Word Embedding

All texts need to be converted to numbers before starts processing by the machine. Specifically, vectors of numbers.

Text is messy in nature and machine learning algorithms prefer well defined fixed-length inputs and outputs.

Word Embedding is one such technique where we can represent the text using vectors. Before deep learning era, the popular forms of word embeddings were:

- **BoW**, which stands for Bag of Words
- **TF-IDF**, which stands for Term Frequency-Inverse Document Frequency

Bag-of-Words (BoW)

The **Bag-of-Words (BoW)** model is a way of representing text data when modeling text with machine learning algorithms. The **Bag-of-Words (BoW)** model is popular, simple to understand, and has seen great success in **language modeling** and **document classification**.

A bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:

- A vocabulary of known words.
- A measure of the presence of known words.

Example (BoW)

Consider the following 4 sentences:-

- It was the best of times.
- it was the worst of Times.
- it is the time of stupidity.
- it is the age of foolishness.

Form this above example, let's consider each line as a separate **"document"** and the 4 lines as our entire corpus of documents.

Vocabulary

What would be the total vocabulary???

Bag of Words (BoW) Model

1. Design the Vocabulary

The unique words by ignoring case, punctuations, and making them into root words are:

1. it
2. was
3. the
4. best
5. of
6. time
7. worst
8. stupidity
9. is
10. age
11. foolishness

Vocabulary contains 11 words while the full corpus contains 24 words.

2. Create Document Vectors

The objective is to turn each document of text into a vector so that we can use as input or output for a machine learning model.

Because we know the vocabulary has 11 words, we can use a fixed-length document representation of 11, with one position in the vector to score each word. The simplest scoring method is to mark the presence of words as a boolean value, 0 for absent, non-zero (positive value) for present. There can be other methods such as count based methods of the terms if more than one occurance of a trem.

In this example the binary vector of four documents would look as follows:

	it	was	the	best	of	time	worst	stupidity	is	age	foolishness
Document #1 [It was the best of times.]	1	1	1	1	1	1	0	0	0	0	0
Document #2 [it was the worst of Times.]	1	1	1	0	1	1	1	0	0	0	0
Document #3 [it is the time of stupidity.]	1	0	1	0	1	1	0	1	1	0	0
Document #4 [it is the age of foolishness.]	1	0	1	0	1	0	0	0	1	1	1

Problems

- Ordering of words have been discarded which **ignores the context**. These unordered words **can't preserve document semantics** For instance, **“this is interesting”** vs **“is this interesting”**. Moreover, "**stupidity**" and "**foolishness**" are considered two different words in the dictionary.
- We are retaining no information on the **grammar of the sentences**.

- New documents that overlap with the vocabulary of known words, but may contain **words outside of the vocabulary**.
- If the vocabulary size increases the **document representation dimension** also increases.

Managing Vocabulary

In the previous example, the **length of the document vector** is equal to the number of known words which is 11 words.

For a very large corpus, such as thousands of books, the length of the vector **might be thousands or millions of positions**. Further, each document may contain **very few of the known words in the vocabulary**. This results in a vector with lots of zero scores, called a sparse vector or sparse representation. Sparse vectors require more memory and computational resources **(space and time complexity)**

It's very important to decrease the size of the vocabulary when using a bag-of-words model.

Solution #1

There are simple text cleaning techniques that can be used as a first step, such as:

- Ignoring case
- Ignoring punctuation
- Ignoring frequent words that don't contain much information, called stop words, like "a," "of," etc.
- Fixing misspelled words.
- Reducing words to their stem (e.g. "play" from "playing") using stemming algorithms.

Solution #2

Each word or token is called a "gram". Creating a vocabulary of two-word pairs is, in turn, called a **bigram model**.

An **N-gram** is an N-token sequence of words: a 2-gram (more commonly called a bigram) is a two-word sequence of words like "please turn", "turn your", or "your homework", and a **3-gram (more commonly called a trigram)** is a three-word sequence of words like "please turn your", or "turn your homework".

