Report of mini project 2

Nazmus Sakib
nazmus.x.sakib@abo.fi

Problem statement:

The project aims to perform sentiment analysis using machine learning techniques. Sentiment analysis involves determining the sentiment expressed in a text, such as whether it is positive or negative. Our focus lies in classifying tweets whether it is positive or negative from the Sentiment140 dataset, which consists of approximately 1,60,000 annotated tweets. These tweets have been labeled with sentiment. The tasks in this project includes preparing the tweets through data preprocessing techniques to clean and transform it for the model, selecting appropriate machine learning models for sentiment analysis, and comparing their effectiveness. Data preprocessing involves steps such as removing unnecessary characters, tokenizing, padding and embedding. Model selection involves exploring various algorithms, from traditional methods like Logistic Regression to advanced deep learning architectures like CNNs and RNNs. Finally, performance visualization entails calculating metrics such as accuracy and employing visualizations like confusion matrices to compare the models' performance.

Here is the google colab link
https://colab.research.google.com/drive/1QH7f9ewMcRgWd6IHf3fy0TdwgIDtrDZ1?usp=sharing

Data processing:

- **Contraction Expansion**: A dictionary of contractions was used which contained commonly used contractions and their expanded forms. Each contraction in the tweet text was replaced with its expanded form using the contractions dictionary.
- **Removal of URLs**: URLs were replaced with an empty string (''). Regular expressions were used to identify and remove URLs from the tweet text. A regex pattern was defined to match URLs, including those starting with 'http://', 'https://', or 'www.'.
- **Removal of User Mentions**: User mentions were replaced with an empty string (''). Regular expressions were employed to identify and remove user mentions from the tweet text. A regex pattern was defined to match user mentions, which typically start with '@' followed by a username.
- **Removal of Special Characters and Symbols**: Non-alphanumeric characters and symbols were removed from the tweet text to ensure uniformity and simplify subsequent analysis. A regex pattern was defined to match non-alphanumeric characters and symbols. Special characters and symbols were replaced with a space (' ') to separate words.
- **Emoji Removal**: Regular expressions were utilized to identify and remove emojis from the tweet text. Separate regex patterns were defined to match smiley emojis, sad emojis, neutral emojis, and lol emojis. Emojis were replaced with an empty string (''').

- **Repeated Character Reduction:** A regex pattern was defined to identify sequences of three or more consecutive letters in the tweet text. Repeated characters were reduced to two occurrences to eliminate redundancy and normalize the text.

Here is how the preprocessing has updated the text.

```
for i in range(2,10):
  print("Text:", df.iloc[i,1])
  print("Processed:", df.iloc[i,2])
```

```
Text: @greeniebach I reckon he'll play, even if he's not 100%...but i know nothing!! ;) It won't be the same without him.
Processed:  i reckon he will play  even if he is not 100   but i know nothing    it will not be the same without him
Text: @vaLewee I know!  Saw it on the news!
Processed:  i know   saw it on the news
Text: very sad that http://www.fabchannel.com/ has closed down. One of the few web services that I've used for over 5 years
Processed: very sad that  has closed down  one of the few web services that i have used for over 5 years
Text: @Fearnecotton who sings 'I Remember'? i alwaysss hear it on Radio 1 but never catch the artist
Processed:  who sings  i remember   i alwayss hear it on radio 1 but never catch the artist
Text: With God on ur side anything is possible....
Processed: with god on ur side anything is possible
Text: @LoveSmrs why being stupid?
Processed:  why being stupid
Text: Having delved back into the guts of Expression Engine, its a flexible CMS if you have to use it as a dev, not great for clients though
Processed: having delved back into the guts of expression engine  its a flexible cms if you have to use it as a dev  not great for clients though
Text: @emoskank awww   take him with you!
Processed:  aww  take him with you
```

What are the modeling approaches:

1. Logistic Regression: Logistic Regression is a linear model used for binary classification tasks. Despite its name, it's a classification algorithm rather than a regression algorithm. It models the probability of the input belonging to a particular class using the logistic function.

