NLP TUTORIAL PART 2

By Prashant Sundge

NLP TUTORIAL PART 1

- Kaggle : Prashant Kumar Sundge
- Github : prashantsundge
- Linked In : prashantsundge

Colab Markdown Index

7. NLP Preprocessing Workflow Index

8. Model Building

BAG OF WORDS

Bag of Words (BoW) Approach

The Bag of Words (BoW) is a simple yet powerful technique used in natural language processing (NLP) for text analysis and document classification. It represents a document as a collection of words, disregarding grammar and word order, and focuses solely on the presence or absence of words.

How BoW Works

1. **Tokenization:** The first step is to tokenize the text, splitting it into individual words or tokens. Punctuation marks are typically removed during this process.
2. **Vocabulary Creation:** Next, a vocabulary is created by collecting all unique words from the entire corpus of documents. Each word in the vocabulary is assigned a unique index.
3. **Vectorization:** Each document is represented as a vector, where the length of the vector is equal to the size of the vocabulary. The value at each index represents the frequency of the corresponding word in the document. If a word appears multiple times in the document, its corresponding value will be higher.

Example

Consider the following two documents:

Consider the following two documents:

- Document 1: "This is a very good movie."
- Document 2: "This is not a good movie."

Step 1: Tokenization

Tokenizing each document results in the following sets of words:

- Document 1: { "this", "is", "a", "very", "good", "movie" }
- Document 2: { "this", "is", "not", "a", "good", "movie" }

Step 2: Vocabulary Creation

Creating the vocabulary involves collecting all unique words from both documents:

Vocabulary: { "this", "is", "a", "very", "good", "movie", "not" }

Step 3: Vectorization

Each document is represented as a vector based on the frequency of words in the vocabulary:

- Document 1: [1, 1, 1, 1, 1, 1, 0]
- Document 2: [1, 1, 1, 0, 1, 1, 1]

In the vector representation, each position corresponds to a word in the vocabulary, and the value at each position indicates the frequency of that word in the document.

In [ ]:

```
import numpy as np
import pandas as pd
```

In [ ]:

```
df=pd.DataFrame({'text':['people Watch movies','movies watch movies','people write comme
nt','movies write comment'], 'output':[1,1,0,0]})
```

In [ ]:

```
df
```

Out[ ]:

	text	output
0	people Watch movies	1
1	movies watch movies	1
2	people write comment	0
3	movies write comment	0

In [ ]:

```
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer()
```

In [ ]:

```
bow=cv.fit_transform(df['text'])
```

In [ ]:

```
#vocab
print(cv.vocabulary_)
```

```
{'people': 2, 'watch': 3, 'movies': 1, 'write': 4, 'comment': 0}
```

In [ ]:

```
cv.get_feature_names_out()
```

Out[ ]:

```
array(['comment', 'movies', 'people', 'watch', 'write'], dtype=object)
```

In [ ]:

```
print(bow.toarray())
```

```
[[0 1 1 1 0]
 [0 2 0 1 0]
 [1 0 1 0 1]
 [1 1 0 0 1]]
```

In [ ]:

```
bow.shape
```

Out[ ]:

```
(4, 5)
```

In [ ]:

```
print(bow[0].toarray())
print(bow[1].toarray())
```

```
[[0 1 1 1 0]]
[[0 2 0 1 0]]
```

In [ ]:

```
print(bow[2])
```

```
  (0, 2) 1
  (0, 4) 1
  (0, 0) 1
```

In [ ]:

```
cv.transform(["campusx watch and write comment of campusx"]).toarray()
```

Out[ ]:

```
array([[1, 0, 0, 1, 1]])
```

Pros and Cons of Bag of Words (BoW) Model

Pros:

1. **Simplicity:** BoW is easy to understand and implement, making it accessible even for beginners in natural language processing (NLP).
2. **Efficiency:** It is computationally efficient, especially for large corpora, as it only requires counting the frequency of words.
3. **Versatility:** BoW can be used for various NLP tasks such as text classification, sentiment analysis, and document clustering.
4. **Language Agnostic:** BoW is language-agnostic, meaning it can be applied to any language without significant modifications.
5. **Robustness:** BoW can handle misspellings and variations of words well, as it focuses on word frequency rather than exact word matching.