For example, the bigrams in the first line of text in the previous section: **"It was the best of times"** are as follows:

- "it was"
- "was the"
- "the best"

- “best of”
- “of times”

A vocabulary then tracks triplets of words is called a trigram model and the general approach is called the **n-gram model**, where n refers to the number of grouped words.

Note: Often a simple bigram approach is better than a 1-gram bag-of-words model.

One-Hot Representation

The one hot representation, as the name suggests, starts with a zero vector, and sets as 1 the corresponding entry in the vector if the word is present in the sentence or document.

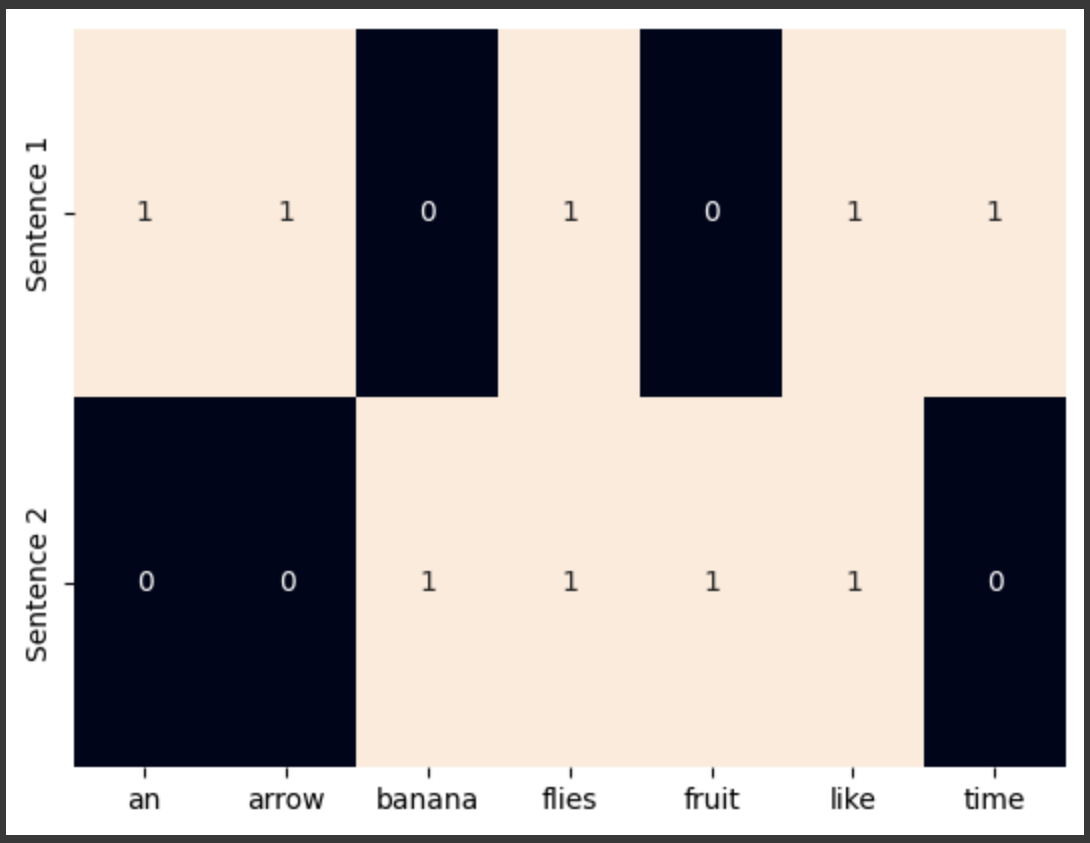
Tokenizing the sentences, ignoring punctuation, and treating everything as lowercase, will yield a vocabulary of size 8: `{time, fruit, flies, like, a, an, arrow, banana}`.

