Sentiment Analysis for Yelp Reviews

Individual Report

Anuradha Tidke

Natural Language Processing
Department of Data Science
George Washington University

Table of Content

1. Introduction

The goal of this project is to estimate the Yelp ratings polarity (0 or 1) of a local business review based on the review text. This was done using NLP models with an emphasis on sentiment analysis. Yelp is a review-based website where people can discover and exchange information about local businesses. When consumers conduct a rapid search for businesses, they are more inclined to assess quality only on the star rating without reading the review language. As a result, our group is looking for probable connections between review wording and the corresponding rating.

In this project we will be exploring NLP algorithms like Recurrent Neural Networks (LSTM) and comparing the results with pre-trained models like DistilBERT and ELECTRA.

We will be using PyTorch packages such as Torch, Variable, get_tokenizer, nn, etc. to implement the model. PyTorch is an open source framework and is capable of using GPUs effectively and efficiently. It is very similar to Numpy hence it is easier to comprehend.

2. Dataset Description

We have taken our dataset from Kaggle. The Yelp reviews polarity dataset is constructed using the original dataset from the Yelp Dataset Challenge 2015. The ratings from this dataset are modified by considering stars 1 and 2 negative (label: 0), and 3 and 4 positive (label: 1). Thus this is a binary classification data. It contains over 560,000 training sample points and 38,000 testing sample points. The train and test sample points are distributed equally for both labels. The feature column is that of the review text and the target column is the label.

I worked solely on Data preprocessing and LSTM modeling.

3. Data Preprocessing

Data Download

As we are downloading our dataset from Kaggle, I had to install the kaggle API first followed by a few more steps. To install the API I ran the following command on the Anaconda command terminal: *pip install kaggle*

In order to use the Kaggle API, I downloaded my API token .json file from my Kaggle account and placed it in the location C:\Users<Windows-username>.kaggle\

Following the above step, I changed the directory in the Anaconda command terminal to my project directory by using command: *cd Users/[username]/Downloads/Group2Project (in my case)*

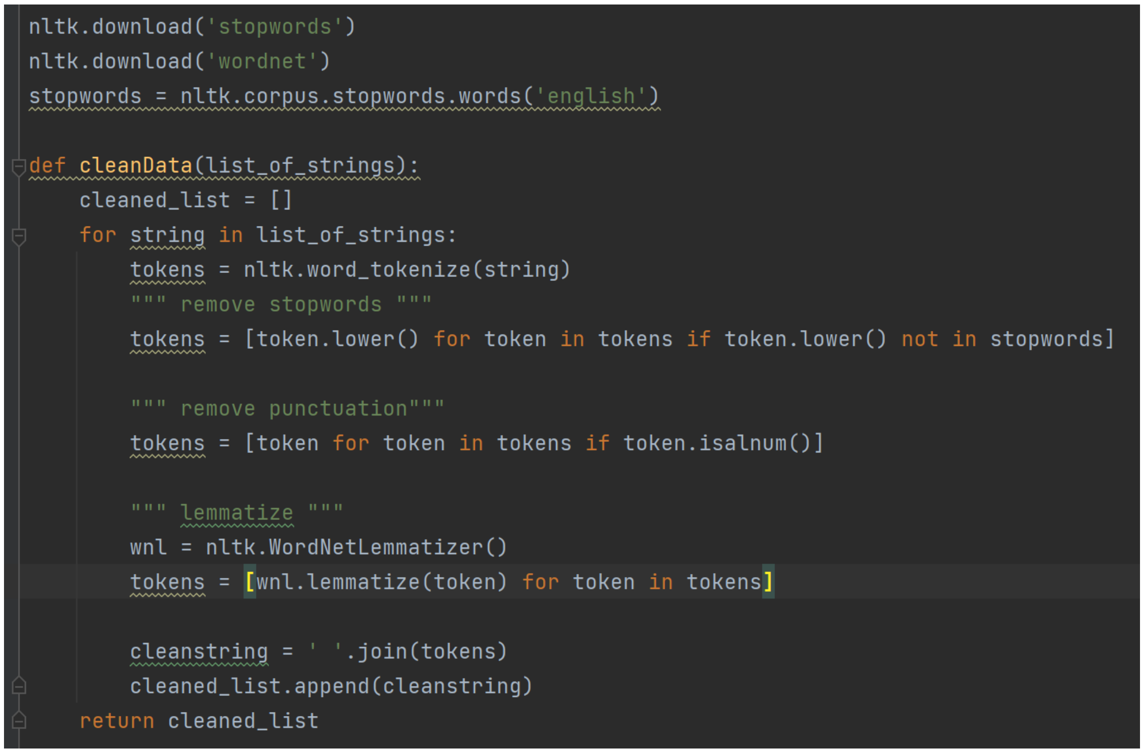
Once that was done, I downloaded our dataset .zip file from Kaggle using the following command: *kaggle datasets download ilhamfp31/yelp-review-dataset*