Cons:

1. **Loss of Word Order**: BoW disregards the order of words in the document, leading to a loss of sequential information and context.
2. **Sparse Representation**: In datasets with large vocabularies, BoW representations can be very sparse, leading to high-dimensional feature vectors and increased computational complexity.
3. **Lack of Semantic Understanding**: BoW treats each word as independent, ignoring the semantic relationships between words and resulting in a shallow understanding of the text.
4. **Inability to Capture Phrases**: BoW fails to capture phrases or multi-word expressions, which can be crucial for tasks requiring contextual understanding.
5. **Limited to Bag of Words**: BoW does not consider other linguistic features such as part-of-speech tags, syntactic structure, or semantic meaning, limiting its expressive power compared to more advanced models.

BAG OF WORDS HYPERPARAMETERS

In [ ]:

```
from sklearn.feature_extraction.text import CountVectorizer

# Sample text data
text_data = ["This is a very good movie.", "This is not a good movie."]



# Custom vocabularies
custom_vocabulary_dict = {'this': 0, 'is': 1, 'good': 2, 'movie': 3}
custom_vocabulary_list = ['this', 'very', 'good', 'movie']

# Initialize CountVectorizer with custom hyperparameters
cv = CountVectorizer(
    input='content',            # Input type is 'content'
    encoding='utf-8',           # Encoding is UTF-8
    decode_error='strict',      # Decoding errors handled strictly
    strip_accents=None,         # Do not strip accents
    lowercase=True,             # Convert all characters to lowercase
    preprocessor=None,          # No custom preprocessing function
    tokenizer=None,             # No custom tokenizer function
    stop_words=None,            # No stopwords removal
    token_pattern='(?u)\\b\\w\\w+\\b',  # Regular expression pattern for tokenization
    ngram_range=(1, 2),         # Generate unigrams and bigrams
    analyzer='word',            # Generate N-grams at the word level
    max_df=1.0,                 # No maximum document frequency threshold
    min_df=1,                   # Minimum document frequency is 1
    max_features=None,          # No maximum number of features you want to use in featurs
    vocabulary=custom_vocabulary_list,           # Automatically determine the vocabular
y we can specify the imp vocabulary
    binary=False                # Term frequencies are not binary
)

# Fit and transform the text data
ngrams = cv.fit_transform(text_data)

# Get the vocabulary (feature names)
vocab = cv.get_feature_names_out()

# Convert the result to a DataFrame for visualization
ngrams_df = pd.DataFrame(ngrams.toarray(), columns=vocab)
print(ngrams_df)
```

```
   this  very  good  movie
0     1     1     1      1
1     1     0     1      1
```

N-Grams

N-grams are contiguous sequences of n items from a given sample of text or speech. In the context of text

processing, these items are usually words, but they can also be letters or syllables depending on the application. Here are some examples of n-grams in English:

Unigrams (1-grams): Single words.

Example: "apple", "banana", "orange" Bigrams (2-grams): Sequences of two adjacent words.

Example: "big data", "machine learning", "natural language" Trigrams (3-grams): Sequences of three adjacent words.

Example: "artificial intelligence", "deep learning model", "data science project" Four-grams (4-grams): Sequences of four adjacent words.

Example: "neural network architecture", "supervised learning algorithms", "text classification task"

In natural language processing (NLP), it's common to limit the maximum length of N-grams generated from a document. This limitation is applied to ensure efficient processing and to prevent potential issues such as memory overflow.

First, let's generate the unigrams, bigrams, and trigrams for each document:

Document 1: "This is a very good movie."

- **Unigrams: ["this", "is", "a", "very", "good", "movie"]**
- **Bigrams: ["this is", "is a", "a very", "very good", "good movie"]**
- **Trigrams: ["this is a", "is a very", "a very good", "very good movie"]**

Document 2: "This is not a good movie."

- **Unigrams: ["this", "is", "not", "a", "good", "movie"]**
- **Bigrams: ["this is", "is not", "not a", "a good", "good movie"]**
- **Trigrams: ["this is not", "is not a", "not a good", "a good movie"]**

Now, let's compare the N-gram representations of the documents:

Bigrams:

- **Document 1: ["this is", "is a", "a very", "very good", "good movie"]**
- **Document 2: ["this is", "is not", "not a", "a good", "good movie"]**

In the bigram representation, we can see that the bigram "not good" appears in Document 2 but not in Document 1. This indicates a negation ("not") followed by a positive adjective ("good"), suggesting a negative sentiment.