The binary encoding for **“like a banana”** would then be:

```
[0, 0, 0, 1, 1, 0, 0, 1]
```

```
from sklearn.feature_extraction.text import CountVectorizer
import seaborn as sns

corpus = ['Time flies flies like an arrow.',
          'Fruit flies like a banana.']

one_hot_vectorizer = CountVectorizer(binary=True)
one_hot = one_hot_vectorizer.fit_transform(corpus).toarray()

print (one_hot)

print (one_hot_vectorizer.vocabulary_)

dictionary = sorted(one_hot_vectorizer.vocabulary_)

print(dictionary)

sns.heatmap(one_hot, annot=True, cbar=False, xticklabels=dictionary,
                                             yticklabels=['Sentence 1','Sentence 2'])
```

```
[[1 1 0 1 0 1 1]
 [0 0 1 1 1 1 0]]
{'time': 6, 'flies': 3, 'like': 5, 'an': 0, 'arrow': 1, 'fruit': 4, 'banana': 2}
['an', 'arrow', 'banana', 'flies', 'fruit', 'like', 'time']
<Axes: >
```



Term Frequency (TF)

Term Frequent (**TF**) is a measure of how frequently a term, t, appears in a document, d:

$$TF_{t,d} = \frac{n_{t,d}}{\text{Total number of terms in document } d}$$

$n_{t,d}$ = Number of times term t appears in a document d. Thus, each document and term would have its own **TF** value.

Consider these 3 documents like **BoW** model:-

- It was the best of the time.
- it was the worst of Times.
- it is the time of stupidity.

The vocabulary or dictionary of the entire corpus would be:-

1. it
2. was
3. the
4. best
5. of
6. time
7. worst
8. is
9. stupidity

Now we will calculate the **TF** values for the **Document 3**.

Document 3 :- **it is the time of stupidity.**

- Number of words in Document 3 = **6**
- TF for the word **'the'** = (number of times **'the'** appears in Document 3) / (number of terms in Document 3) = **1/6**

Likewise:-

- TF('**it**') = 1/6
- TF('**was**') = 0/6 = 0
- TF('**the**') = 1/6
- TF('**best**') = 0/6 = 0
- TF('**of**') = 1/6
- TF('**time**') = 1/6
- TF('**worst**') = 0/6 = 0
- TF('**is**') = 1/6
- TF('**stupidity**') = 1/6

We can calculate all the term frequencies for all the terms of all the documents in this manner:-

Term	Document#1	Document#2	Document#3	TF (Document#1)	TF (Document#2)	TF (Document#3)
it	1	1	1	1/7	1/6	1/6
was	1	1	0	1/7	1/6	0
the	2	1	1	2/7	1/6	1/6
best	1	0	0	1/7	0	0
of	1	1	1	1/7	1/6	1/6
time	1	1	1	1/7	1/6	1/6
worst	0	1	0	0	1/6	0
is	0	0	1	0	0	1/6
stupidity	0	0	1	0	0	1/6

```
import math

print(math.log((3),10))

print(math.log((330),10))

print(math.log((3/3),10))

print(math.log((4/3),10))

print(math.log((4/5),10))
```

```
0.47712125471966244
2.518513939877887
0.0
0.1249387366082999
-0.09691001300805638
```

Inverse Document Frequency (IDF)

IDF is a measure of how important a term is. We need the IDF value because computing just the **TF alone is not sufficient** to understand the importance of words:

$$IDF_t = log\ (\frac{\text{Total Number of Documents}}{\text{The Number of Documents with Term } t})$$

A problem with scoring word frequency is that highly frequent words **('is', 'the', 'a' etc)** start to dominate in the document (e.g. larger score), but may not contain as much **"useful information"** to the model comapre to the rarer but **domain specific words**.

One approach is to rescale the frequency of words by **how often they appear in all documents**, so that the scores for frequent words like "the" that are also frequent **across all documents are penalized**.

This approach to scoring is called Term Frequency – Inverse Document Frequency, or TF-IDF for short, where:

- **Term Frequency:** is a scoring of the frequency of the word in the current document.
- **Inverse Document Frequency:** is a scoring of how rare the word is across documents.

Thus the idf of a rare term is high, whereas the idf of a frequent term is likely to be low.