After downloading the .zip file, I ran the following code to unzip it and save it to a folder "yelp_review_polarity_csv":

```
import os
from zipfile import ZipFile
Path = os.getcwd()
file_name = Path + "\\yelp-review-dataset.zip"

with ZipFile(file_name, 'r') as zip:
    zip.printdir()
    print('Extracting all the files now...')
    zip.extractall()
    print('Done!')
```

Data Cleaning

Once the dataset was downloaded, I performed the data cleaning steps to remove the unwanted words and simplify the text. Firstly, I removed the stopwords, followed by punctuations and lemmatizing the data. This helped me reduce the number of unique tokens in the dataset, thus reducing the computational load.

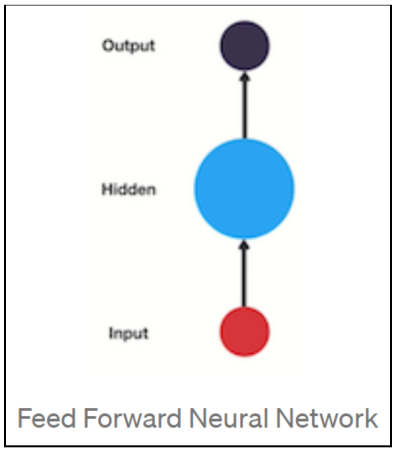


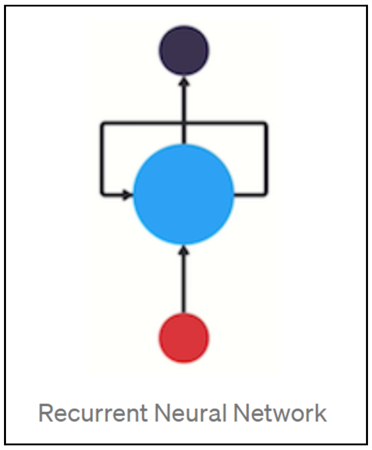
```python
nltk.download('stopwords')
nltk.download('wordnet')
stopwords = nltk.corpus.stopwords.words('english')

def cleanData(list_of_strings):
    cleaned_list = []
    for string in list_of_strings:
        tokens = nltk.word_tokenize(string)
        """ remove stopwords """
        tokens = [token.lower() for token in tokens if token.lower() not in stopwords]

        """ remove punctuation"""
        tokens = [token for token in tokens if token.isalnum()]

        """ lemmatize """
        wnl = nltk.WordNetLemmatizer()
        tokens = [wnl.lemmatize(token) for token in tokens]

        cleanstring = ' '.join(tokens)
        cleaned_list.append(cleanstring)
    return cleaned_list
```

Cleaning the data resulted in some sequences (reviews) reducing to zero length. Running the LSTM code with padding enabled was giving errors for zero length sequences. As the percentage of such sequences was minimal, I decided to drop them. The train dataset was split into train and validation dataset with a split ratio of 0.9. Once the cleaned data was saved in different csv files, I used it to run our LSTM models.

4. Architecture of Recurrent Neural Networks

Recurrent neural networks, also known as RNNs, are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. Let’s look at a traditional neural network also known as a feed-forward neural network. It has its input layer, hidden layer, and the output layer. We get a recurrent neural network if we add a loop in the neural network that can pass prior information forward. An RNN has a looping mechanism that acts as a connection to allow information to flow from one step to the next. This information is the hidden state, which is a representation of previous inputs.