Trigrams:

- **Document 1: ["this is a", "is a very", "a very good", "very good movie"]**
- **Document 2: ["this is not", "is not a", "not a good", "a good movie"]**

Similarly, in the trigram representation, we observe the trigram "not a good" only in Document 2, which reinforces the presence of negation followed by a positive adjective.

By considering N-grams, we capture more contextual information, including sequences of words, which helps in distinguishing the opposite meanings between the documents. In this case, N-grams provide better insight into the sentiment expressed in each document compared to the Bag of Words approach.

In [ ]:

```
from nltk import ngrams
sentence= 'You must be the change you wish to see in the world'

unigram=ngrams(sentence.split(),1)
print("-"*100)
print("Unigrams")
for grams in unigram:
  print(grams)

bigrams=ngrams(sentence.split(),2)
```

```
print("-"*100)
print("Bigrams")
for grams in bigrams:
  print(grams)


trigram=ngrams(sentence.split(),3)
print("-"*100)
print("Bigrams")
for grams in trigram:
  print(grams)
```

```
----------------------------------------------------------------------------------------------------
Unigrams
('You',)
('must',)
('be',)
('the',)
('change',)
('you',)
('wish',)
('to',)
('see',)
('in',)
('the',)
('world',)
----------------------------------------------------------------------------------------------------
Bigrams
('You', 'must')
('must', 'be')
('be', 'the')
('the', 'change')
('change', 'you')
('you', 'wish')
('wish', 'to')
('to', 'see')
('see', 'in')
('in', 'the')
('the', 'world')
----------------------------------------------------------------------------------------------------
Bigrams
('You', 'must', 'be')
('must', 'be', 'the')
('be', 'the', 'change')
('the', 'change', 'you')
('change', 'you', 'wish')
('you', 'wish', 'to')
('wish', 'to', 'see')
('to', 'see', 'in')
('see', 'in', 'the')
('in', 'the', 'world')
```

In [ ]:

```
df

from sklearn.feature_extraction.text import CountVectorizer
ngram_range=(1,1)
cv = CountVectorizer(ngram_range=ngram_range)
ngrams=cv.fit_transform(df['text'])
print("Vocabulary ",cv.vocabulary_)
cv.get_feature_names_out()
print(ngrams.toarray())

print("-"*100)
ngram_range=(2,2)
cv = CountVectorizer(ngram_range=ngram_range)
ngrams=cv.fit_transform(df['text'])
print("Vocabulary ",cv.vocabulary_)
```

```
print(cv.get_feature_names_out())
ngrams.toarray()

cv.transform(["people watch movies in dec"]).toarray()
```

```
Vocabulary  {'people': 2, 'watch': 3, 'movies': 1, 'write': 4, 'comment': 0}
[[0 1 1 1 0]
 [0 2 0 1 0]
 [1 0 1 0 1]
 [1 1 0 0 1]]
--------------------------------------------------------------------------------
------------
Vocabulary  {'people watch': 2, 'watch movies': 4, 'movies watch': 0, 'people write': 3,
'write comment': 5, 'movies write': 1}
['movies watch' 'movies write' 'people watch' 'people write'
 'watch movies' 'write comment']
```

Out[ ]:

```
array([[0, 0, 1, 0, 1, 0]])
```

Pros and Cons of N-grams in Natural Language Processing (NLP)

Pros:

1. **Contextual Understanding:** N-grams capture contextual information by considering sequences of words, which helps in understanding the meaning and sentiment of text more accurately.
2. **Phrase Detection:** N-grams can detect and capture phrases or multi-word expressions, providing deeper insights into the structure and semantics of the text.
3. **Improved Feature Representation:** N-grams offer richer feature representations compared to Bag of Words (BoW) by preserving word order and capturing dependencies between adjacent words.
4. **Flexibility:** N-grams can be customized to different lengths (e.g., bigrams, trigrams) based on the specific requirements of the NLP task, offering flexibility in modeling text data.
5. **Robustness to Variations:** N-grams are robust to variations in word order, spelling, and syntax, making them suitable for handling noisy or unstructured text data.