We can calculate the IDF values for **Document 3**:

Document 3 :- **it is the time of stupidity.**

IDF('it') = log(total number of documents/number of documents containing the word 'it') = log(3/3) = log(1) = 0

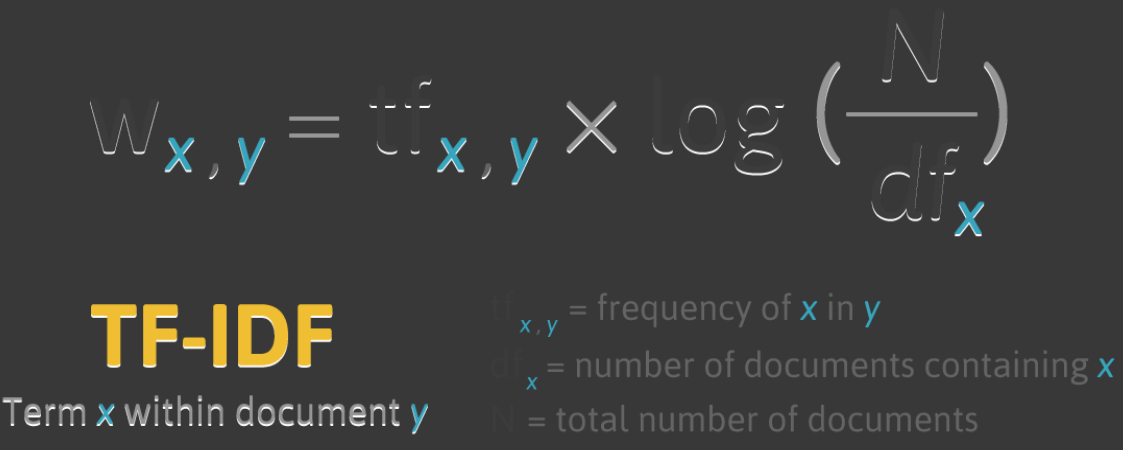
We can calculate the IDF values for each word like this. Thus, the IDF values for the entire vocabulary would be:

Term	Document#1	Document#2	Document#3	IDF
it	1	1	1	0.00
was	1	1	0	0.18
the	2	1	1	0.00
best	1	0	0	0.48
of	1	1	1	0.00
time	1	1	1	0.00
worst	0	1	0	0.48
is	0	0	1	0.48
stupidity	0	0	1	0.48

We can now compute the TF-IDF score for each word in the corpus. Words with a higher score are more important, and those with a lower score are less important:

$$(TF - IDF)_{t,d} = TF_{t,d} * IDF_t$$

You can find the overall summary in the following figure.



We can now calculate the TF-IDF score for every word in **Document 3**:

Document 3 :- **it is the time of stupidity.**

TF-IDF('it', Document 3) = TF('it', Document 3) * IDF('it') = 1/6 * 0 = 0

Likewise:-

- TF('**it**') = (1/6) * 0 = 0
- TF('**is**') = (1/6) * 0.48
- TF('**the**') = (1/6) * 0 = 0
- TF('**best**') = (0/6) * 0.48 = 0
- TF('**time**') = (1/6) * 0 = 0
- TF('**of**') = (1/6) * 0 = 0
- TF('**stupidity**') = (1/6) * 0.48

Similarly, we can calculate the TF-IDF scores for all the words with respect to all the documents.