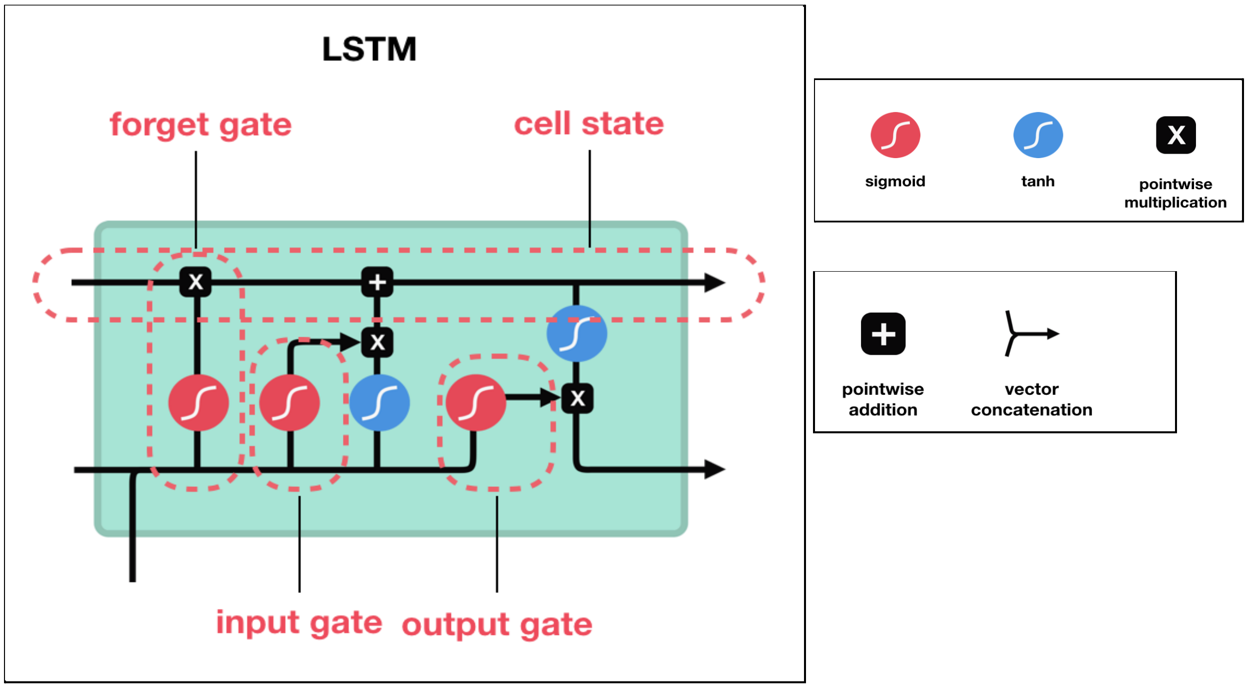


Training a neural network has three major steps. First, it does a forward pass and makes a prediction. Second, it compares the prediction to the ground truth using a loss function. The loss function outputs an error value which is an estimate of how poorly the network is performing. Last, it uses that error value to do back propagation which calculates the gradients for each node in the network.

As the RNN processes more steps, it has trouble retaining information from previous steps. This happens because of the vanishing gradient problem. Gradients are values used to update a neural network's weights. The bigger the gradient, the bigger the adjustments and vice versa. When doing back propagation, each node in a layer calculates its gradient with respect to the effects of the gradient in the layer before it. So if the adjustments to the layers before it are small, adjustments to the current layer will be even smaller. That causes gradients to exponentially shrink as it back-propagates. The earlier layers fail to do any learning as the internal weights are barely being adjusted due to extremely small gradients. And that’s the vanishing gradient problem.

LSTM ’s and GRU’s were created as the solution to short-term memory. They have internal mechanisms called gates that can regulate the flow of information.

5. Working of Long Short Term Memory (LSTM)



There are three gates which regulate the flow of information from past cells to the next. The forget gate decides what is relevant to keep from prior steps. The input gate decides what information is relevant to add from the current step. The output gate determines what the next hidden state should be.