- **Implementation**: Implemented Logistic Regression using scikit-learn, a popular machine learning library in Python. The model is trained using the training data and then evaluated on the test data. No complex architecture is involved, making it computationally efficient.
- **Strengths**:
    - Simple and interpretable model, making it easy to understand.
    - Efficient for small datasets and low-dimensional feature spaces.
    - Less prone to overfitting, especially with regularization.
- **Weaknesses**:
    - Assumes linear relationship between features and target.
    - Limited capacity to capture complex patterns in data.
    - Can be sensitive to outliers and noise in the data.

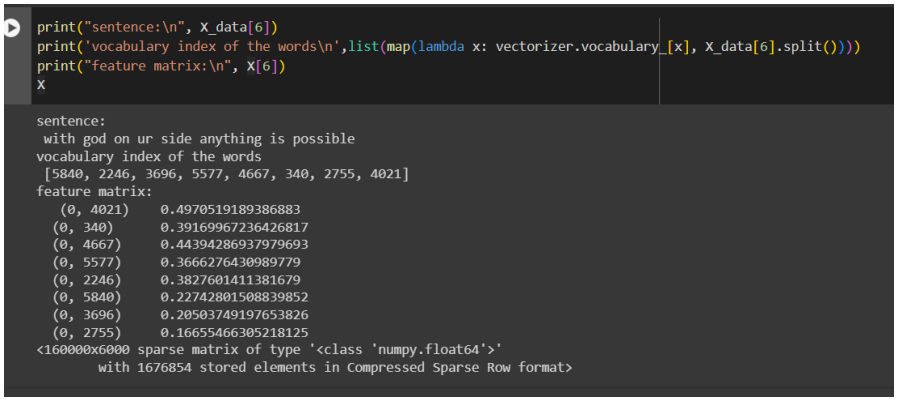
2. LSTM (Long Short-Term Memory): LSTM is a type of recurrent neural network (RNN) architecture designed to overcome the vanishing gradient problem. It's well-suited for sequential data analysis, such as natural language processing (NLP), time series forecasting, and speech recognition.

- Implementation: Implemented LSTM using Keras, a high-level neural networks API running on top of TensorFlow. The model includes an embedding layer followed by LSTM layers, global max pooling, and a dense layer with sigmoid activation for binary classification.
- Strengths:
    - Captures long-term dependencies in sequential data.
    - Handles variable-length input sequences efficiently through padding.
    - Suitable for complex relationships between features.
- Weaknesses:
    - Requires large amounts of data to train effectively.
    - Computationally intensive compared to simpler models.
    - Prone to overfitting, especially with limited data.

How the models were built:

Implementation of Logistic Regression:

TF-IDF vectorization: I utilized the TF-IDF vectorization technique with a maximum of 6000 features to transform the tweet data into numerical format suitable for machine learning. This involved initializing the TfidfVectorizer object and applying the fit_transform() method to convert the raw text data into a TF-IDF feature matrix. By limiting the number of features, I controlled the dimensionality of the matrix, which helps manage complexity and potentially improves model performance. Overall, TF-IDF vectorization effectively prepared the tweet data for sentiment analysis tasks. Here is an example of how TF-IDF converts a sentence to a feature matrix.



```
print("sentence:\n", X_data[6])
print('vocabulary index of the words\n',list(map(lambda x: vectorizer.vocabulary_[x], X_data[6].split())))
print("feature matrix:\n", X[6])
X
```

```
sentence:
 with god on ur side anything is possible
vocabulary index of the words
 [5840, 2246, 3696, 5577, 4667, 340, 2755, 4021]
feature matrix:
   (0, 4021)    0.4970519189386883
  (0, 340)      0.39169967236426817
  (0, 4667)     0.44394286937979693
  (0, 5577)     0.3666276430989779
  (0, 2246)     0.3827601411381679
  (0, 5840)     0.22742801508839852
  (0, 3696)     0.20503749197653826
  (0, 2755)     0.16655466305218125
<160000x6000 sparse matrix of type '<class 'numpy.float64'>'
        with 1676854 stored elements in Compressed Sparse Row format>
```

Splitting data: I split the dataset into training and testing sets using the train_test_split function from scikit-learn. The dataset was divided into 80% for training and 20% for testing. This division helps in evaluating the performance of the model on unseen data.