- First, notice how if there is a very common word that occurs in all documents (i.e., n = N), IDF(w) is 0 and the TFIDF score is 0, thereby completely penalizing that term.
- Second, if a term occurs very rarely, perhaps in only one document, the IDF will be the maximum possible value, log N

```
from sklearn.feature_extraction.text import TfidfVectorizer
import seaborn as sns
import matplotlib as plt

corpus = ['Neural networks are a fundamental component of artificial intelligence, playing a
          'Their ability to mimic the human brain interconnected structure and learning capa
          'Neural networks have revolutionized various industries, such as healthcare, finar
          'They have significantly enhanced natural language processing, making virtual assi
          'Furthermore, neural networks have propelled computer vision to new heights, enabl

tfidf_vectorizer = TfidfVectorizer()
tfidf = tfidf_vectorizer.fit_transform(corpus).toarray()

print (tfidf)

print (tfidf_vectorizer.vocabulary_)

dictionary = sorted(tfidf_vectorizer.vocabulary_)

print(dictionary)


sns.heatmap(tfidf, annot=True, cbar=False,linewidths=.5, xticklabels=dictionary,
                                          yticklabels=['Sentence 1','Sentence 2','Sentence
```

```
[[0.         0.         0.26824958 0.         0.         0.26824958
  0.26824958 0.         0.         0.         0.         0.
  0.         0.         0.26824958 0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.26824958 0.         0.         0.         0.         0.
  0.26824958 0.         0.26824958 0.         0.         0.
  0.         0.         0.         0.         0.26824958 0.
  0.         0.17964987 0.17964987 0.         0.         0.26824958
  0.         0.26824958 0.26824958 0.         0.         0.
  0.         0.         0.26824958 0.         0.         0.
  0.         0.26824958 0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.        ]
 [0.21862917 0.21862917 0.         0.         0.12317186 0.
  0.         0.         0.         0.         0.21862917 0.
  0.21862917 0.21862917 0.         0.         0.         0.
  0.21862917 0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.21862917
  0.         0.         0.         0.21862917 0.         0.21862917
  0.         0.         0.         0.21862917 0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.21862917 0.         0.
  0.         0.         0.         0.         0.21862917 0.21862917
  0.         0.         0.21862917 0.21862917 0.21862917 0.
  0.35277728 0.         0.21862917 0.         0.         0.
  0.         0.         0.21862917]
 [0.         0.         0.         0.23177513 0.26115613 0.
  0.         0.23177513 0.         0.23177513 0.         0.23177513
  0.         0.         0.         0.         0.23177513 0.
  0.         0.         0.         0.23177513 0.         0.23177513
  0.         0.         0.15522251 0.23177513 0.         0.
  0.         0.23177513 0.         0.         0.         0.
  0.         0.         0.23177513 0.         0.         0.
  0.         0.15522251 0.15522251 0.         0.         0.
  0.23177513 0.         0.         0.         0.         0.
  0.         0.23177513 0.         0.         0.         0.
  0.23177513 0.         0.         0.         0.         0.
  0.         0.         0.         0.23177513 0.23177513 0.23177513
  0.         0.         0.        ]
 [0.         0.         0.         0.         0.1418874  0.
  0.         0.         0.25184911 0.         0.         0.
  0.         0.         0.         0.         0.         0.25184911
  0.         0.         0.25184911 0.         0.         0.
  0.         0.         0.16866629 0.         0.         0.
  0.         0.         0.         0.         0.50369823 0.
  0.         0.25184911 0.         0.         0.         0.25184911
  0.25184911 0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.25184911 0.
  0.         0.         0.         0.25184911 0.         0.
  0.         0.         0.         0.         0.         0.25184911
  0.         0.25184911 0.         0.         0.         0.
  0.25184911 0.         0.        ]
 [0.         0.         0.         0.         0.14418901 0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.25593446 0.         0.
  0.         0.25593446 0.         0.         0.25593446 0.
```

```
       0.         0.25593446 0.17140229 0.         0.25593446 0.
       0.         0.         0.         0.         0.         0.
```

Summary

Bag of Words just creates a set of vectors containing the count of word occurrences in the document, while the TF-IDF model contains information on the more important words and the less important ones as well.

Bag of Words vectors are easy to interpret. However, TF-IDF usually performs better in machine learning models.

Understanding the context of words is important. Detecting the similarity between the words 'time' and 'age', or 'stupidity' and 'foolishness'.

This is where Word Embedding techniques such as **Word2Vec, Continuous Bag of Words (CBOW), Skipgram**, etc come into play.

