Datasets:

```
https://www.kaggle.com/utathya/imdb-review-dataset
http://ai.stanford.edu/~amaas/data/sentiment/ (better)
```

In [ ]:

```
import pandas as pd
# Reading the csv dataset into a pandas dataframe
df = pd.read_csv('E:/Internships/TCS-iON/Code/MyCode/IMDB/imdb_master.csv', encoding = 'ISO
# Adding a column representing 1 for 'pos' and 0 for 'neg' sentiments
df['senti'] = df.apply(lambda x: 1 if x['label'] == 'pos' else 0, axis = 1)
# Deleting unnecessary columns
df = df.drop(['Unnamed: 0', 'type', 'file'], axis = 1)
# Converting the data type to string
df['review'] = df["review"].astype("str")
# Converting all text to lowercase for use
df['review'] = df['review'].str.lower()
df.head()
```

review	label	senti
once again mr. costner has dragged out a movie...	neg	0
this is an example of why the majority of acti...	neg	0
first of all i hate those moronic rappers, who...	neg	0
not even the beatles could write songs everyon...	neg	0
brass pictures (movies is not a fitting word f...	neg	0

once again mr. costner has dragged out a movie for far longer than necessary. aside from the terrific sea rescue sequences, of which there are very few i just did not care about any of the characters. most of us have ghosts in the closet, and costner's character are realized early on, and then forgotten until much later, by which time i did not care. the character we should really care about is a very cocky, overconfident ashton kutcher. the problem is he comes off as kid who thinks he's better than anyone else around him and shows no signs of a cluttered closet. his only obstacle appears to be winning over costner. finally when we are well past the half way point of this stinker, costner tells us all about kutcher's ghosts. we are told why kutcher is driven to be the best with no prior inkling or foreshadowing. no magic here, it was all i could do to keep from turning it off an hour in.

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):
    # Removing the name titles and the period symbols after it
    myText = re.sub(r'[mdsr]r(s)?\.', '', myText)
    # Removing punctuation
    myPunct = string.punctuation
    punctToSpace = str.maketrans(myPunct, len(myPunct)*' ')
    myText = myText.translate(punctToSpace)
    # Removing the '@username' mentions
    myText = re.sub(r'@\w+', '', myText)
    # Removing urls
    myText = re.sub(r'(http(s)?://)?(www\.)?.+\.com', '', 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['review'])):
    df['review'][i] = textCleanser(df['review'][i])
df.head()
```

review	label	senti
[costner, dragged, movie, far, longer, necessa...	neg	0
[example, majority, action, film, generic, bor...	neg	0
[first, hate, moronic, rapper, could, nt, act,...	neg	0
[even, beatles, could, write, song, everyone, ...	neg	0
[brass, picture, movie, fitting, word, really,...	neg	0