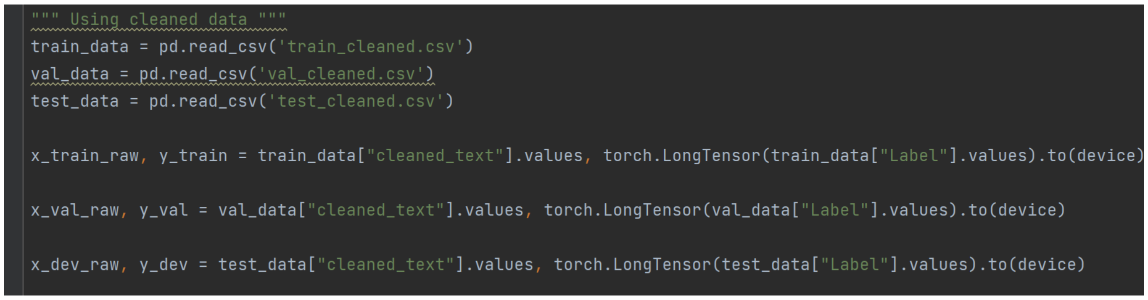
7. LSTM Model Code Structure

Step 1: Define LSTM class

Our first step was to define the LSTM class. The embedding and all the model layers were initialized. In order to pad the sequences, we created an array of lengths of all the sequences. Maximum sequence length was used as an argument to pad zeroes after all the sequences.

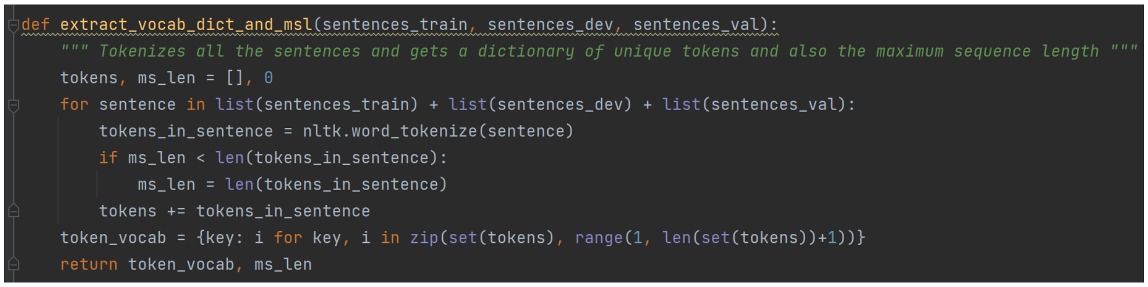
Step 2: Data preparation

The cleaned data was imported into pandas dataframes. The feature column from train, validation and test dataset were converted to lists, the label columns were converted to tensors and moved to the GPU.



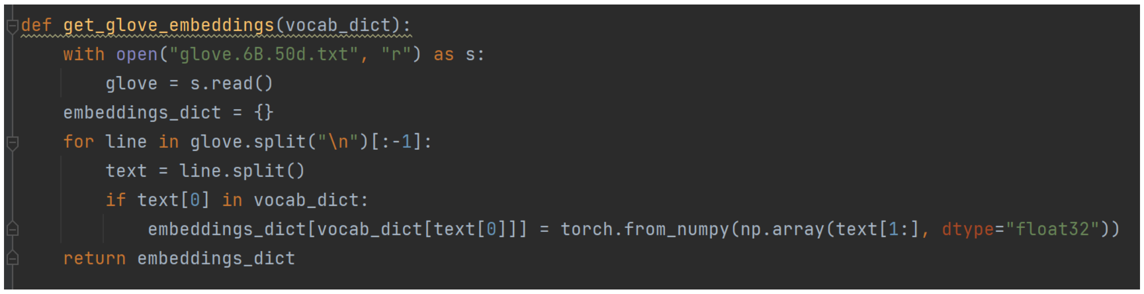
```
""" Using cleaned data """
train_data = pd.read_csv('train_cleaned.csv')
val_data = pd.read_csv('val_cleaned.csv')
test_data = pd.read_csv('test_cleaned.csv')

x_train_raw, y_train = train_data["cleaned_text"].values, torch.LongTensor(train_data["Label"].values).to(device)

x_val_raw, y_val = val_data["cleaned_text"].values, torch.LongTensor(val_data["Label"].values).to(device)

x_dev_raw, y_dev = test_data["cleaned_text"].values, torch.LongTensor(test_data["Label"].values).to(device)
```