Bag-of-Words Text Classification

We will show how to build a simple Bag of Words (BoW) text classifier using PyTorch. The classifier is trained on IMDB movie reviews dataset.

```
from pathlib import Path

import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from google_drive_downloader import GoogleDriveDownloader as gdd
from torch.utils.data import DataLoader, Dataset
from sklearn.feature_extraction.text import CountVectorizer
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
```

```
device(type='cpu')
```

```
# DATA_PATH = '/content/imdb.csv'
# if not Path(DATA_PATH).is_file():
#     gdd.download_file_from_google_drive(
#         file_id='1EWReHFoPXK2Z-zdELR_LC6J6VRgp-QgN',
#         dest_path=DATA_PATH,
#     )
```

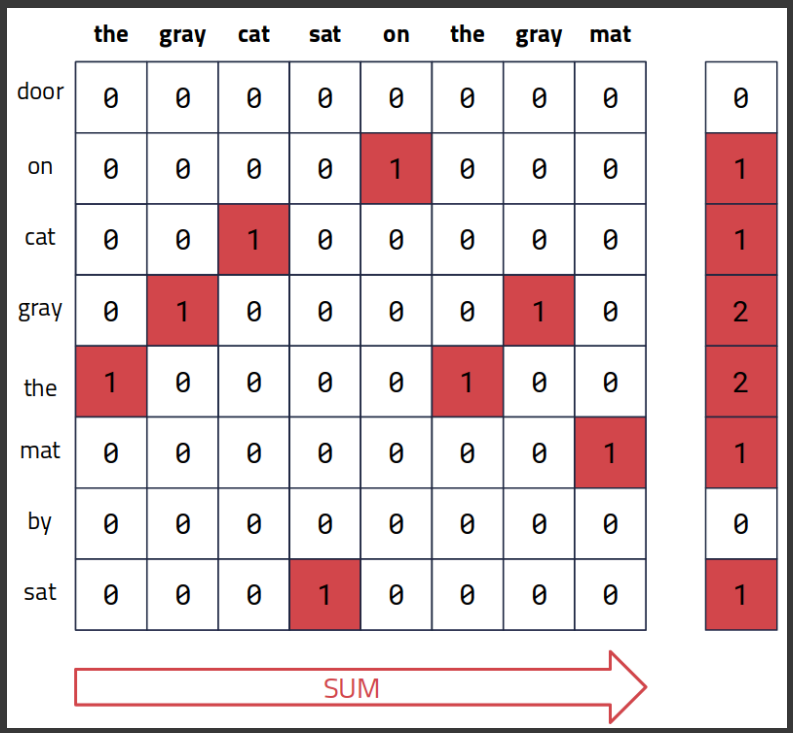
```
# Upload imdb.csv file in colab
DATA_PATH = '/content/imdb.csv'
import pandas as pd
df=pd.read_csv(DATA_PATH)
print(df.head())
```

```
                                              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
```

```
import numpy as np
x=np.array(pd)
print(x)
```

```
<module 'pandas' from '/usr/local/lib/python3.10/dist-packages/pandas/__init__.py'>
```

