HW 2: Data Preparation for Sentiment Classification

In this homework we will prepare the IMDB movie review sentiment dataset. We will prepare it to fit a model that will predict whether a new review has a positive or negative sentiment.

Start by downloading the IMDB_Dataset from the .csv file into a pandas DataFrame

In [1]:

```
import pandas as pd
import numpy as np

#Download the dataset into a Pandas DataFrame and display the first 5 rows
## YOUR CODE HERE
imdb = pd.read_csv("IMDB_Dataset.csv")
imdb
```

Out[1]:

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. <br /><br />The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...	positive
...	...	...
49995	I thought this movie did a down right good job...	positive
49996	Bad plot, bad dialogue, bad acting, idiotic di...	negative
49997	I am a Catholic taught in parochial elementary...	negative
49998	I'm going to have to disagree with the previou...	negative
49999	No one expects the Star Trek movies to be high...	negative

50000 rows × 2 columns

We have to process the data first, so that we have fixed length sequences.

We want to split the dataset into reviews and labels.

In [2]:

```
imdb_x = imdb['review']
imdb_y = imdb['sentiment']
```

In [3]:

```
# Paste the toBinary function created in HW 1 from this hw set (week 2)
def toBinary(data, positive):
    data = (data == positive).astype(int)
    return data
```

Use the toBinary method to tranform the sentiment column into binary, 1 for positive and 0 for negative.

In [4]:

```
imdb_y = toBinary(imdb_y, 'positive')
imdb.head()
```

Out[4]:

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. <br /><br />The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...	positive

In [5]:

```
imdb_x.head()
```

Out[5]:

```
0    One of the other reviewers has mentioned that ...
1    A wonderful little production. <br /><br />The...
2    I thought this was a wonderful way to spend ti...
3    Basically there's a family where a little boy ...
4    Petter Mattei's "Love in the Time of Money" is...
Name: review, dtype: object
```

"Lemmatization is the process of converting a word to its base form. The difference between stemming and lemmatization is, lemmatization considers the context and converts the word to its meaningful base form, whereas stemming just removes the last few characters, often leading to incorrect meanings and spelling errors." https://www.machinelearningplus.com/nlp/lemmatization-examples-python/

Lemmatize the sentences using any library from the article. Make sure to filter out non-alphabetical characters.

In [6]:

```
import nltk
from nltk.stem import WordNetLemmatizer
# nltk.download('wordnet')
def Lemmatize(data):
    # Init the Wordnet Lemmatizer
    data = data.str.split(" |,|<br /><br />|\"|\'|!") # splits the data from sentences to
    lemmatizer = WordNetLemmatizer()
    def helper(sentence):
        out = []
        for word in sentence:
            out.append(lemmatizer.lemmatize(word))
        return out
    data.apply(helper)

    return data

imdb_x = Lemmatize(imdb_x)
```

The data has to be put into integer form, so each integer represents a unique word, 0 represents a PAD character, 1 represents a START character and 2 represents a character that is unknown because it is not in the top `num_words`. Thus 3 represents the first real word.

Also the words should be in decreasing order of frequency, so the word that 3 represents is the most common word in the dataset.

Complete CreateDict which will take in a data column
1) Create a Dict that maps {Word: Apperances in dataset} *Do not implement dictionary keys for PAD, START, and unknown characters (except "" see hint), this will be done later.*
2) Choose the top N most recurring words and give ascending indexes starting at 3

In [7]:

```
import re
numWords = 1000
def CreateDict(data, topN):
    regex = re.compile('[@_!#$%^&*()<>?/\|}{~:]')
    out = {}
    for sentence in data.to_numpy():
        for word in sentence:
            if regex.search(word) == None:
                if word in out:
                    out[word] += 1
                else:
                    out[word] = 1
    ordered = sorted(out, key=out.get)
    sorted_dict = {}
    for w in ordered:
        sorted_dict[w] = out[w]
    output = {}
    for i, word in enumerate(sorted_dict.keys()):
        if i < topN:
            output[word] = i
        else:
            break
    return output

    ##Hint: "" (or the empty string) is actually
    ### the most common word but we want to cast it as an unknown word, what index should

#     pass

wordCounter = CreateDict(imdb_x, numWords)
```

Complete replaceByIndex which will replace known words with their index and unknown words with a 2.

In [ ]:
```
def replaceByIndex(data, wordCounter):
    for i, sentence in enumerate(data.to_numpy()):
        before = data[i]
        after = []
        for word in before:
            if word in wordCounter:
                after.append(wordCounter[word])
            else:
                after.append(2)
        data[i] = after
    return data

imdb_x = replaceByIndex(imdb_x, wordCounter)
```

In [ ]:
```
imdb_x.head()
```

We want to process the data into NumPy arrays of sequences that are all length 200. We will use these criteria:

- We want to add a 1 at the beginning of every review to signal the beginning of the text.
- If a given sequence is shorter than 200 tokens we want to pad the beginning of the sequence out with zeros so that the sequence is 200 long.
- Else if the sequence is longer than 200 (including the starting 1) we want to cut it down to length 200.