Cons:

1. **Increased Dimensionality:** N-grams representations can lead to high-dimensional feature spaces, especially for longer sequences, which may increase computational complexity and require more data for training.
2. **Data Sparsity:** In datasets with limited text samples, some N-grams may have low frequencies or occur only in specific contexts, resulting in sparse representations and potential overfitting.
3. **Memory and Storage Requirements:** Storing and processing N-grams representations can require more memory and storage space compared to simpler models like BoW, especially for large datasets.
4. **Curse of Dimensionality:** The exponential growth of feature space with increasing N-gram length can exacerbate the curse of dimensionality, leading to challenges in model training and generalization.
5. **Semantic Ambiguity:** While N-grams capture local context, they may not capture broader semantic relationships between distant words in the text, potentially limiting their ability to understand complex linguistic structures.

TF-IDF

stands for "Term Frequency-Inverse Document Frequency." It is a statistical measure used in natural language processing and information retrieval to evaluate the importance of a word in a document relative to a collection of documents.

TF-IDF is calculated as the product of two metrics:

1. **Term Frequency (TF)**: This measures the frequency of a term (word) in a document. It indicates how often a term appears in a document relative to the total number of terms in that document. It is calculated as:

is calculated as:

$$\mathrm{TF}(t, d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

2. **Inverse Document Frequency (IDF)**: This measures the importance of a term across a collection of documents. It is calculated as the logarithm of the ratio of the total number of documents to the number of documents containing the term t:

$$\mathrm{IDF}(t, D) = \log\left(\frac{N}{\text{Number of documents containing term } t}\right)$$

where N is the total number of documents in the collection.

The TF-IDF score for a term in a document is the product of its TF and IDF scores:

$$\text{TF-IDF}(t, d, D) = \mathrm{TF}(t, d) \times \mathrm{IDF}(t, D)$$

TF-IDF assigns higher weights to terms that are frequent in a document but rare across the entire document collection. It helps in identifying the most relevant terms in a document for tasks such as document classification, information retrieval, and text mining.

Documents:

"people Watch movies"

"movies watch movies"

"people write comment"

"movies write comment"

manually calculate TF (Term Frequency) and IDF (Inverse Document Frequency) for the given documents.

Term Frequency (TF):

Document 1: "people Watch movies"

- Term Frequency of "people": $\frac{1}{3}$ (appears once out of three words)
- Term Frequency of "Watch": $\frac{1}{3}$ (appears once out of three words)
- Term Frequency of "movies": $\frac{1}{3}$ (appears once out of three words)

Document 2: "movies watch movies"

- Term Frequency of "movies": $\frac{2}{3}$ (appears twice out of three words)
- Term Frequency of "watch": $\frac{1}{3}$ (appears once out of three words)

Document 3: "people write comment"

- Term Frequency of "people": $\frac{1}{3}$ (appears once out of three words)
- Term Frequency of "write": $\frac{1}{3}$ (appears once out of three words)
- Term Frequency of "comment": $\frac{1}{3}$ (appears once out of three words)

Document 4: "movies write comment"

- Term Frequency of "movies": $\frac{1}{3}$ (appears once out of three words)
- Term Frequency of "write": $\frac{1}{3}$ (appears once out of three words)

- Term Frequency of "write": $\frac{1}{3}$ (appears once out of three words)
- Term Frequency of "comment": $\frac{1}{3}$ (appears once out of three words)

Inverse Document Frequency (IDF):

Total number of documents $N = 4$.

- IDF of "people": $\log\left(\frac{4}{2}\right) = \log(2) = 0.693$ (appears in 2 out of 4 documents)
- IDF of "Watch": $\log\left(\frac{4}{1}\right) = \log(4) = 1.386$ (appears in 1 out of 4 documents)
- IDF of "movies": $\log\left(\frac{4}{3}\right) = \log\left(\frac{4}{3}\right) = 0.287$ (appears in 3 out of 4 documents)
- IDF of "write": $\log\left(\frac{4}{2}\right) = \log(2) = 0.693$ (appears in 2 out of 4 documents)
- IDF of "comment": $\log\left(\frac{4}{2}\right) = \log(2) = 0.693$ (appears in 2 out of 4 documents)
- IDF of "watch": $\log\left(\frac{4}{1}\right) = \log(4) = 1.386$ (appears in 1 out of 4 documents)

These are the TF (Term Frequency) and IDF (Inverse Document Frequency) values for each term in the given documents.

