Progress report :- ML Project

EMOTION DETECTION USING TEXT

Phase - 1

Our data is distributed in total 8 classes:
['neutral', 'joy', 'sadness', 'fear', 'surprise', 'anger', 'shame', 'disgust']

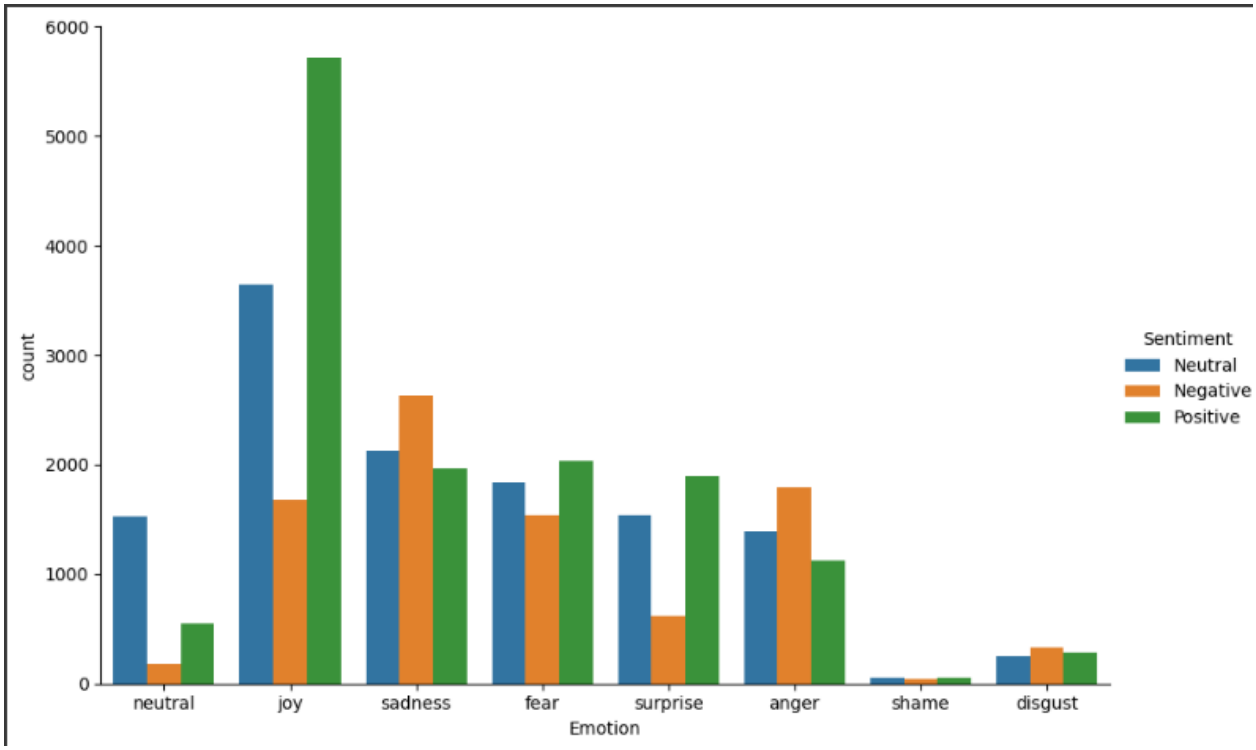
Very first we did sentiment analysis using **TEXTBLOB** (A python library for processing textual data) got to know the sentiment of the sentences, Now plotted the sentiment graphs of each classes and Interpreted that for joy emotion most of the statements are positive , similarly for neutral emotion most statements are neutral and for sadness, anger , disgust most of the statements are negative.

```python
# Sentimental Anaylisi
from textblob import TextBlob
# Function to perform sentiment analysis using TextBlob
def get_sentiment(text):
    # Create a TextBlob object to analyze sentiment in the given text
    blob = TextBlob(text)

    # Get the polarity score for sentiment analysis
    sentiment = blob.sentiment.polarity

    # Determine the sentiment based on the polarity score
    if sentiment > 0:
        result = "Positive"
    elif sentiment < 0:
        result = "Negative"
    else:
        result = "Neutral"

    # Return the sentiment result
    return result
```



Now we used a lot of graphs and plots for data visualization in which **WordCloud (Basically it gives graphical representation of words based on the frequency of the words)** is nice for the graphical representation and got to know that there is a lot of data preprocessing needed.

Preprocessing steps used : - Using neattext module

1. Lowercasing: Convert all text to lowercase. This ensures uniformity and reduces the complexity of the data.

2. Removing Punctuation: Remove any special characters and punctuation marks from the tweets as they don't usually contribute significantly to emotion detection.