Model: Then, I instantiated a Logistic Regression model (model) with the max_iter parameter set to 1000 to ensure convergence during training as it was giving warning about it. Next, I trained the model using the training data with the fit method. This process involved learning the relationship between the input features (X_train_lr) and the target labels (y_train_lr). After training the model, I made predictions on the testing data (X_test_lr) using the predict method. The predicted labels were stored in y_pred_lr.

Evaluation: Finally, I evaluated the performance of the Logistic Regression model using the classification_report function from scikit-learn. This report provides a comprehensive summary of various evaluation metrics such as precision, recall, F1-score, and support for each class, as well as the overall accuracy. It helps in assessing how well the model performs in classifying the data into different classes.

```
62] model = LogisticRegression()
    model.fit(X_train_lr, y_train_lr)
    y_pred_lr = model.predict(X_test_lr)
    print(classification_report(y_test_lr, y_pred_lr))
```

	precision	recall	f1-score	support
0	0.79	0.77	0.78	16002
1	0.78	0.80	0.79	15998
accuracy			0.78	32000
macro avg	0.78	0.78	0.78	32000
weighted avg	0.78	0.78	0.78	32000

<u>Implementation of LSTM:</u>

Splitting data:

- I splitted the dataset into training and testing by keeping 10% for the testing.

Word Embeddings using Word2Vec Model:

- I imported the Word2Vec module from the gensim library.
- With a specified embedding dimension of 100, I prepared the data for Word2Vec training by splitting the text sequences into individual words.
- Utilizing the Word2Vec function, I trained the model on the prepared dataset, setting parameters such as vector size, number of workers, and minimum count.
- Upon completion of training, I checked the vocabulary length of the Word2Vec model.

Tokenization and Padding:

- I initialized a Tokenizer object from the Keras library, configuring it to disregard filters and maintain case sensitivity, while specifying a maximum vocabulary length to 60,000.

- After fitting the Tokenizer on the training data, I converted the text sequences into integer sequences using the trained Tokenizer.

```
X_train_sequences = tokenizer.texts_to_sequences(X_train_lstm)
X_test_sequences = tokenizer.texts_to_sequences(X_test_lstm)

print(X_train_lstm[4])
print(X_train_sequences[4])
```

```
got to hurry to the train station  damn  no time to work out today
[42, 3, 1751, 3, 4, 572, 1498, 229, 40, 52, 3, 47, 38, 44]
```

- To ensure uniform sequence length, I padded both the training and test sequences to a maximum length of 60 tokens using the pad_sequences function.

```
input_length = 60

X_train_lstm = pad_sequences(X_train_sequences, maxlen=input_length)
X_test_lstm  = pad_sequences(X_test_sequences , maxlen=input_length)

print("X_train.shape:", X_train_lstm.shape)
print("X_test.shape :", X_test_lstm.shape)
X_train_lstm[10], X_test_lstm[10]
```

```
X_train.shape: (144000, 60)
X_test.shape : (16000, 60)
(array([    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,   519, 16858,    28, 21013,   451,
        11396,  1507,   641,    13,  1879,    50,    20,  2224,    21,
           55,   975,    25,   468,    72,    37], dtype=int32),
```

Embedding Matrix Creation:

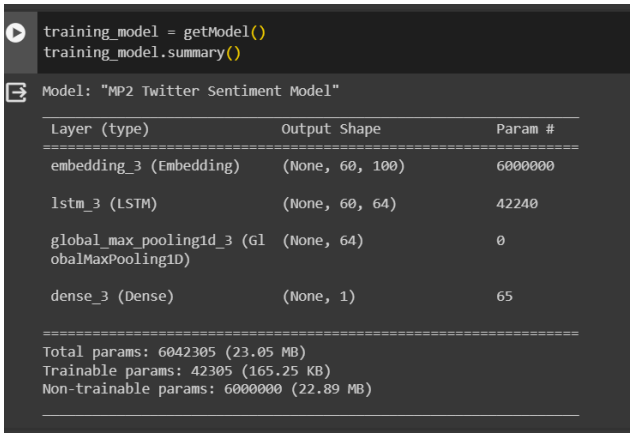
- I created an embedding matrix of zeros with dimensions corresponding to the vocabulary length (60,000) and embedding dimensions (100).
- I populated the embedding matrix with the learned word embeddings from the Word2Vec model. The trained word vectors are stored in a KeyedVectors instance, named model.wv
- Here is the embedding matrix and I printed the 100 feature values of a word.

```
embedding_matrix = np.zeros((vocab_length, Embedding_dimensions))

for word, token in tokenizer.word_index.items():
    if word2vec_model.wv.__contains__(word):
        embedding_matrix[token] = word2vec_model.wv.__getitem__(word)

print("Embedding Matrix Shape:", embedding_matrix.shape)
embedding_matrix[50]
```

```
Embedding Matrix Shape: (60000, 100)
array([-0.13217646, -0.53420472,  0.05173017,  1.0970484 ,  0.20852919,
       -0.95836043,  0.53415745,  0.76773465, -0.25055051, -1.02038682,
       -0.20266728, -0.02624628, -0.31316924,  0.65396422,  0.48232552,
        0.61620218,  0.02007193,  0.91078067, -0.23476641, -0.05137894,
        2.1802671 ,  0.29784936, -0.63429046, -0.51827842,  0.3110787 ,
       -0.083652  , -0.29119694, -0.90277046, -0.43691009, -0.73866957,
       -1.07728481,  0.07886416,  0.54075277, -0.7503078 ,  0.14311247,
       -0.6598776 ,  0.31625277,  0.124787  ,  0.04142545, -0.86332262,
        0.36647522,  0.6402663 ,  0.40681115,  0.05920042, -1.27501988,
        0.40247217, -0.85293829,  0.42725411,  0.88640165,  0.22492962,
       -0.95598346, -0.38966811, -0.19993624, -0.32720366, -0.14770451,
        0.16428892,  0.1309043 , -0.08664328, -0.17477344,  0.26601785,
       -0.27102941,  0.34752831, -1.10388076,  0.41199467,  0.85658699,
       -0.13719624, -0.22570992,  1.25856185, -0.4463349 , -0.58603227,
        0.1483013 , -0.20899759,  0.57676721, -0.43157274,  0.07645549,
       -0.13463914, -0.22048216, -0.32385284, -0.09056717, -0.05784751,
       -0.56337857,  0.86184269,  0.08029959,  0.22367013, -0.16255032,
        0.60086626, -0.47792181, -0.49057972, -0.37140197, -0.08995006,
        0.95099121, -0.02215476,  0.29043624,  1.6338762 ,  0.36994305,
       -0.32092372, -1.53986251, -0.34268329, -0.20670593,  0.40559506])
```

Model Architecture:

- I defined a Sequential model using the Keras API, comprising layers of Embedding, LSTM, GlobalMaxPool1D, and Dense.
- **Embedding Layer**: This layer is responsible for converting tokens into their vector representations using the Word2Vec model. The embedding layer was configured to utilize the pre-trained embedding matrix while keeping it non-trainable to retain the learned word representations.
- **LSTM(Long Short Term Memory)**: LSTM is a type of recurrent neural network (RNN) that has memory cells to learn the context of words. I used dropout to reduce overfitting
- **GlobalMaxPool1D**: This layer downsamples the input representation by taking the maximum value over the different dimensions. It is commonly used to reduce the dimensionality of the input and extract the most important features.
- **Dense Layer**: Adds fully connected layer, with Sigmoid activation for binary classification.



```
training_model = getModel()
training_model.summary()
```

```
Model: "MP2 Twitter Sentiment Model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (None, 60, 100)           6000000

 lstm_3 (LSTM)               (None, 60, 64)            42240

 global_max_pooling1d_3 (Gl  (None, 64)                0
 obalMaxPooling1D)

 dense_3 (Dense)             (None, 1)                 65

=================================================================
Total params: 6042305 (23.05 MB)
Trainable params: 42305 (165.25 KB)
Non-trainable params: 6000000 (22.89 MB)
_________________________________________________________________
```

Model Training:

- I initialized callbacks including ReduceLROnPlateau and EarlyStopping to stop training when a monitored metric has stopped improving.
- Using the fit method, I trained the sentiment analysis model on the padded training sequences, specifying parameters such as batch size, epochs, validation split, callbacks, and verbosity.
- Throughout the training process, I monitored the progress by storing accuracy, validation accuracy, loss, and validation loss metrics in the history object.
- Finally, I obtained the training and validation accuracy, as well as loss metrics, to evaluate the performance of the trained model.
- Validation accuracy appears to be higher than training because of used dropout in LSTM layer.

