Dataset:

https://data.world/crowdflower/sentiment-analysis-in-text

In [ ]:

```
import pandas as pd
# Reading the csv dataset into a pandas dataframe
df = pd.read_csv('E:/Internships/TCS-iON/Code/MyCode/Tweets/text_emotion.csv', encoding = '
# Adding a column representing 1 for positive and 0 for negative sentiments
df['senti'] = df.apply(lambda x: 1 if (x['sentiment'] == 'enthusiasm' or x['sentiment'] ==
# Deleting unnecessary columns
df = df.drop(['tweet_id', 'sentiment', 'author'], axis = 1)
# Converting the data type to string
df['content'] = df["content"].astype("str")
# Converting all text to lowercase for use
df['content'] = df['content'].str.lower()
df.head()
```

	review	senti
	@tiffanylue i know i was listenin to bad habi...	0
	layin n bed with a headache ughhhh...waitin o...	0
	funeral ceremony...gloomy friday...	0
	wants to hang out with friends soon!	1
	@dannycastillo we want to trade with someone w...	0

i am going to start reading the harry potter series again because that is one awesome story.

In [ ]:

```
import re
import string
from nltk import WordNetLemmatizer
from nltk.stem.snowball import SnowballStemmer
from nltk.corpus import stopwords
# Initialising the nltk stop_words, stemmer and lemmatizer functions
stop_words = set(stopwords.words("english"))
lemmatizer = WordNetLemmatizer()
stemmer = SnowballStemmer("english")
# Creating a function for text cleaning
def textCleanser(myText):
    # Converting each tweet to string
    myText = str(myText)
    # Removing the name titles and the period symbols after it
    myText = re.sub(r'[mdsr]r(s)?\.', '', myText)
    # Removing the '@username' mentions
    myText = re.sub(r'@\w+\s', '', myText)
    myText = re.sub(r'@\w+', '', myText)
    # Removing hashtags
    myText = re.sub(r'#', '', myText)
    # Removing punctuation
    myPunct = string.punctuation
    punctToSpace = str.maketrans(myPunct, len(myPunct)*' ')
    myText = myText.translate(punctToSpace)
    # Removing urls
    myText = re.sub(r'((http(s?)?):://?)(www\.?).+\.com', '', myText)
    myText = re.sub(r'http(s?)', '', myText)
    # Removing numbers
    myText = re.sub(r'\d+', '', myText)
    # Removing stopwords
    myText = [word for word in myText.split(' ') if not word in stop_words]
    myText = [word for word in myText if word != '']
    # Lemmatizing the text
    myText = [lemmatizer.lemmatize(token) for token in myText]
    # Stemming the text
    # myText = [stemmer.stem(token) for token in myText]
    return myText
for i in range(len(df['content'])):
    df['content'][i] = textCleanser(df['content'][i])
df.head()
```

	review	senti
	[know, listenin, bad, habit, earlier, started,...	0
	[layin, n, bed, headache, ughhhh, waitin, call]	0
	[funeral, ceremony, gloomy, friday]	0
	[want, hang, friend, soon]	1
	[want, trade, someone, houston, ticket, one]	0

['going', 'start', 'reading', 'harry', 'potter', 'series', 'one', 'awesome', 'story']

In [ ]:

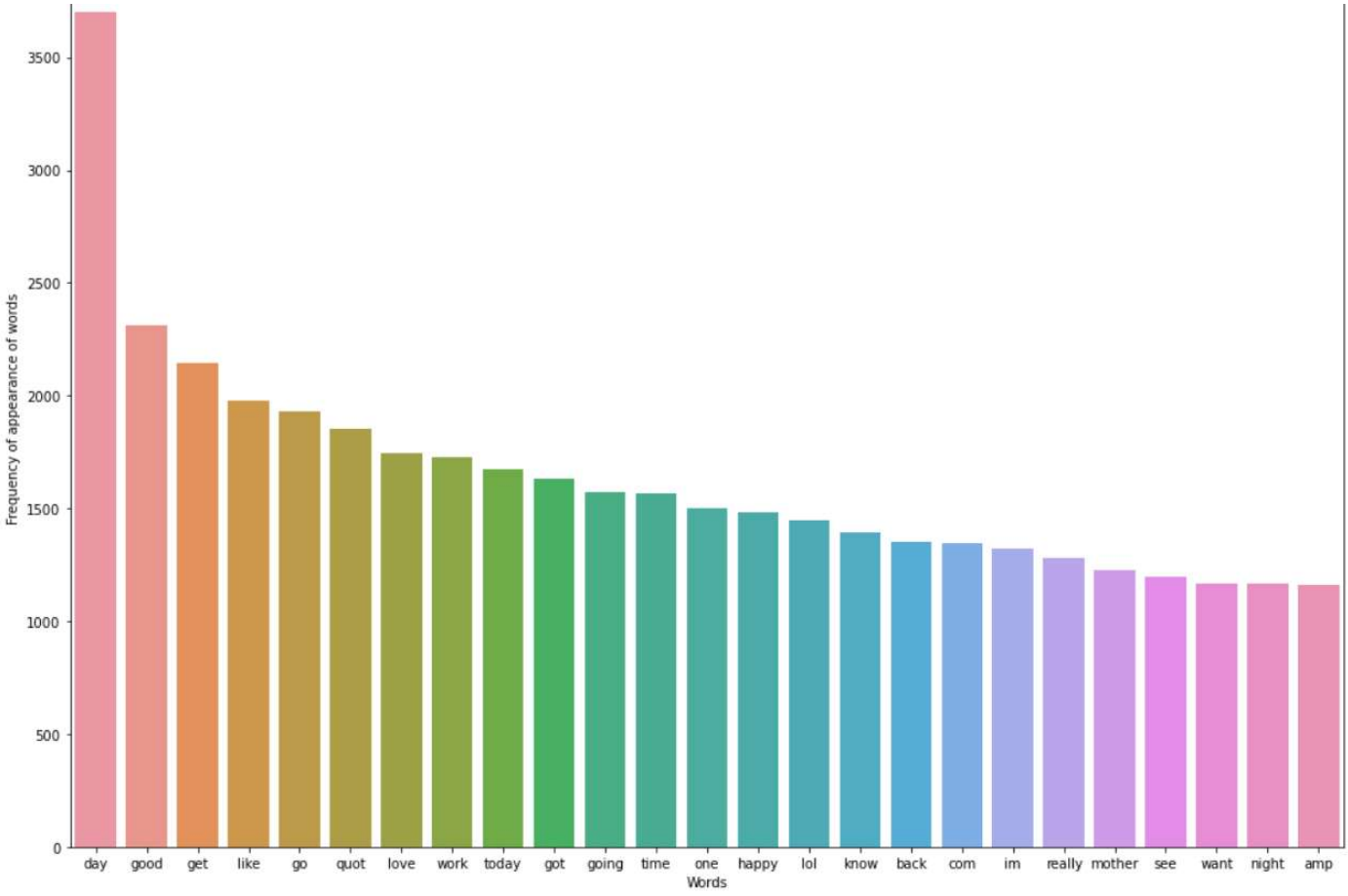
```
myReviews = []
for i in range(len(df['content'])):
    for j in df['content'][i]:
        if j != 'br' and j != 'http':
            myReviews.append(j)
```

In [ ]:

```
from collections import Counter
import collections
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.metrics import classification_report
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
import numpy as np
np.random.seed(1234)
# Initialising the Count Vectorizer
cv = CountVectorizer()
myBow = cv.fit_transform(myReviews)
wordFrequency = dict(zip(cv.get_feature_names(), np.asarray(myBow.sum(axis = 0)).ravel()))
wordCounter = collections.Counter(wordFrequency)
# Storing the frequency of appearance of words
dfWordCounter = pd.DataFrame(wordCounter.most_common(25), columns = ['word', 'frequency'])
```

In [ ]:

```
# Plotting the top 25 most frequently occuring words
plt.close('all')
fig, ax = plt.subplots(figsize = (17, 12))
sns.barplot(x = 'word', y = 'frequency', data = dfWordCounter, ax = ax)
sns.set_palette('pastel')
plt.xlabel('Words')
plt.ylabel('Frequency of appearance of words')
plt.show()
```


Frequency of appearance of words
3500
3000
2500
2000
1500
1000
500
0
day
good
get
like
go
quot
love
work
today
got
going
time
one
happy
lol
know
back
com
im
really
mother
see
want
night
amp
Words