```
['costner', 'dragged', 'movie', 'far', 'longer', 'necessary', 'aside', 'terrific', 'sea', 'rescue', 'sequence', 'care', 'character', 'u', 'ghost', 'closet', 'costner', 'character', 'realized', 'early', 'forgotten', 'much', 'later', 'time', 'care', 'character', 'really', 'care', 'cocky', 'overconfident', 'ashton', 'kutcher', 'problem', 'come', 'kid', 'think', 'better', 'anyone', 'else', 'around', 'show', 'sign', 'cluttered', 'closet', 'obstacle', 'appears', 'winning', 'costner', 'finally', 'well', 'past', 'half', 'way', 'point', 'stinker', 'costner', 'tell', 'u', 'kutcher', 'ghost', 'told', 'kutcher', 'driven', 'best', 'prior', 'inkling', 'foreshadowing', 'magic', 'could', 'keep', 'turning', 'hour']
```

In [ ]:

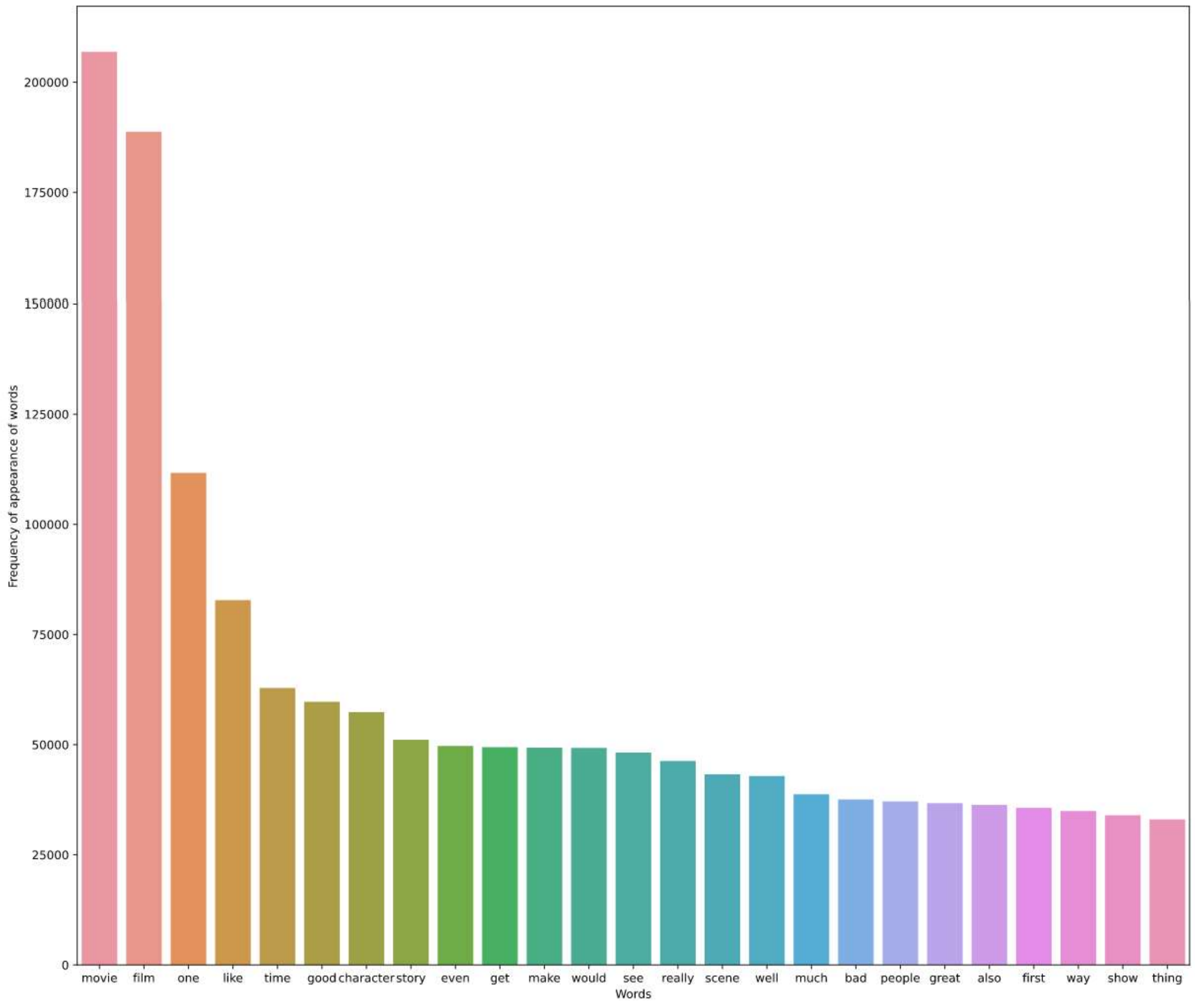
```
myReviews = []
for i in range(len(df['review'])):
    for j in df['review'][i]:
        if j != 'br':
            myReviews.append(j)
```

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, 15))
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
Words
0
25000
50000
75000
100000
125000
150000
175000
200000
movie
film
one
like
time
good
character
story
even
get
make
would
see
really
scene
well
much
bad
people
great
also
first
way
show
thing

In [ ]:

```
df.head()
```

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 functions to generate bigrams
def myBigrams(x):
    n_grams = set(zip(*[x[i:] for i in range(2)]))
    for n_gram in n_grams:
        x.append(' '.join(n_gram))
    return x
def dummyfn(doc):
    return doc
# Generating bigrams
TEXT = data.Field(tokenize = 'spacy', preprocessing = myBigrams)
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), (None, None), ('s', LABEL)]
train_data, valid_data, test_data = data.TabularDataset.splits(
    path = 'E:/Internships/TCS-iON/Code/MyCode/IMDB/',
    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, output_dim, pad_idx):
        super().__init__()
        # Initialising the embedding layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=pad_idx)
        # Initialising the linear layer
        self.fc = nn.Linear(embedding_dim, output_dim)
    def forward(self, text):
        embedded = self.embedding(text)
        embedded = embedded.permute(1, 0, 2)
        # Averaging the sentences in 2d
        pooled = fn.avg_pool2d(embedded, (embedded.shape[1], 1)).squeeze(1)
        return self.fc(pooled)
inputDimension = len(TEXT.vocab)
embeddingDimension = 100
outputDimension = 1
padding = TEXT.vocab.stoi[TEXT.pad_token]
model = myModel(inputDimension, embeddingDimension, outputDimension, 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 5,000,301 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()
        predictions = model(batch.c).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:
            predictions = model(batch.c).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
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(), 'myModelBigram.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}%')
```

Epoch: 01

```
    Epoch Time: 13m 26s
    Train Loss: 0.523 | Train Acc: 70.48%
     Val. Loss: 0.502 |  Val. Acc: 71.19%
```

Epoch: 02

```
    Epoch Time: 12m 46s
    Train Loss: 0.492 | Train Acc: 71.14%
     Val. Loss: 0.488 |  Val. Acc: 72.08%
```

Epoch: 03

```
    Epoch Time: 12m 39s
    Train Loss: 0.467 | Train Acc: 74.98%
     Val. Loss: 0.454 |  Val. Acc: 73.42%
```

Epoch: 04

```
    Epoch Time: 13m 45s
    Train Loss: 0.443 | Train Acc: 76.57%
     Val. Loss: 0.429 |  Val. Acc: 73.89%
```

Epoch: 05

```
    Epoch Time: 13m 57s
    Train Loss: 0.431 | Train Acc: 77.14%
     Val. Loss: 0.418 |  Val. Acc: 74.12%
```

In [ ]:

```
# Testing the model
model.load_state_dict(torch.load('myModelBigram.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: 0.423
Test Acc: 74.06%
```