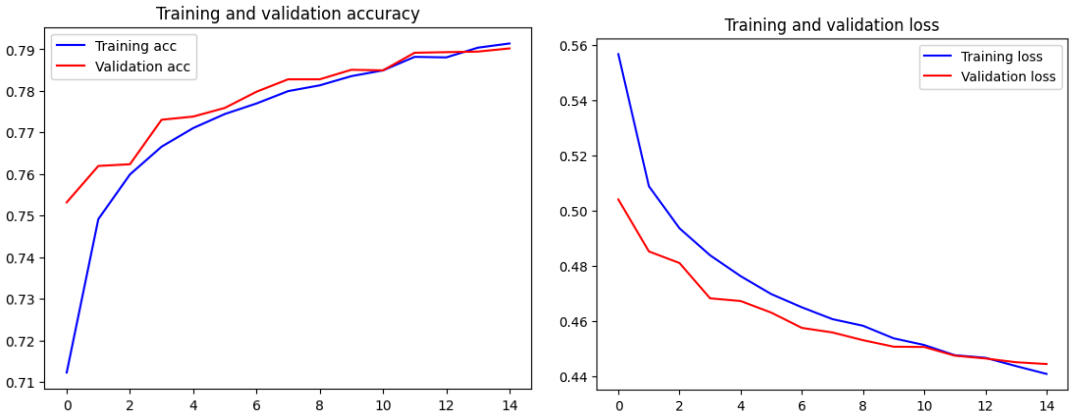


Fig: Accuracy and loss of LSTM

Comparison of model performance:

- In my case, **both LSTM and Logistic Regression models produced very similar results** across various evaluation metrics, including precision, recall, F1-score, and accuracy. This outcome suggests that, for the specific sentiment analysis task I'm working on, the complexity introduced by the LSTM architecture may not have provided significant advantages over the simpler Logistic Regression model.
  Several factors could contribute to this scenario:
- **Simplicity of the Task**: The sentiment analysis task may involve relatively straightforward patterns and dependencies within the text data. In such cases, the additional complexity of LSTM models may not be necessary to achieve good performance, and simpler models like Logistic Regression can suffice.
- **Data Size and Quality**: If the dataset used for sentiment analysis is limited in size or lacks subtle patterns, both models may perform similarly regardless of their architectural differences. Additionally, if the dataset contains noise or inconsistencies, both models may struggle equally to generalize well to unseen data.
- **Feature Representation**: It's possible that the features captured by both models, whether learned automatically by the LSTM or derived from simple transformations in Logistic Regression, are sufficiently informative for the sentiment analysis task. In other words, the differences in feature representation between the models may not significantly impact their performance.