The vocab dictionary including unique tokens from all the above datasets was created and maximum sequence length was calculated using the following function:



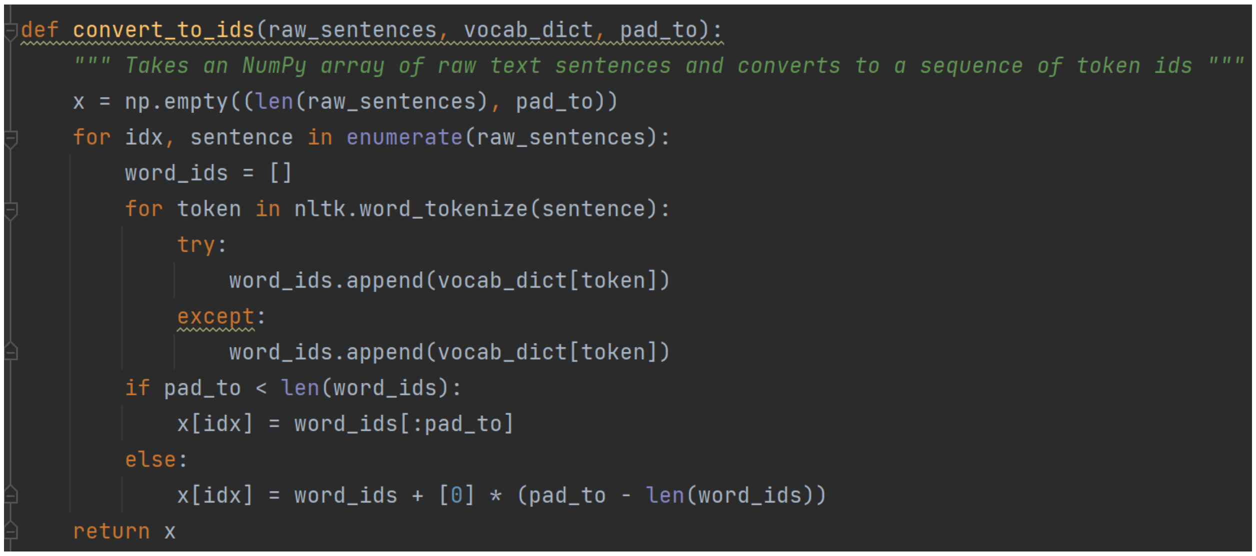
```python
def extract_vocab_dict_and_msl(sentences_train, sentences_dev, sentences_val):
    """ Tokenizes all the sentences and gets a dictionary of unique tokens and also the maximum sequence length """
    tokens, ms_len = [], 0
    for sentence in list(sentences_train) + list(sentences_dev) + list(sentences_val):
        tokens_in_sentence = nltk.word_tokenize(sentence)
        if ms_len < len(tokens_in_sentence):
            ms_len = len(tokens_in_sentence)
        tokens += tokens_in_sentence
    token_vocab = {key: i for key, i in zip(set(tokens), range(1, len(set(tokens))+1))}
    return token_vocab, ms_len
```

Embedding dictionary was created from the vocab dictionary using the glove embeddings:



```python
def get_glove_embeddings(vocab_dict):
    with open("glove.6B.50d.txt", "r") as s:
        glove = s.read()
    embeddings_dict = {}
    for line in glove.split("\n")[:-1]:
        text = line.split()
        if text[0] in vocab_dict:
            embeddings_dict[vocab_dict[text[0]]] = torch.from_numpy(np.array(text[1:], dtype="float32"))
    return embeddings_dict
```

Feature variable lists from the train, validation and test datasets were converted to a matrix of token ids using the following function. The number of columns of the matrix was taken as the maximum sequence length we obtained before.

