Name: 胡紹良

StudentID: 110006202

LAB 2 HOMEWORK REPORT

PREPROCESSING STEPS

1. **Load the json data into panda dataframe**

First I load the json data into panda df by using a built in function from panda called read_json. However, the json file was nested so I had to use another function called json_normalize that can flatten the nested file so I can directly extract the values from the data. After that I did some data cleaning to make the dataset easier to read by renaming the features.

2. **Load the csv files containing the type of data and the emotions then combine them to the panda df**

3. **Data preprocessings**

For this step, I used tqdm library so I can track the progress of the text cleaning

```
De-emojify the emojis

    df['text'] = df['text'].progress_apply(lambda x: emoji.demojize(x)) # convert emojis to sentences

Cleaning text: 100%|██████████| 1867535/1867535 [01:34<00:00, 19747.75it/s]
```

The first thing I did was de-emojify the text, the library emoji provides a function that can convert emojis into sentences, I find this helps increase the model's performance

```
Remove punctuations

    df['text'] = df['text'].progress_apply(lambda x: x.replace("'", ' ').replace('"',' ').replace(':', ' ').replace('.',' ').replace(',',' ').replace('',' ')) #remove punctuations

Cleaning text: 100%|████████| 1867535/1867535 [00:02<00:00, 846092.98it/s]
```

After that I removed all the punctuations that doesn't provide any contexts to the sentence and the punctuations makes the TFIDF that I used to calculate the words differently, since the word "fun" and "fun!" are considered two different words.

```
make all text lowercase

    df['text'] = df['text'].str.lower() # convert train_df into lowercase
```

Lastly, I converted all the text into lowercase for the same reason as above

4. **Split the df into train_df and test_df**

```
1.4 split the dataframe into training and testing dataframe

    #split the df based on its identifications
    train_df = df[df['identification']=='train']

    test_df = df[df['identification']=='test']
```

I split the df based on whether identification is train or test

FEATURE ENGINEERING

2.2 tf-IDF vectorizer for training and testing data

```
TFIDF = TfidfVectorizer(max_features = 30000, tokenizer=nltk.word_tokenize, ngram_range=(1,2), min_df = 15, max_df = 0.95) # use 30000 features & 1-2 Ngrams

# Apply analyzer to train data
TFIDF.fit(train_df['text'])

# Transform documents to document-term matrix.
train_data_TFIDF = TFIDF.transform(train_df['text']) #for train_df

test_data_TFIDF = TFIDF.transform(test_df['text']) #for test_df
```

For the feature engineering, I mainly used TfidfVectorizer, the details of my parameters are: 30000 features, 1-2 ngram range, 15 min_df, and 0.95 max_df, I got these numbers from trying out the parameters to find which one produces the best outcome.

I trained the TFIDF model only on the train dataset because it imitates how the model will work when it encounters new words like in the testing data.

Using one-hot encoding to transfrom train_val into numerical data

```
label_encoder = LabelEncoder()
label_encoder.fit(train_val)
print('check label: ', label_encoder.classes_)
print('\n## Before convert')
print('y_train[0:4]:\n', train_val[0:4])
print('\ny_train.shape: ', train_val.shape)
print('y_test.shape: ', train_val.shape)

def label_encode(le, labels):
    enc = le.transform(labels)
    return keras.utils.to_categorical(enc)

def label_decode(le, one_hot_label):
    dec = np.argmax(one_hot_label, axis=1)
    return le.inverse_transform(dec)

train_val = label_encode(label_encoder, train_val)


print('\n\n## After convert')
print('y_train[0:4]:\n', train_val[0:4])
print('\ny_train.shape: ', train_val.shape)
print('y_test.shape: ', train_val.shape)
```

As for the labels, I used the one hot encoding from the master

Split train data into X_train and X_test

```
X_train, X_test, y_train, y_test = train_test_split(
    train_data_TFIDF, train_val, test_size=0.2, random_state=42)
```

And to try out the model, I split the train data into X_train and X_test with 20% split

MODEL

3.1 I/O check

```
# I/O check
input_shape = X_train.shape[1]
print('input_shape: ', input_shape)

output_shape = len(label_encoder.classes_)
print('output_shape: ', output_shape)
```

```
input_shape:  30000
output_shape:  8
```