Bag-of-Words Sentiment Classification



So the final bag-of-words vector for `['the', 'gray', 'cat', 'sat', 'on', 'the', 'gray', 'mat']` is `[0, 1, 1, 2, 2, 1, 0, 1]`

```
class Sequences(Dataset):
    def __init__(self, data):
        self.vectorizer = CountVectorizer(stop_words='english')
        self.sequences = self.vectorizer.fit_transform(data.review.tolist())
        self.labels = data.sentiment.tolist()
        self.token2idx = self.vectorizer.vocabulary_
        self.idx2token = {idx: token for token, idx in self.token2idx.items()}

    def __getitem__(self, i):
        return self.sequences[i, :].toarray(), self.labels[i]

    def __len__(self):
        return self.sequences.shape[0]
```

```
df = pd.read_csv(DATA_PATH)

print((df))
codes=[0,1]
df.columns = ["review", "sentiment"]
df["sentiment"] = df["sentiment"].astype('category')
df["sentiment"] = df["sentiment"].cat.codes

df_train = df.head(900)
df_test = df.tail(100)
print(df_train)
dataset = Sequences(df_train)

train_loader = DataLoader(dataset, batch_size=900)
```

```
                                                 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
...                                                 ...       ...
1494  Zoey 101 is basically about a girl named Zoey ...  negative
1495  This movie is terrible, it was so difficult to...  negative
1496  The only thing serious about this movie is the...  positive
1497  2005 was one of the best year for movies. We h...  positive
1498  According to John Ford's lyrically shot, ficti...  positive

[1499 rows x 2 columns]
                                                review  sentiment
0    One of the other reviewers has mentioned that ...          1
1    A wonderful little production. <br /><br />The...          1
2    I thought this was a wonderful way to spend ti...          1
3    Basically there's a family where a little boy ...          0
4    Petter Mattei's "Love in the Time of Money" is...          1
..                                                 ...        ...
895  But it is kinda hilarious, at least if you gre...          1
896  One of the two Best Films of the year. A well ...          1
897  I managed to see this at the New York Internat...          1
```

```
898  Why else would he do this to me?<br /><br />No...          0
899  Minimal script, minimal character development,...          0

[900 rows x 2 columns]
```

```
class BagOfWordsClassifier(nn.Module):
    def __init__(self, vocab_size, hidden1, hidden2):
        super().__init__()
        ### 1st hidden layer: vocab_size --> 128
        self.linear_1 = nn.Linear(vocab_size, hidden1)
        ### Non-linearity in 1st hidden layer
        self.relu_1 = nn.ReLU()

        ### 2nd hidden layer: 128 --> 64
        self.linear_2 = nn.Linear(hidden1, hidden2)
        ### Non-linearity in 2nd hidden layer
        self.relu_2 = nn.ReLU()

        ### Output layer: 64 --> 1
        self.linear_out = nn.Linear(hidden2, 1)

    def forward(self, inputs):
        ### 1st hidden layer
        out = self.linear_1(inputs.squeeze(1).float())
        ### Non-linearity in 1st hidden layer
        out = self.relu_1(out)

        ### 2nd hidden layer
        out = self.linear_2(out)
        ### Non-linearity in 2nd hidden layer
        out = self.relu_2(out)

        # Linear layer (output)
        logits  = self.linear_out(out)

        return logits
```

```
model = BagOfWordsClassifier(len(dataset.token2idx), 128, 64)
model
```

```
BagOfWordsClassifier(
  (linear_1): Linear(in_features=16683, out_features=128, bias=True)
  (relu_1): ReLU()
  (linear_2): Linear(in_features=128, out_features=64, bias=True)
  (relu_2): ReLU()
  (linear_out): Linear(in_features=64, out_features=1, bias=True)
)
```

```
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
train_losses = []

for epoch in range(50):
    losses = []
    total = 0
    for inputs, target in train_loader:
        model.zero_grad()
        #print(target)
        output = model(inputs)
        loss = criterion(output.squeeze(), target.float())

        loss.backward()

        optimizer.step()

        losses.append(loss.item())
        total += 1

    epoch_loss = sum(losses) / total
    train_losses.append(epoch_loss)

    print(f'Epoch #{epoch + 1}\tTrain Loss: {epoch_loss:.3f}')
```

```
Epoch #1        Train Loss: 0.693
Epoch #2        Train Loss: 0.677
Epoch #3        Train Loss: 0.656
Epoch #4        Train Loss: 0.628
Epoch #5        Train Loss: 0.593
Epoch #6        Train Loss: 0.553
Epoch #7        Train Loss: 0.510
Epoch #8        Train Loss: 0.465
Epoch #9        Train Loss: 0.418
Epoch #10       Train Loss: 0.371
Epoch #11       Train Loss: 0.325
Epoch #12       Train Loss: 0.281
Epoch #13       Train Loss: 0.240
Epoch #14       Train Loss: 0.203
Epoch #15       Train Loss: 0.169
Epoch #16       Train Loss: 0.140
Epoch #17       Train Loss: 0.115
Epoch #18       Train Loss: 0.094
Epoch #19       Train Loss: 0.077
Epoch #20       Train Loss: 0.062
Epoch #21       Train Loss: 0.050
Epoch #22       Train Loss: 0.041
Epoch #23       Train Loss: 0.033
Epoch #24       Train Loss: 0.027
Epoch #25       Train Loss: 0.022
Epoch #26       Train Loss: 0.018
Epoch #27       Train Loss: 0.015
Epoch #28       Train Loss: 0.012
Epoch #29       Train Loss: 0.010
```

```
Epoch #30       Train Loss: 0.009
Epoch #31       Train Loss: 0.007
Epoch #32       Train Loss: 0.006
Epoch #33       Train Loss: 0.005
Epoch #34       Train Loss: 0.005
Epoch #35       Train Loss: 0.004
Epoch #36       Train Loss: 0.004
Epoch #37       Train Loss: 0.003
Epoch #38       Train Loss: 0.003
Epoch #39       Train Loss: 0.003
Epoch #40       Train Loss: 0.002
Epoch #41       Train Loss: 0.002
Epoch #42       Train Loss: 0.002
Epoch #43       Train Loss: 0.002
Epoch #44       Train Loss: 0.002
Epoch #45       Train Loss: 0.002
Epoch #46       Train Loss: 0.001
Epoch #47       Train Loss: 0.001
Epoch #48       Train Loss: 0.001
Epoch #49       Train Loss: 0.001
Epoch #50       Train Loss: 0.001
```

```
def predict_sentiment(text):
  test_vector = torch.LongTensor(dataset.vectorizer.transform([text]).toarray())

  output = model(test_vector)

  prediction = torch.sigmoid(output).item()

  if prediction > 0.5:
    print(f'{prediction:0.3}: Positive sentiment')
    return 1
  else:
    print(f'{prediction:0.3}: Negative sentiment')
    return 0
```

```
test_text = "The story itself is just predictable and lazy."
predict_sentiment(test_text)
```

```
0.4: Negative sentiment
0
```

```
test_text = "This movie is not bad."
predict_sentiment(test_text)
```

```
0.397: Negative sentiment
0
```