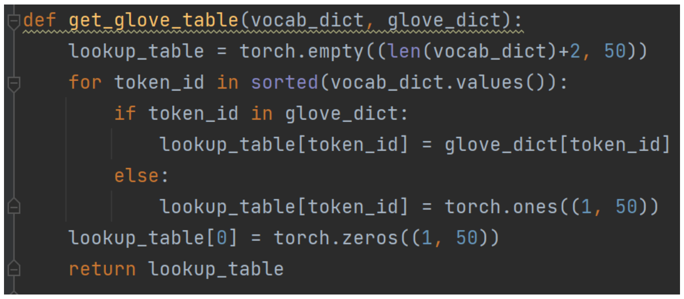


```python
def convert_to_ids(raw_sentences, vocab_dict, pad_to):
    """ Takes an NumPy array of raw text sentences and converts to a sequence of token ids """
    x = np.empty((len(raw_sentences), pad_to))
    for idx, sentence in enumerate(raw_sentences):
        word_ids = []
        for token in nltk.word_tokenize(sentence):
            try:
                word_ids.append(vocab_dict[token])
            except:
                word_ids.append(vocab_dict[token])
        if pad_to < len(word_ids):
            x[idx] = word_ids[:pad_to]
        else:
            x[idx] = word_ids + [0] * (pad_to - len(word_ids))
    return x
```

Following the above step, the features lists were converted to tensors and moved to the GPU.

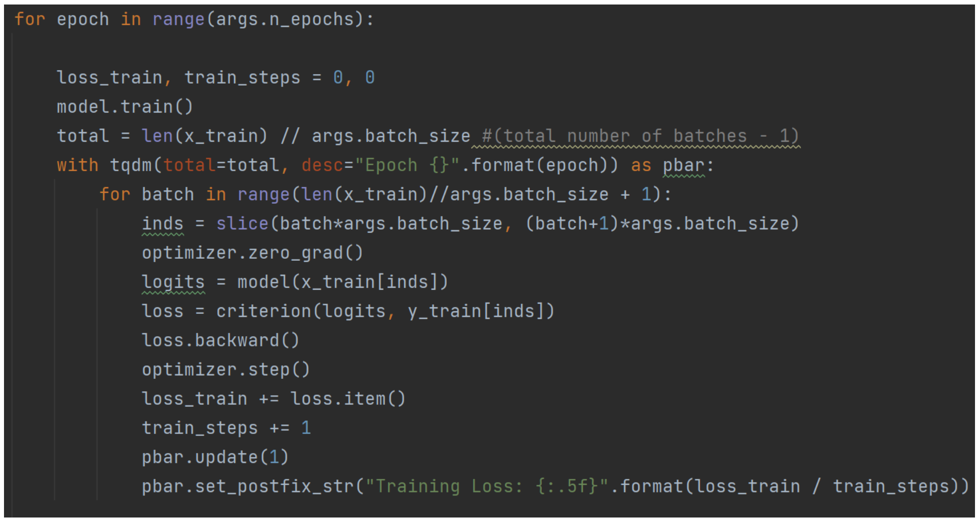
Step 3: Training the model

A lookup table was created using the vocab dictionary and the embedding dictionary. It contained the embedding vectors of all the tokens in the vocab dictionary arranged row wise. The table was fed to our LSTM model as the embedding weights. We used ADAM optimizer and cross entropy loss as the loss criteria for our model.



```python
def get_glove_table(vocab_dict, glove_dict):
    lookup_table = torch.empty((len(vocab_dict)+2, 50))
    for token_id in sorted(vocab_dict.values()):
        if token_id in glove_dict:
            lookup_table[token_id] = glove_dict[token_id]
        else:
            lookup_table[token_id] = torch.ones((1, 50))
    lookup_table[0] = torch.zeros((1, 50))
    return lookup_table
```

We ran 10 epochs to train this model. The learning rate was taken as 1e-3 and the batch size was 16. For each batch, the optimizer was initialized to zero to avoid gradient explosion. The predictions and loss were calculated for the data points of that batch. The loss was back propagated to calculate the gradient of previous batches. The ADAM optimizer used the gradients to update the weights. At the end of each epoch, the model was applied to the validation dataset and saved if the validation accuracy increased.



```python
for epoch in range(args.n_epochs):

    loss_train, train_steps = 0, 0
    model.train()
    total = len(x_train) // args.batch_size #(total number of batches - 1)
    with tqdm(total=total, desc="Epoch {}".format(epoch)) as pbar:
        for batch in range(len(x_train)//args.batch_size + 1):
            inds = slice(batch*args.batch_size, (batch+1)*args.batch_size)
            optimizer.zero_grad()
            logits = model(x_train[inds])
            loss = criterion(logits, y_train[inds])
            loss.backward()
            optimizer.step()
            loss_train += loss.item()
            train_steps += 1
            pbar.update(1)
            pbar.set_postfix_str("Training Loss: {:.5f}".format(loss_train / train_steps))
```

.