3. Removing Stopwords: Remove common stop words (e.g., 'and', 'is', 'in') that do not carry significant meaning and are often noise in text data.

4. Handling Emoticons and Special Characters: Emoticons and special characters might carry emotional meaning. You can choose to keep, remove, or replace them based on the context of your analysis.

5. Handling URLs and User Mentions: Remove or replace URLs and user mentions (starting with '@') as they usually don't contribute to emotion detection.

6. Handling Numbers: Decide whether to keep numbers, convert them to words, or remove them based on the context of your analysis.

Here, we used **CountVectorizer** to convert the text data into numerical representation and that we have used in all the 3 models discussed below,

```
[48] from sklearn.feature_extraction.text import CountVectorizer
     from sklearn.pipeline import Pipeline
     from sklearn.linear_model import LogisticRegression

     # Create a pipeline with CountVectorizer and a classifier (e.g., Logistic Regression)
     pipe_lr = Pipeline([
         ('cv', CountVectorizer()),
         ('lr', LogisticRegression())
     ])
```

We used 3 different models :-

1. Logistic regression
   Score after using Logistic regression : `0.6054799770070894`

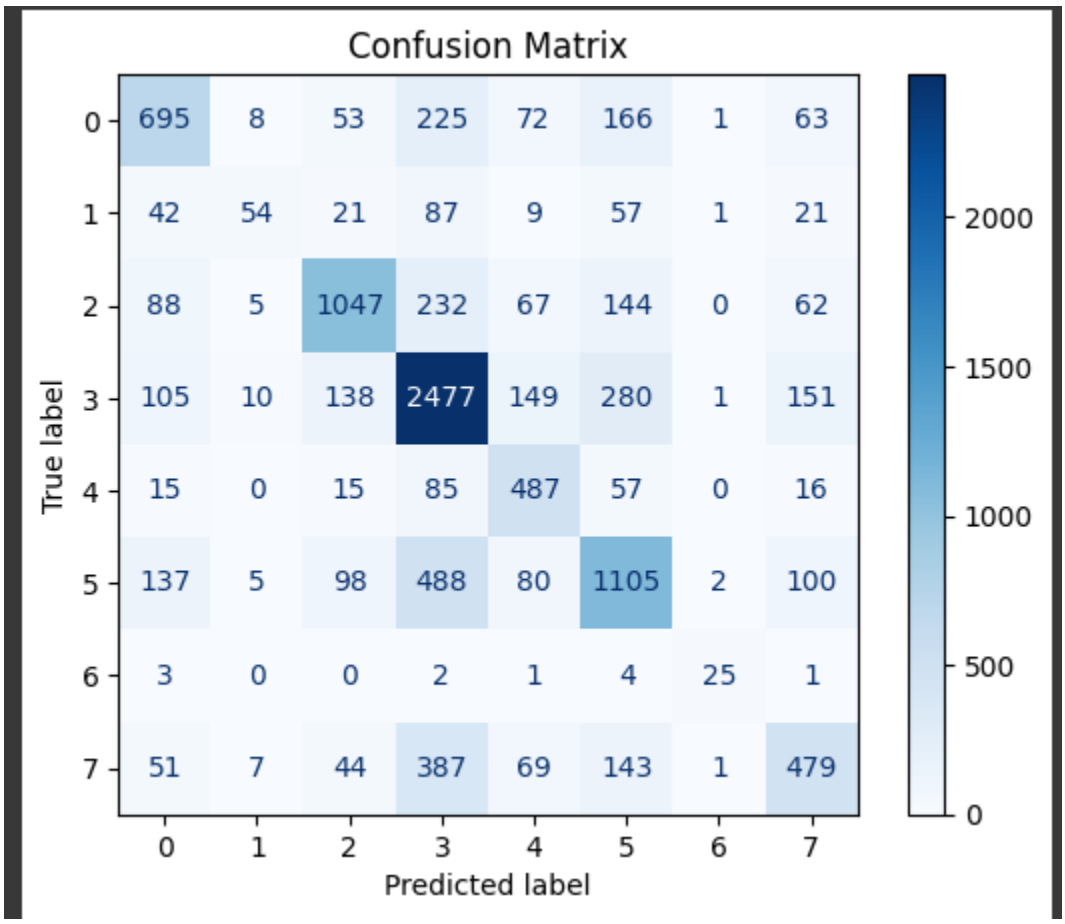
2. SVM
   Score using SVM : `0.5966660279747078`

3. Random Forest
   Score using random forest : `0.558919333205595`

For the current models the Pipeline is complete from taking the test cases , data preprocessing

Countvectorising and model being used to predict.