In [ ]:

```
df
```

Out[ ]:

	text	output
0	people Watch movies	1
1	movies watch movies	1
2	people write comment	0
3	movies write comment	0

In [ ]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfid=TfidfVectorizer()

vector=tfid.fit_transform(df['text'])
print(tfid.get_feature_names_out())
vector.toarray()
```

```
['comment' 'movies' 'people' 'watch' 'write']
```

Out[ ]:

```
array([[0.        , 0.49681612, 0.61366674, 0.61366674, 0.        ],
       [0.        , 0.8508161 , 0.        , 0.52546357, 0.        ],
       [0.57735027, 0.        , 0.57735027, 0.        , 0.57735027],
       [0.61366674, 0.49681612, 0.        , 0.        , 0.61366674]])
```

In [ ]:

```
vector.shape
```

Out[ ]:

```
(4, 5)
```

Pros:

1. **Reflects Term Importance:** TF-IDF assigns higher weights to terms that are frequent in a document but rare

across the entire document collection. For example, in a collection of movie reviews, the term "cinematography" might have a high TF-IDF score in a positive review because it's relatively rare in the entire collection but frequent in positive reviews.
2. **Reduces Noise**: By giving less weight to common terms (stopwords) and more weight to rare terms, TF-IDF helps in reducing the impact of noisy and irrelevant terms in text data. For instance, in an analysis of customer feedback, the term "the" would likely have a low TF-IDF score because it's a common stopword and doesn't carry much meaningful information.
3. **Flexible**: TF-IDF can be easily customized by adjusting parameters such as stop words, n-gram range, and normalization techniques. For example, you can exclude domain-specific stopwords or adjust the n-gram range to capture different levels of phrase complexity depending on the task at hand.
4. **Language Agnostic**: TF-IDF can be applied to text data in any language, making it suitable for multilingual text analysis tasks. Whether you're analyzing English tweets or Japanese news articles, TF-IDF can effectively extract important terms and features from the text.
5. **Simple and Efficient**: TF-IDF is computationally efficient and easy to implement. It does not require complex feature engineering or deep linguistic knowledge. This simplicity makes it accessible to practitioners without specialized expertise in natural language processing.

Cons:

1. **Lack of Semantic Understanding**: TF-IDF does not consider the semantic relationships between terms. It treats each term independently, which may limit its ability to capture the full meaning of text data. For example, it may not distinguish between synonyms or related terms like "buy" and "purchase."
2. **Sensitivity to Document Length**: TF-IDF scores are influenced by the length of documents. Longer documents tend to have lower TF-IDF scores for individual terms compared to shorter documents, which may affect the relative importance of terms across documents. A longer document may have more occurrences of a term, but if it's also longer overall, those occurrences will be diluted in the TF-IDF calculation.
3. **Domain-specific Stopwords**: While TF-IDF can filter out common stopwords, it may not capture domain-specific stopwords or terms that are not explicitly defined as stopwords but are irrelevant in a specific context. For instance, in a medical text analysis, terms like "patient" or "treatment" might be common across all documents but still important for understanding the domain.
4. **Sparse Representation**: The TF-IDF matrix can be sparse, especially for large document collections with many unique terms. This sparse representation may require additional computational resources for storage and processing. For example, in a dataset with millions of documents and thousands of unique terms, the TF-IDF matrix may consume a large amount of memory.
5. **Limited Contextual Information**: TF-IDF considers only the frequency of terms within individual documents and across the entire document collection. It does not capture contextual information such as word order, syntactic structure, or co-occurrence patterns, which may be important in certain text analysis tasks. For example, it may not capture the relationship between "hot" and "cold" in the phrase "hot and cold beverages," which could be important for sentiment analysis or topic modeling.

Custom Features in Natural Language Processing (NLP)

Custom features in NLP refer to additional information or characteristics extracted from text data beyond the raw text itself. These features are designed to capture specific aspects of the text relevant to the task at hand, such as sentiment analysis, text classification, named entity recognition, or any other NLP task.

Examples of Custom Features:

1. **Word Embeddings**:
   - Example: Using pre-trained Word2Vec embeddings to represent words as dense vectors.
2. **Part-of-Speech (POS) Tags**:
   - Example: Extracting POS tags (e.g., noun, verb, adjective) for each word in a sentence.
3. **Named Entity Recognition (NER) Tags**:
   - Example: Identifying and classifying named entities (e.g., person names, organization names) in text.
4. **Sentiment Scores**:
   - Example: Computing sentiment scores (e.g., positive, negative, neutral) using sentiment lexicons or pre-

Example: Computing sentiment scores (e.g., positive, negative, neutral) using sentiment lexicons or pre-trained sentiment classifiers.

5. **Topic Modeling:**
   - Example: Extracting topic distributions from documents using Latent Dirichlet Allocation (LDA) or Non-negative Matrix Factorization (NMF).
6. **Text Length and Complexity:**
   - Example: Calculating text length, average word length, or readability metrics like Flesch-Kincaid Grade Level.
7. **Syntax Trees or Dependency Parses:**
   - Example: Parsing text to extract syntactic structures like syntax trees or dependency parses.
8. **Bag-of-Words (BoW) or TF-IDF Representations:**
   - Example: Creating Bag-of-Words (BoW) or TF-IDF vectors with custom preprocessing steps (e.g., stemming, lemmatization).

Choosing Features for Different Datasets and Dataset Types:

- **Text Classification:**
  - Features: Word embeddings, BoW or TF-IDF representations, POS tags, sentiment scores.
  - Example Datasets: Movie reviews (positive/negative sentiment), news articles (topic classification).
- **Named Entity Recognition (NER):**
  - Features: NER tags, word embeddings, syntax trees.
  - Example Datasets: News articles (identifying entities like person names, organization names), biomedical texts.
- **Sentiment Analysis:**
  - Features: Sentiment scores, word embeddings, BoW or TF-IDF representations.
  - Example Datasets: Social media data (tweets, comments), product reviews.
- **Topic Modeling:**
  - Features: Topic distributions, word embeddings, BoW or TF-IDF representations.
  - Example Datasets: Academic papers (identifying research topics), online forums (identifying discussion topics).

IMDB Dataset of 50K Movie Reviews

Define Library

In [ ]:

```
import gdown
import os
import zipfile
import pandas as pd
import seaborn as sns


import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
```

In [ ]:

```
url="https://drive.google.com/file/d/1xD3t-GybNU6Gqp3ebhYtCWI5wHfchE6X/view?usp=drive_lin
```

```
k"
prefix = "https://drive.google.com/uc?/export=download&id="
file_name= "movie_reviews.zip"
```

In [ ]:

```
def download_zip(url, prefix, file_name):
  file_id=url.split("/")[-2]
  gdown.download(prefix+file_id , file_name)
  print("File Downloaded in Colab")
  !unzip "/content/movie_reviews.zip"
```

In [ ]:

```
download_zip(url, prefix, file_name)
```

```
Downloading...
From (original): https://drive.google.com/uc?/export=download&id=1xD3t-GybNU6Gqp3ebhYtCWI5wHfchE6X
From (redirected): https://drive.google.com/uc?/export=download&id=1xD3t-GybNU6Gqp3ebhYtCWI5wHfchE6X&confirm=t&uuid=b3c73744-319e-4b17-a708-99acd5381459
To: /content/movie_reviews.zip
100%|██████████| 27.0M/27.0M [00:00<00:00, 81.7MB/s]
```

```
File Downloaded in Colab
Archive:  /content/movie_reviews.zip
  inflating: IMDB Dataset.csv
```

LOAD DATA in DATASET

In [ ]:

```
df = pd.read_csv("/content/IMDB Dataset.csv")
```

In [ ]:

```
df.head()
```

Out[ ]:

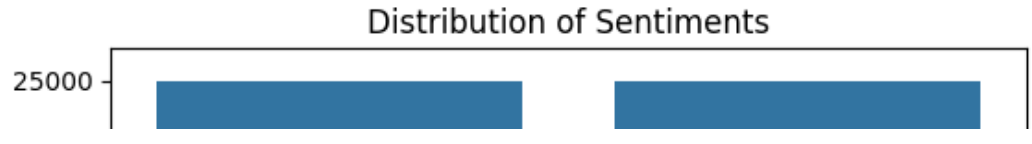
	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...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

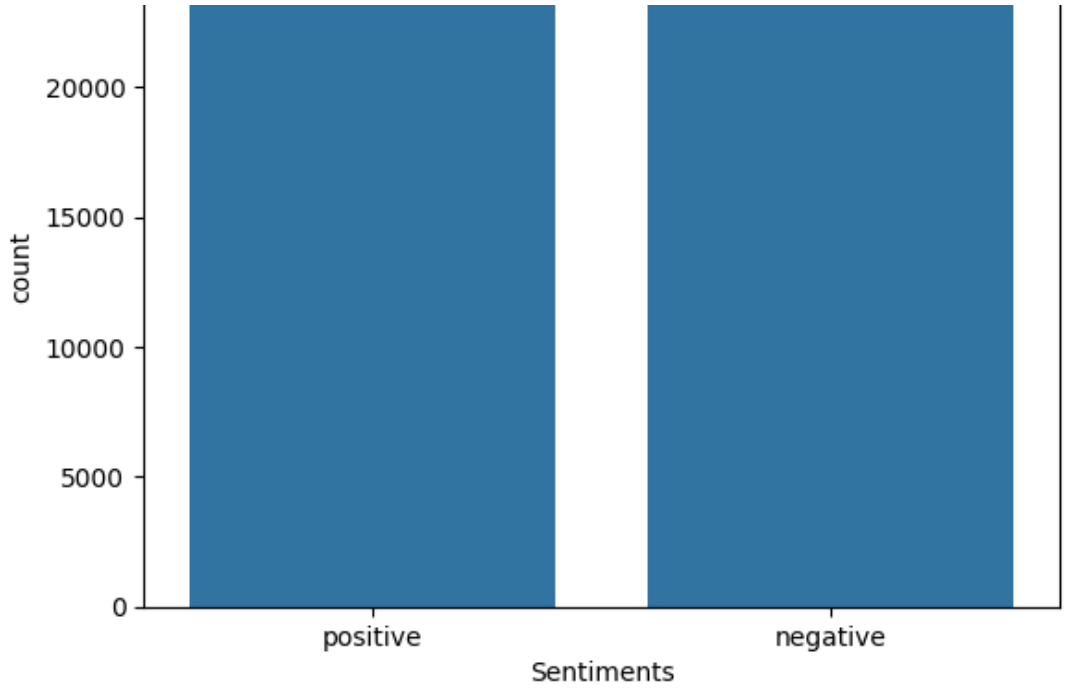
First will understand the sentiment by ploting the distribution

UNDERSTAND DISTRIBUTION OF OUTPUT COLUMN

In [ ]:

```
sns.countplot(x='sentiment' , data=df)
plt.xlabel('Sentiments')
plt.ylabel("count")
plt.title("Distribution of Sentiments")
plt.show()
```





- **From the distribution we could understand the both positive and negative sentiments are equal**

DATA PREPROCESSING

In [ ]:

```
df.describe()
```

Out[ ]:

	review	sentiment
count	50000	50000
unique	49582	2
top	Loved today's show!!! It was a variety and not...	positive
freq	5	25000

In [ ]:

```
df.dtypes
```

Out[ ]:

```
review       object
sentiment    object
dtype: object
```

In [ ]:

```
df.isnull().sum()
```

Out[ ]:

```
review       0
sentiment    0
dtype: int64
```

In [ ]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 2 columns):
```

```
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   review     50000 non-null  object
 1   sentiment  50000 non-null  object
dtypes: object(2)
memory usage: 781.4+ KB
```

In [ ]:

```
df.duplicated().sum()
```

Out[ ]:

```
418
```

In [ ]:

```
df.drop_duplicates(inplace = True)
```

In [ ]:

```
df.duplicated().sum()
```

Out[ ]:

```
0
```

Data Seems to be good no missing values no duplicates , data types are good

BEFORE PROCESING WILL UNDERSTAND THE DATASET BY FOLLOWING QUESTIONS

- **Find out the Corpus total size**
- **find out the document char size and word size**
- **find out the bigrams and trygrams**
- **find out the vocabulary**

In [ ]:

```
corpus_count = df['review'].count()
print("Document Count : ", corpus_count)
```

```
Document Count :  49582
```