Step 4: LSTM results

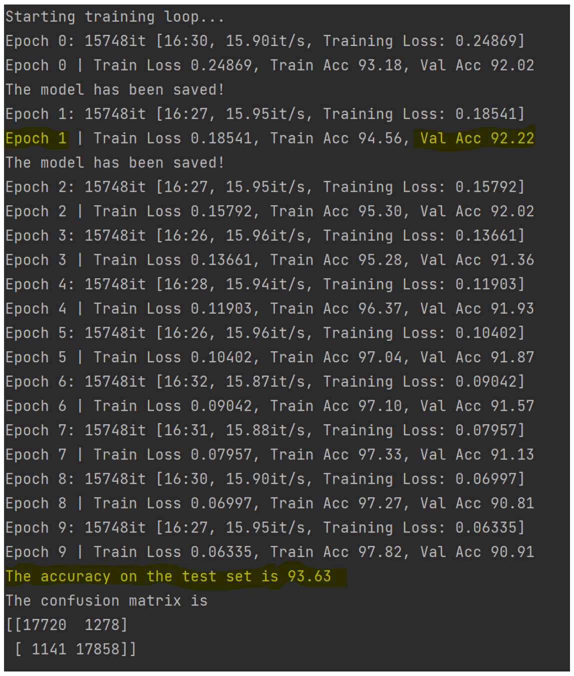
Version 1

Full train dataset was used to train the model

- train_data length: 503,924

Results:

- **Best val_acc: 92.22 %**
- Time req for all epochs = 2 hrs 45 min
- val_acc decreased after epoch 1 which indicates that the model was overfitting the train data
- **test_acc using the above best model: 93.63 %**



```
Starting training loop...
Epoch 0: 15748it [16:30, 15.90it/s, Training Loss: 0.24869]
Epoch 0 | Train Loss 0.24869, Train Acc 93.18, Val Acc 92.02
The model has been saved!
Epoch 1: 15748it [16:27, 15.95it/s, Training Loss: 0.18541]
Epoch 1 | Train Loss 0.18541, Train Acc 94.56, Val Acc 92.22
The model has been saved!
Epoch 2: 15748it [16:27, 15.95it/s, Training Loss: 0.15792]
Epoch 2 | Train Loss 0.15792, Train Acc 95.30, Val Acc 92.02
Epoch 3: 15748it [16:26, 15.96it/s, Training Loss: 0.13661]
Epoch 3 | Train Loss 0.13661, Train Acc 95.28, Val Acc 91.36
Epoch 4: 15748it [16:28, 15.94it/s, Training Loss: 0.11903]
Epoch 4 | Train Loss 0.11903, Train Acc 96.37, Val Acc 91.93
Epoch 5: 15748it [16:26, 15.96it/s, Training Loss: 0.10402]
Epoch 5 | Train Loss 0.10402, Train Acc 97.04, Val Acc 91.87
Epoch 6: 15748it [16:32, 15.87it/s, Training Loss: 0.09042]
Epoch 6 | Train Loss 0.09042, Train Acc 97.10, Val Acc 91.57
Epoch 7: 15748it [16:31, 15.88it/s, Training Loss: 0.07957]
Epoch 7 | Train Loss 0.07957, Train Acc 97.33, Val Acc 91.13
Epoch 8: 15748it [16:30, 15.90it/s, Training Loss: 0.06997]
Epoch 8 | Train Loss 0.06997, Train Acc 97.27, Val Acc 90.81
Epoch 9: 15748it [16:27, 15.95it/s, Training Loss: 0.06335]
Epoch 9 | Train Loss 0.06335, Train Acc 97.82, Val Acc 90.91
The accuracy on the test set is 93.63
The confusion matrix is
[[17720  1278]
 [ 1141 17858]]
```