Using Neural Network with layer like in the homework

```
# input layer
from tensorflow.keras.layers import Input, Dense, Dropout, ReLU, Softmax
model_input = Input(shape=(input_shape, ))  # input shape
X = model_input

# 1st hidden layer
X_W1 = Dense(units=128)(X)  # 128
H1 = ReLU()(X_W1)
# 2nd hidden layer
H1_W2 = Dense(units=128)(H1)  # 128
H2 = ReLU()(H1_W2)
# output layer

H2_W3 = Dense(units=128)(H2)  # 128
H3 = ReLU()(H2_W3)

H3_W4 = Dense(units=output_shape)(H3)  # 8
H4 = Softmax()(H3_W4)

model_output = H4

# create model
model = Model(inputs=[model_input], outputs=[model_output])

# loss function & optimizer
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# show model construction
model.summary()
```

For the model I used neural network like we did in the master, but I increased the number of hidden layers to 3 and each layer has 128 neurons, so it is twice as much as the one we did in the master

3.2 Train the model

Neural Network

```
csv_logger = CSVLogger('logs/training_log.csv')

# training setting
epochs = 2
batch_size = 512

# training! (with split)
history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, callbacks=[csv_logger],validation_data = (X_test, y_test))

# training (without split)
#history = model.fit(train_data_TFIDF, train_val, epochs = epochs, batch_size = batch_size)
print('training finish')
```

To train the model I used only 2 epochs and 512 batch size, as this model is very prone to overfitting. And I have two versions: the first model utilizes the training data split to closely monitor the performance of the model, while the second model uses all of the training data to be used as the submission file.

The process is very similar to what I did in the master.

3.3 Predict on testing data & check accuracy

```
pred_result = model.predict(X_test, batch_size=128)
pred_result = label_decode(label_encoder, pred_result)
print('testing accuracy: {}'.format(round(accuracy_score(label_decode(label_encoder, y_test), pred_result), 2)))
```

```
2275/2275 ━━━━━━━━━━━━━━━━━━━━ 14s 6ms/step
testing accuracy: 0.71
```

```
pred_result = model.predict(test_data_TFIDF, batch_size=128)
pred_result = label_decode(label_encoder, pred_result)

submission = test_df.drop(columns = ['score', 'crawldate', 'text', 'hashtags']) # drop columns that are not in submission file
submission['emotion'] = pred_result # store the result in emotion column
submission = submission.reset_index() # reset index for tweets_id
submission.rename(columns = {'tweet_id':'id'}, inplace = True) # change the index name to id
submission.to_csv('submission.csv', index = False) # save csv file
```

And lastly for the submission, I created a new panda df from the test_df , aggregate the result of the prediction to this new dataset then save it into a csv file called submission.csv

WHAT I TRIED BESIDES THIS

1. **Using simpler model like NaïveBayes**
   This failed because the model is too simple to be able to extract the useful information from tfidf
2. **Using Word2Vec, FastText and then averaging the embeddings**

This failed because averaging the embeddings doesn't necessarily help in determining the sentiment of the text, so I received a lower accuracy

3. **Using BERT and LLAMA embeddings**
   This failed because these models are very large and so it requires an unreasonable amount of time to run, so I couldn't properly test these models.
4. **Using SVD to reduce the dimension**
   This failed because I didn't know the right parameters to use so using SVD only introduced noise to the model and lowers the accuracy

INSIGHTS

From this lab, I learned about a few things: first, is that it is very important to find a method that balances time and performance. It is unreasonable and unrealistic to wait 3 days for our model to run because there is no guarantee that the results will be worth the time it took to run the model. Second, since time is very important, we need to carefully choose our sampling method because a good sample will save us time to try out more things. Third, I found that datamining is not just about knowledge, sometimes luck plays a huge role in determining how our model will perform, just because two people use the same method doesn't mean that their accuracy will be the same, there are so many variables that comes to play like the parameter of our models, the random state that we used and so on. But just because we need luck doesn't mean that we can blame our problems solely on luck, because if we used the right method then our base performance can be better than the worse method with high luck.

Other than that, I found that trying different methods is way better than trying to fine tune our model, at least when we haven't tried other methods. I tried to fine tune my model but turns out that these fine tuning would only give minor improvements compared to when I finally try other models, so in the future, before I fine tune my model, I would first find what other models that I can try so I wont have to waste my precious time only to improve my performance by a small margin.