In [ ]:
```
def process_data(data):

    for sentence in data.to_numpy():
        np.insert(sentence, 0, 1)
        if sentence.size > 200:
            sentence = sentence[:200]
        elif sentence.size < 200:
            sentence = np.pad(sentence, (0, 200 - sentence.size), 'constant')
    return data

imdb_x = process_data(imdb_x)
```

Separate the dataset into train and test sets, test set should be 1/3 of the set.

This sklearn method will make your life much easier: train_test_split

In [ ]:

```
## YOUR CODE HERE

x_train_proc, x_test_proc, y_train, y_test = train_test_split(imdb_x, imdb_y, test_size=0.
```

At this point **your job is done!!!** Congratulations, if done correctly, the sentences are processed and ready to be used as features and labels to train a Recurrent Neural Network (LSTM). You will learn how to do this yourself in the next couple weeks. For now, you can just sit back and "follow along" as we build this model using Keras and then train it.

The first thing we will do is initialize the model using Sequential.

In [ ]:

```
import keras
from keras import Sequential

imdb_model = Sequential()
```

Now we want to add an embedding layer. The purpose of an embedding layer is to take a sequence of integers representing words in our case and turn each integer into a dense vector in some embedding space. (This is essentially the idea of Word2Vec https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf). We want to create an embedding layer with vocab size equal to the max num words we allowed when we loaded the data (in this case 1000), and a fixed dense vector of size 32. Then we have to specify the max length of our sequences and we want to mask out zeros in our sequence since we used zero to pad. Use the docs for embedding layer to fill out the missing entries: https://keras.io/layers/embeddings/

In [ ]:

```
from keras.layers.embeddings import Embedding
imdb_model.add(Embedding(1000, 32, input_length=200, mask_zero=True))
```

(a) We add an LSTM layer with 32 outputs, then a Dense layer with 16 neurons, then a relu activation, then a dense layer with 1 neuron, then a sigmoid activation. Then we print out the model summary. The Keras documentation is here: https://keras.io/

In [ ]:

```
from keras.layers.recurrent import LSTM
from keras.layers import Dense, Activation
imdb_model.add(LSTM(32))
```

In [ ]:

```
imdb_model.add(Dense(units=16, activation='relu'))
imdb_model.add(Dense(units=1, activation='sigmoid'))
```

In [ ]:

```
imdb_model.summary()
```

(b) Now we compile the model with binary cross entropy, and the adam optimizer. We include accuracy as a metric in the compile. Then train the model on the processed data.

```
In [ ]:
imdb_model.compile(loss=keras.losses.binary_crossentropy, optimizer=keras.optimizers.Adam(
```

```
In [ ]:
imdb_model.fit(x_train_proc, y_train)
```

```
In [ ]:
print("Accuracy: ", imdb_model.evaluate(x_test_proc, y_test)[1])
```

If you did the data pre-processing correctly you should be getting around an 80% accuracy. congratulations, that is much better than random!

If you are getting a test accuracy that is significantly lower, you probably did something wrong, slack your NMEP team or go to office hours to get help sorting it out :)

Now we can look at our predictions and the sentences they correspond to.

```
In [ ]:
y_pred = imdb_model.predict(x_test_proc)
```

```
In [ ]:
word_to_id = wordCounter
word_to_id["<PAD>"] = 0
word_to_id["<START>"] = 1
word_to_id["<UNK>"] = 2

id_to_word = {value:key for key,value in word_to_id.items() if value < 2000}
def get_words(token_sequence):
    return ' '.join(id_to_word[token] for token in token_sequence)

def get_sentiment(y_pred, index):
    return 'Positive' if y_pred[index] else 'Negative'
```

```
In [ ]:
y_test = [i for i in y_test]
y_pred = np.vectorize(lambda x: int(x >= 0.5))(y_pred)
correct = []
incorrect = []
for i, pred in enumerate(y_pred):
    if y_test[i] == pred:
        correct.append(i)
    else:
        incorrect.append(i)
```

Now we print out one of the sequences we got correct.

```
In [ ]:
print(get_sentiment(y_pred, correct[10]))
print(get_words(x_test_proc[correct[10]]))
```

And one we got wrong.

```
In [ ]:
print(get_sentiment(y_pred, incorrect[10]))
print(get_words(x_test_proc[incorrect[10]]))
```

As you can see the amount of UNKNOWN characters in the sequence cause by having only 1000 vocab words is hurting our performance. If you want, go back and increase the number of vocab words to 2000 and compare your accuracy

(If you do so, remember to change your embedding parameter from 1000 to 2000 as well; you should get ~85% accuracy).

And that's it! Now you should feel like a data engineering/preprocessing expert :)