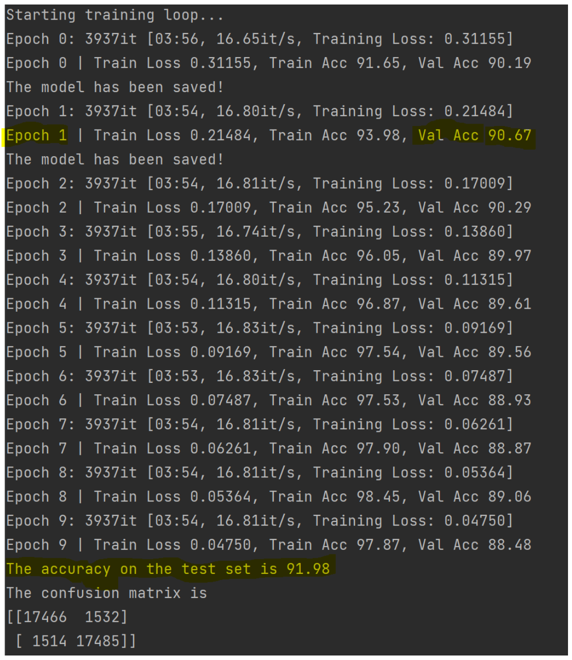
Version 2

The train dataset was shrunk to 25% of what it was before and used to train the model.

- train_data length: 125,982

Results:

- **Best val_acc: 90.67 %**
- Time req for all epochs > 40 min
- Solution is to reduce learning rate after epoch 1
- **test_acc using the above best model: 91.98%**



```
Starting training loop...
Epoch 0: 3937it [03:56, 16.65it/s, Training Loss: 0.31155]
Epoch 0 | Train Loss 0.31155, Train Acc 91.65, Val Acc 90.19
The model has been saved!
Epoch 1: 3937it [03:54, 16.80it/s, Training Loss: 0.21484]
Epoch 1 | Train Loss 0.21484, Train Acc 93.98, Val Acc 90.67
The model has been saved!
Epoch 2: 3937it [03:54, 16.81it/s, Training Loss: 0.17009]
Epoch 2 | Train Loss 0.17009, Train Acc 95.23, Val Acc 90.29
Epoch 3: 3937it [03:55, 16.74it/s, Training Loss: 0.13860]
Epoch 3 | Train Loss 0.13860, Train Acc 96.05, Val Acc 89.97
Epoch 4: 3937it [03:54, 16.80it/s, Training Loss: 0.11315]
Epoch 4 | Train Loss 0.11315, Train Acc 96.87, Val Acc 89.61
Epoch 5: 3937it [03:53, 16.83it/s, Training Loss: 0.09169]
Epoch 5 | Train Loss 0.09169, Train Acc 97.54, Val Acc 89.56
Epoch 6: 3937it [03:53, 16.83it/s, Training Loss: 0.07487]
Epoch 6 | Train Loss 0.07487, Train Acc 97.53, Val Acc 88.93
Epoch 7: 3937it [03:54, 16.81it/s, Training Loss: 0.06261]
Epoch 7 | Train Loss 0.06261, Train Acc 97.90, Val Acc 88.87
Epoch 8: 3937it [03:54, 16.81it/s, Training Loss: 0.05364]
Epoch 8 | Train Loss 0.05364, Train Acc 98.45, Val Acc 89.06
Epoch 9: 3937it [03:54, 16.81it/s, Training Loss: 0.04750]
Epoch 9 | Train Loss 0.04750, Train Acc 97.87, Val Acc 88.48
The accuracy on the test set is 91.98
The confusion matrix is
[[17466  1532]
 [ 1514 17485]]
```

8. Summary and Conclusion

I ran my LSTM, firstly using the full train dataset of around 500,000 points and using 25% of the dataset which is around 125,000 data points after that. Fortunately, shrinking the training data didn't affect my test accuracy by a lot (decreased by 1.7%). So we decided to use this size for the train data for our pretrained models.

I concluded that it didn't affect the accuracy due to the following possible reasons:

1. The original train dataset was more than enough to train the model
2. Data preprocessing steps helped the model learn better by removing unnecessary words and lemmatizing the text.

9. Percentage of copied code

My code files: Total 281 lines

Copied from internet: 205

Modified: 19

Added by myself: 76

Copy percentage = 66%

10. References

a. https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

b. https://github.com/amir-jafari/Deep-Learning/blob/master/Pytorch/RNN/2_TextClassification/example_LSTM_sentiment_analysis.py