In [ ]:

```
from sklearn.model_selection import train_test_split
import random
import torch
import torchtext
from torchtext import data
import spacy
import en_core_web_sm
import torch.nn as nn
import torch.nn.functional as fn
SEED = 1234
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
# Creating Field objects and including the lengths of each batch text field
TEXT = data.Field(tokenize = 'spacy', include_lengths = True)
LABEL = data.LabelField(dtype = torch.float)
# Creating training and testing datasets
training, testing = train_test_split(df, test_size=0.2, random_state=42)
training, validating = train_test_split(training, test_size=0.2, random_state=42)
training.to_csv('training.csv')
validating.to_csv('validating.csv')
testing.to_csv('testing.csv')
fields = [(None, None), ('c', TEXT), ('s', LABEL)]
train_data, valid_data, test_data = data.TabularDataset.splits(
    path = 'E:/Internships/TCS-iON/Code/MyCode/Tweets/',
    train = 'training.csv',
    validation = 'validating.csv',
    test = 'testing.csv',
    format = 'csv',
    fields = fields,
    skip_header = True
)
MAX_VOCAB_SIZE = 50000
# Building the vocabularies
TEXT.build_vocab(
    train_data,
    max_size = MAX_VOCAB_SIZE,
    # Using the glove.6B.100d vector
    vectors = "glove.6B.100d",
    unk_init = torch.Tensor.normal_)
LABEL.build_vocab(train_data)
BATCH_SIZE = 64
device = torch.device('cpu')
# Creating vocabulary iterators
train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    sort = False,
    batch_size = BATCH_SIZE,
    device = device)
# Building the training model
class myModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout, pad_idx):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx = pad_idx)
        self.rnn = nn.LSTM(embedding_dim,
                           hidden_dim,
                           num_layers=n_layers,
                           bidirectional=bidirectional,
                           dropout=dropout)
```

```
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.dropout = nn.Dropout(dropout)
    def forward(self, text, text_lengths):
        embedded = self.dropout(self.embedding(text))
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths, enforce
        packed_output, (hidden, cell) = self.rnn(packed_embedded)
        output, output_lengths = nn.utils.rnn.pad_packed_sequence(packed_output)
        hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))
        return self.fc(hidden)
inputDimension = len(TEXT.vocab)
embeddingDimension = 100
outputDimension = 1
hiddenDimension = 256
padding = TEXT.vocab.stoi[TEXT.pad_token]
layersCount = 2
bidir = True
dropout = 0.5
model = myModel(
    inputDimension,
    embeddingDimension,
    hiddenDimension,
    outputDimension,
    layersCount,
    bidir,
    dropout,
    padding)
```

In [ ]:

```
# Copying the pre-trained vectors to the embedding layer
pretrained_embeddings = TEXT.vocab.vectors
model.embedding.weight.data.copy_(pretrained_embeddings)
# Setting the unknown and padding elements to zero since they are of no use
unknown = TEXT.vocab.stoi[TEXT.unk_token]
model.embedding.weight.data[unknown] = torch.zeros(embeddingDimension)
model.embedding.weight.data[padding] = torch.zeros(embeddingDimension)
# Counting the number of trainable parameters
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 4,472,157 trainable parameters
```

In [ ]:

```
import time
import torch.optim as optim
optimizer = optim.Adam(model.parameters())
criterion = nn.BCEWithLogitsLoss()
# Defining function to calculate accuracy
def binary_accuracy(preds, y):
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float()
    acc = correct.sum() / len(correct)
    return acc
# Defining function for training
def train(model, iterator, optimizer, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.train()
    for batch in iterator:
        optimizer.zero_grad()
        text, text_lengths = batch.c
        predictions = model(text, text_lengths).squeeze(1)
        loss = criterion(predictions, batch.s)
        acc = binary_accuracy(predictions, batch.s)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        epoch_acc += acc.item()
    return epoch_loss / len(iterator), epoch_acc / len(iterator)
# Defining function for testing
def evaluate(model, iterator, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.eval()
    with torch.no_grad():
        for batch in iterator:
            text, text_lengths = batch.c
            predictions = model(text, text_lengths).squeeze(1)
            loss = criterion(predictions, batch.s)
            acc = binary_accuracy(predictions, batch.s)
            epoch_loss += loss.item()
            epoch_acc += acc.item()
    return epoch_loss / len(iterator), epoch_acc / len(iterator)
# Defining function to calculate time
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

In [ ]:

```
# Training the model
print(time.time())
N_EPOCHS = 5
best_valid_loss = float('inf')
for epoch in range(N_EPOCHS):
    start_time = time.time()
    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)
    end_time = time.time()
    epoch_mins, epoch_secs = epoch_time(start_time, end_time)
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'myModelRNN.pt')
    print(f'Epoch: {epoch+1:02}')
    print(f'\tEpoch Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
print(time.time())
```

Epoch: 01

```
    Epoch Time: 57m 17s
    Train Loss: 0.630 | Train Acc: 65.61%
     Val. Loss: 0.573 |  Val. Acc: 70.73%
```

Epoch: 02

```
    Epoch Time: 70m 58s
    Train Loss: 0.559 | Train Acc: 71.87%
     Val. Loss: 0.559 |  Val. Acc: 71.84%
```

Epoch: 03

```
    Epoch Time: 58m 22s
    Train Loss: 0.532 | Train Acc: 74.14%
     Val. Loss: 0.548 |  Val. Acc: 73.48%
```

Epoch: 04

```
    Epoch Time: 55m 11s
    Train Loss: 0.509 | Train Acc: 75.92%
     Val. Loss: 0.555 |  Val. Acc: 72.83%
```

Epoch: 05

```
    Epoch Time: 57m 51s
    Train Loss: 0.483 | Train Acc: 77.10%
     Val. Loss: 0.562 |  Val. Acc: 72.48%
```

In [ ]:

```
# Testing the model
model.load_state_dict(torch.load('myModelRNN.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion)
# Calculating test accuracy
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.542
Test Acc: 73.14%
```

In [ ]:

```

```