In all the three models above the score is almost around 55-60%. Which is decent but after looking at the confusion matrix we can see that most of the predictions are just joy sentiment (30% of ~35000 cases which is approximately 10500) and the model is predicting almost 5500 to be joy sentiment. Which is the reason for the low score and inconsistent confusion matrix.



Phase - 2

Model-1

Here, We did data preprocessing using **NLTK (Natural Language Toolkit),**
Preprocessing steps used : - Using NLTK

```
#text preprocessing
ps = PorterStemmer()

def preprocess(line):
    review = re.sub('[^a-zA-Z]',' ',line) #leave only charactes from a to z
    review = review.lower() #lower the text
    review = review.split() #turn string into list of words
    #apply stemming
    review = [ps.stem(word) for word in review if not word in stopwords.words('english')] #del
    #turn list into sentences
    return " ".join(review)
```

1. Leave only the character's from a to z.

2. Lowercasing: Convert all text to lowercase. This ensures uniformity and reduces the complexity of the data.

3. Apply stemming : The primary purpose of stemming is to reduce words to a common base or root, Reducing Variability, Improving Text Matching , Reducing Dimensionality are the basic reasons for applying stemming.

Here, we used **CountVectorizer** to convert the text data into numerical representation,

Now, You can see our Sequential Model with 5 Layers using "relu" and "softmax" functions,

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(12,input_shape=(x_train.shape[1],),activation="relu"))
model.add(Dense(8,activation="relu"))
model.add(Dense(8,activation="relu"))
model.add(Dense(8,activation="relu"))
model.add(Dense(8,activation="softmax"))

model.compile(loss="sparse_categorical_crossentropy",optimizer= "adam", metrics=['accuracy'])

model.fit(x_train,y_train,epochs= 10,batch_size=16)
_,accuracy = model.evaluate(x_train,y_train)
print("Accuracy: %.2f" % (accuracy*100))
```

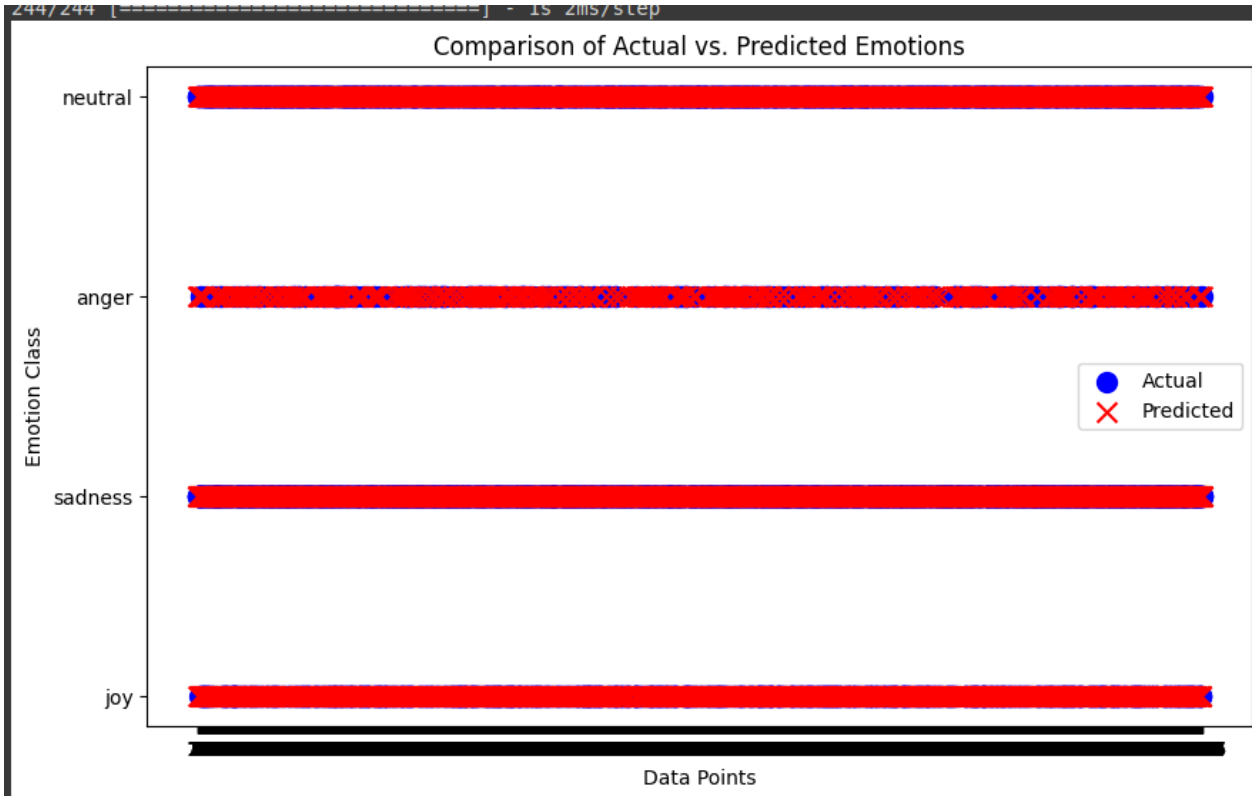
```
Epoch 1/10
1631/1631 [==============================] - 9s 5ms/step - loss: 1.5352 - accuracy: 0.4231
Epoch 2/10
1631/1631 [==============================] - 6s 4ms/step - loss: 1.1471 - accuracy: 0.6031
Epoch 3/10
1631/1631 [==============================] - 8s 5ms/step - loss: 0.9878 - accuracy: 0.6726
Epoch 4/10
1631/1631 [==============================] - 7s 4ms/step - loss: 0.8823 - accuracy: 0.7031
Epoch 5/10
1631/1631 [==============================] - 8s 5ms/step - loss: 0.7915 - accuracy: 0.7272
Epoch 6/10
1631/1631 [==============================] - 7s 4ms/step - loss: 0.7119 - accuracy: 0.7560
Epoch 7/10
1631/1631 [==============================] - 8s 5ms/step - loss: 0.6421 - accuracy: 0.7804
Epoch 8/10
1631/1631 [==============================] - 9s 6ms/step - loss: 0.5798 - accuracy: 0.8034
Epoch 9/10
1631/1631 [==============================] - 7s 4ms/step - loss: 0.5231 - accuracy: 0.8236
Epoch 10/10
1631/1631 [==============================] - 7s 4ms/step - loss: 0.4768 - accuracy: 0.8402
816/816 [==============================] - 3s 4ms/step - loss: 0.3910 - accuracy: 0.8732
Accuracy: 87.32
```

Training accuracy : 87.32
Test accuracy : 56.98

Which completely show's that the model is overfitting. Hence we should use **Dropout() (dropout helps to regularize the neural network by preventing it from relying too heavily on specific neurons or combinations of neurons, leading to a more generalized and robust model.)**

Now,
Training accuracy: 85.54
Test accuracy : 58.34

Now let's merge the similar emotions :
So, Replace shame by sadness, fear by sadness, surprise by joy, disgust by anger.
Now we have a total of 4 classes named: **joy, sadness , anger , and neutral.**



Now we trained our same model on the new changed data,
And we got the:

Training accuracy : 76.54
Test accuracy : 65.67

Model -2 Using LSTM's

Here, We again did text preprocessing on original data and tokenize all the words/characters:

```
print('Before: ')
print(df.head())

x_train = [text_preprocess(t, stop_words=True) for t in train['Text']]
y_train = train['Labels'].values

print('\nAfter:')
for line_and_label in list(zip(x_train[:5], y_train[:5])):
  print(line_and_label)
```

```
Before:
   Emotion                                               Text  Labels
0  neutral                                             Why ?        4
1      joy    Sage Act upgrade on my to do list for tommorow.       3
2  sadness  ON THE WAY TO MY HOMEGIRL BABY FUNERAL!!! MAN ...       5
3      joy   Such an eye ! The true hazel eye-and so brill...       3
4      joy  @Iluvmiasantos ugh babe.. hugggzzz for u .!  b...       3

After:
(['yes', 'to', 'our', 'first', 'anniversary', 'and', 'many', 'more', 'to', 'come', 'cheers'], 3
(['don', 't', 'you', 'think', 'it', 's', 'too', 'bloody'], 4)
(['peace', 'is', 'that', 'state', 'in', 'which', 'fear', 'of', 'any', 'kind', 'is', 'unknown',
(['when', 'fat', 'bitches', 'complain', 'about', 'their', 'weight', 'no', 'one', 'told', 'you',
(['heb', 'een', 'lolly', 'gekregen', 'met', '12', 'lolly', 's', 'erin', 'en', '18', 'chocolade'
```

As we have used LSTM so our main task is to convert the text to numerical representation and additionally find the **weight matrix** (LSTM).

As we just tokenize into words now we replaced each word by corresponding index and padded to the MAX_LEX.

And for weight matrix we downloaded

```
# load pre-trained model
import gensim.downloader as api
model_wiki = api.load('fasttext-wiki-news-subwords-300')
```

Fasttext-wiki-news-subwords-300 A famous Neural Network pretrained model on the millions of articles of WIKIPEDIA.

We also used **`Word2Vec`**

```
from gensim.models import Word2Vec

# train word2vec model on the corpus
model_w2v = Word2Vec(x_train + x_test + x_validation,   # data for model to train on
                     vector_size = 300,                 # embedding vector size
                     min_count = 2).wv
```

Now you can find the Weight matrix using Both the above mentioned model's.