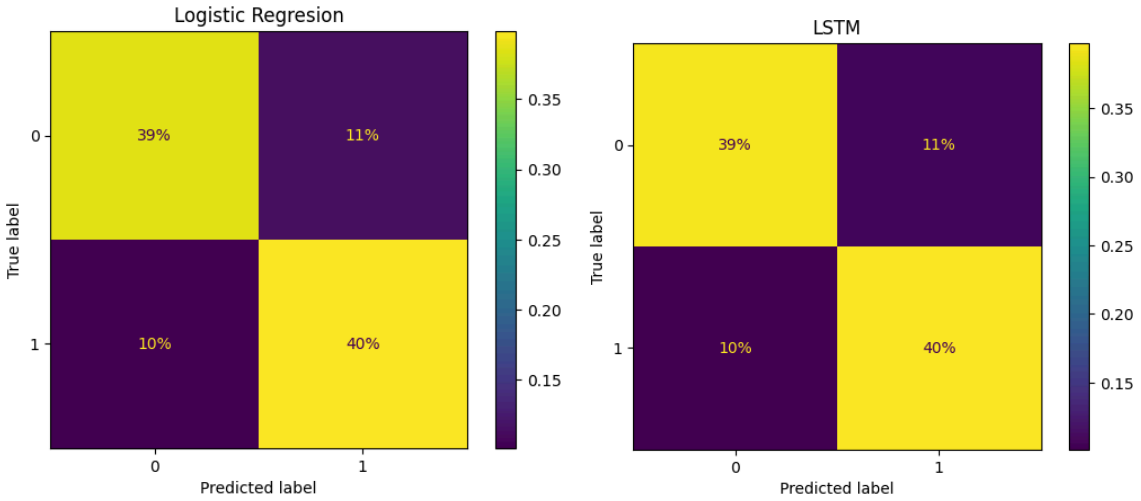


Fig: Normalized confusion matrix of logistic regression and LSTM

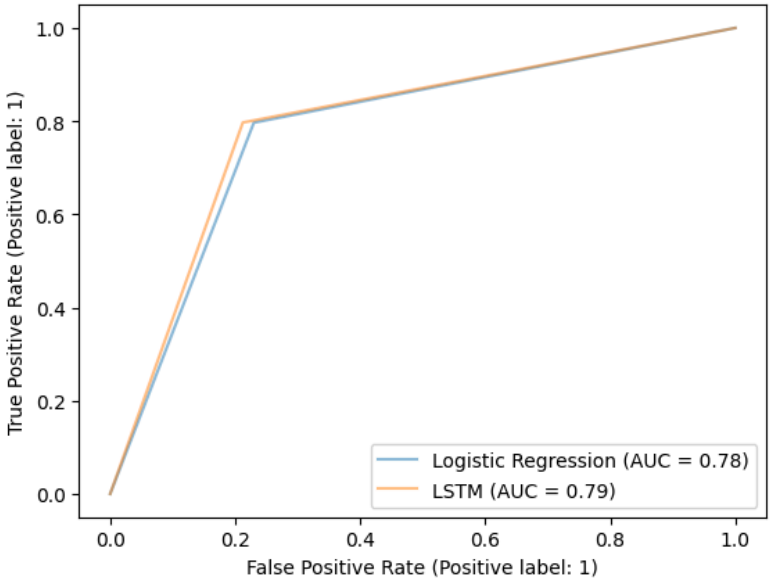


Fig: ROC curve of Logistic regression and LSTM

To enhance model performance:

- **Feature Engineering**: Exploring additional features or alternative representations to better capture underlying patterns in the data may improve accuracy
- **Hyperparameter Tuning**: The performance of both models may be limited by suboptimal hyperparameters or configurations. Fine-tuning hyperparameters, such as regularization strength, learning rates, or model architecture, could potentially yield better performance for one or both models.
- **Model Complexity**: Experimenting with deeper LSTM architectures, incorporating bidirectional LSTMs, adding more layers, or using attention mechanisms could further

enhance the LSTM model's ability to capture intricate patterns in the data. Similarly, considering more complex variants of logistic regression, such as polynomial logistic regression may improve the accuracy.

- **Fine-tuning Word Embeddings**: Fine-tuning pre-trained word embeddings or training word embeddings specifically on the sentiment analysis dataset may improve the quality of representations learned by the models.

Conclusions:

During the sentiment analysis task using both LSTM and Logistic Regression models, several scientific bottlenecks were encountered and addressed:

- **Model Selection**: Choosing the appropriate model architecture posed a bottleneck initially. I tried many different models but all were giving very poor accuracy. After trying a lot, I found LSTM and logistic regression to be more accurate. To overcome this, many different models were implemented and evaluated to determine the most suitable approach.
- **Feature Representation**: Another bottleneck was determining the optimal representation of text data as features. Tokenization and embedding techniques were explored to convert raw text into numerical inputs suitable for the models. This required experimentation to find the most informative feature representation.
- **Parameter Tuning:** Parameters such as epochs, dropout rates etc. were essential for optimizing performance. I tried different values to achieve the obtained accuracy.
- **Data Quality and Quantity**: The quality and quantity of the available data were significant bottlenecks. Insufficient or noisy data could lead to poor model performance. Data preprocessing techniques, such as cleaning, handling missing values, and balancing class distributions, were applied to mitigate these issues.