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import cohen_kappa_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import confusion_matrix
```

```
from sklearn.metrics import classification_report
```

```
pred_labels = []

sentences = list(df_test['review'])
labels = df_test['sentiment']

print(sentences)

for sentence in sentences:
  pred_labels.append(predict_sentiment(sentence))

# accuracy: (tp + tn) / (p + n)
accuracy = accuracy_score(labels, pred_labels)
print('Accuracy: %f' % accuracy)

# precision tp / (tp + fp)
precision = precision_score(labels, pred_labels)
print('Precision: %f' % precision)

# recall: tp / (tp + fn)
recall = recall_score(labels, pred_labels)
print('Recall: %f' % recall)

# f1: 2 tp / (2 tp + fp + fn)
f1 = f1_score(labels, pred_labels)
print('F1 score: %f' % f1)

# confusion matrix
matrix = confusion_matrix(labels, pred_labels)
print(matrix)
print(classification_report(labels, pred_labels,digits=4))
```

```
["I love this show as it action packed with adventure, love and intrigue. Well some t
0.847: Positive sentiment
0.865: Positive sentiment
1.0: Positive sentiment
1.0: Positive sentiment
0.997: Positive sentiment
0.00212: Negative sentiment
0.808: Positive sentiment
0.926: Positive sentiment
0.174: Negative sentiment
```

```
0.927: Positive sentiment
0.871: Positive sentiment
0.00621: Negative sentiment
0.976: Positive sentiment
0.166: Negative sentiment
0.718: Positive sentiment
0.0199: Negative sentiment
0.489: Negative sentiment
0.638: Positive sentiment
0.334: Negative sentiment
0.901: Positive sentiment
0.996: Positive sentiment
0.0022: Negative sentiment
0.506: Positive sentiment
0.0113: Negative sentiment
0.999: Positive sentiment
0.797: Positive sentiment
0.00419: Negative sentiment
0.173: Negative sentiment
0.552: Positive sentiment
0.0473: Negative sentiment
0.925: Positive sentiment
0.0196: Negative sentiment
0.82: Positive sentiment
0.00134: Negative sentiment
0.793: Positive sentiment
0.0159: Negative sentiment
0.346: Negative sentiment
0.145: Negative sentiment
0.956: Positive sentiment
0.999: Positive sentiment
0.161: Negative sentiment
0.964: Positive sentiment
0.212: Negative sentiment
0.113: Negative sentiment
0.855: Positive sentiment
0.997: Positive sentiment
1.0: Positive sentiment
0.991: Positive sentiment
0.45: Negative sentiment
0.586: Positive sentiment
```