```
[ ]  def create_weight_matrix(model, second_model=False):
       '''
       Accepts word embedding model
       and the second model, if provided
       Returns weight matrix of size m*n, where
       m - size of the dictionary
       n - size of the word embedding vector

       '''
       vector_size = model.get_vector('like').shape[0]
       w_matrix = np.zeros((DICT_SIZE, vector_size))
       skipped_words = []

       for word, index in tokenizer.word_index.items():
         if index < DICT_SIZE:
           if word in model.key_to_index:
             w_matrix[index] = model.get_vector(word)
           else:
             if second_model:
               if word in second_model.key_to_index:
                 w_matrix[index] = second_model.get_vector(word)
               else:
                 skipped_words.append(word)
             else:
               skipped_words.append(word)

       print(f'{len(skipped_words)} words were skipped. Some of them:')
       print(skipped_words[:50])
       return w_matrix
```

```
[ ]  weight_matrix = create_weight_matrix(model_wiki, model_w2v)
```

```
0 words were skipped. Some of them:
[]
```

Now you can find our LSTM model below:
With Training accuracy : 69.74 and Test accuracy : 63.30

```
[ ] # initialize sequential model
    model = Sequential()
    model.add(Embedding(input_dim = DICT_SIZE, # the whole vocabulary size
                        output_dim = weight_matrix.shape[1], # vector space dimension
                        input_length = X_train_pad.shape[1], # max_len of text sequence
                        weights=[weight_matrix], # assign the embedding weight with embedding mari
                        trainable=False)) # set the weight to be not trainable (static)
```

```
[ ] model.add(Bidirectional(LSTM(64, return_sequences=True)))
    model.add(Dropout(0.2))
    model.add(Bidirectional(LSTM(128, return_sequences=True)))
    model.add(Dropout(0.2))
    model.add(Bidirectional(LSTM(64, return_sequences=False)))
    model.add(Dense(8, activation = 'softmax'))
    model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics='accuracy')
```

```
history = model.fit(X_train_pad, y_train,
                    validation_data = (X_val_pad, y_validation),
                    batch_size = 64,
                    epochs = 20,
                    callbacks = stop)
```

```
===] - 40s 68ms/step - loss: 1.4652 - accuracy: 0.4473 - val_loss: 1.2963 - val_accuracy: 0.5260
===] - 27s 62ms/step - loss: 1.2408 - accuracy: 0.5532 - val_loss: 1.2116 - val_accuracy: 0.5654
===] - 28s 65ms/step - loss: 1.1552 - accuracy: 0.5887 - val_loss: 1.1427 - val_accuracy: 0.5847
===] - 27s 62ms/step - loss: 1.1081 - accuracy: 0.6034 - val_loss: 1.1210 - val_accuracy: 0.5999
===] - 28s 65ms/step - loss: 1.0631 - accuracy: 0.6177 - val_loss: 1.1039 - val_accuracy: 0.6082
===] - 28s 64ms/step - loss: 1.0251 - accuracy: 0.6346 - val_loss: 1.0897 - val_accuracy: 0.6019
===] - 27s 62ms/step - loss: 0.9895 - accuracy: 0.6439 - val_loss: 1.0673 - val_accuracy: 0.6180
===] - 28s 65ms/step - loss: 0.9550 - accuracy: 0.6537 - val_loss: 1.0564 - val_accuracy: 0.6214
===] - 28s 65ms/step - loss: 0.9249 - accuracy: 0.6660 - val_loss: 1.0169 - val_accuracy: 0.6289
===] - 28s 65ms/step - loss: 0.8943 - accuracy: 0.6769 - val_loss: 1.0506 - val_accuracy: 0.6243
===] - 28s 65ms/step - loss: 0.8618 - accuracy: 0.6880 - val_loss: 1.0216 - val_accuracy: 0.6306
===] - 27s 62ms/step - loss: 0.8347 - accuracy: 0.6974 - val_loss: 1.0249 - val_accuracy: 0.6309
```

```
[ ] model.evaluate(X_test_pad, y_test)
```

```
109/109 [==============================] - 2s 22ms/step - loss: 1.0861 - accuracy: 0.6330
[1.0861282348632812, 0.6330459713935852]
```