditional methods like LSTM with well-trained word embeddings.

Overview comaprison between both models:

- The inclusion of both models provides a comprehensive understanding of different approaches to tackling the regression task. The pretrained model serves as a benchmark, demonstrating the state-of-the-art performance achievable with transfer learning. In contrast, the custom model offers insight into traditional methods and their efficacy when tailored to specific datasets.

- This comparison is crucial for making informed decisions about model selection, balancing the trade-offs between computational resources, model complexity, and performance.

7.9 Conclusion

This report outlines the comprehensive process of implementing a regression task using DistilBERT, from data loading to model evaluation. We discussed the selection process of different models, the architecture and functionality of DistilBERT, and the essential modifications for regression tasks. The implementation leverages the power of transformers and self-attention mechanisms, resulting in an effective solution for predicting scores based on text input. By using a linear activation function in the final layer, we ensure that the model outputs continuous values suitable for regression, and through fine-tuning, we adapt the pre-trained model to our specific task, achieving a balance between performance and computational efficiency.

References

1. Arijit Das and Diganta Saha, "Question Answering System Using Deep Learning in the Low Resource Language Bengali," (2022), `https://doi.org/10.1002/9781119857686.ch10`.

2. Tzu-Hsuan Lin, Yu-Hua Huang, and Alan Putranto, "Intelligent question and answer system for building information modeling and artificial intelligence of things based on the bidirectional encoder representations from transformers model," *Automation in Construction*, Volume 142, (2022), 104483, ISSN 0926-5805, `https://doi.org/10.1016/j.autcon.2022.104483`.

3. Tatenda Beta and Tirivangani Magadza, "Question-And-Answer System Using Natural Language Processing," (2023).

4. S. LAVANYA, "End to End Question-Answering System Using NLP and SQuAD Dataset," *Analytics Vidhya*, `https://www.analyticsvidhya.com/blog/2021/11/end-to-end-question-answering-system-using-nlp-and-sq` Accessed 9 March 2024.

5. Sucholutsky, Ilia & Narayan, Apurva & Schonlau, Matthias & Fischmeister, Sebastian. (2019). Pay attention and you won't lose it: a deep learning approach to sequence imputation. PeerJ Computer Science. 5. e210. 10.7717/peerj-cs.210.

6. Bahdanau, Dzmitry & Cho, Kyunghyun & Bengio, Y.. (2014). Neural Machine Translation by Jointly Learning to Align and Translate. ArXiv. 1409.

7. Devlin, Jacob & Chang, Ming-Wei & Lee, Kenton & Toutanova, Kristina. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.

8. Fields, Clayton & Kennington, Casey. (2023). Vision Language Transformers: A Survey.

9. Liu, Yinhan & Ott, Myle & Goyal, Naman & Du, Jingfei & Joshi, Mandar & Chen, Danqi & Levy, Omer & Lewis, Mike & Zettlemoyer, Luke & Stoyanov, Veselin. (2019). RoBERTa: A Robustly Optimized BERT Pretraining Approach.

10. Zulqarnain, & Zager, Ahmed & Alsaedi, Zager & Ghazali, Rozaida & Ghouse, Muhammad & Sharif, Wareesa & Husaini, Noor. (2021). A comparative analysis on question classification task based on deep learning approaches. PeerJ Computer Science. 7. e570. 10.7717/peerj-cs.570.