In [ ]:

```
total_word_count = df['review'].str.split().apply(len).sum()
print(f"total words in Corpus is : {total_word_count}")
```

```
total words in Corpus is : 11470804
```

Sentence and Word Tokenize

In [ ]:

```
from nltk.tokenize import sent_tokenize, word_tokenize
import nltk
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[ ]:

```
True
```

In [ ]:

```
sentence = []
for sent in df['review']:
  sent_token = sent_tokenize(sent)
  sentence.append(sent_token)
print(len(sentence))
```

```
49582
```

VOCABULARY

In [ ]:

```
from sklearn.feature_extraction.text import CountVectorizer
cv= CountVectorizer()
bow=cv.fit_transform(df['review'])
print(f"Total Unique Words in Movie Review Dataset : {len(cv.vocabulary_)}")
```

```
Total Unique Words in Movie Review Dataset : 101895
```

- **Find the most occured words**
- **Apply bag of bi-grams and bag of tri-grams and write down your observation obout the dimensionality of the vocabulary**
- **apply tf-idf and find out the idf scores of words, also find out the vocabulary**

NLP Preprocessing Workflow Index

In [ ]:

```
df.head()
```

Out[ ]:

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. 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 [ ]:

In [ ]:

```
df['review'][3]
```

Out[ ]:

```
"Basically there's a family where a little boy (Jake) thinks there's a zombie in his clos
et & his parents are fighting all the time.<br /><br />This movie is slower than a soap o
pera... and suddenly, Jake decides to become Rambo and kill the zombie.<br /><br />OK, fi
rst of all when you're going to make a film you must Decide if its a thriller or a drama!
As a drama the movie is watchable. Parents are divorcing & arguing like in real life. And
then we have Jake with his closet which totally ruins all the film! I expected to see a B
OOGEYMAN similar movie, and instead i watched a drama with some meaningless thriller spot
s.<br /><br />3 out of 10 just for the well playing parents & descent dialogs. As for the
shots with Jake: just ignore them."
```

Note Down the Text issues

- **Lower text**
- **< > tags remove**
- **spelling mistakes**
- **old-time-BBC**
- **\'dream\'**
- **\x97a**
- **wwwaaaaayyyyy downhillthe**
- **httpwwwinvocusne**
- **wwwanswerscom**
- **showtimeewwwwwww from the manual text analysis we could see above errors in text which we will clean up below**

FIRST WILL CLEAN THE DATA USING REGULAR EXPRESSION

Lower Text

In [ ]:

```
# make copy of dataset
df_backup = df.copy()
```

regx

1. **Convert text to lowercase:**
2. **Remove HTML tags:**
3. **Remove website-related patterns:**
4. **Remove punctuation:**
5. **Print success messages:**
6. **Return the cleaned DataFrame:**

In [ ]:

```
import re
import string

def regx(df):
  try:
    df['review']= df['review'].str.lower()
    print("Corpus Lower case successfull")
    # Remove HTML tags from the text in the 'review' column
    df['review'] = df['review'].apply(lambda x: re.sub(r'<[^>]*>', '', x))
    print("Corpus html tags removed successfull")
    # remove http, www log words with sigle char wwwaaaaayyyyy downhillthe etc
```

```
    df['review'] = df['review'].apply(lambda x: re.sub(r'(www\w*|http\w*|\\|\b\w*(\w)\2{
5,}\w*\b)' , '', x))
    print("Corpus website tags removed successfull")
    # Punctuation removal
    df['review'] = df['review'].apply(lambda x: "".join(char for char in x if char not i
n string.punctuation))
    print("Corpus punctuation removed successfull")
    print("Data Cleaning is completed Successfully")
    return df
  except Exception as e:
    raise e
```

In [ ]:

```
df_cleaned = regx(df)
```

```
Corpus Lower case successfull
Corpus html tags removed successfull
Corpus website tags removed successfull
Corpus punctuation removed successfull
Data Cleaning is completed Successfully
```

text_analyzer

- **This function will search if any error of any miss typed data is present in the corpus like html , http, www or any symbols if its present it will show the row details**

In [ ]:

```
# This function will search if any error of any miss typed data is present in the corpus
like html ,
# http, www or any symbols if its present it will show the row details

def text_analyzer(df):
  print("wwwaaaaayyyyy downhillthe httpwwwinvocusne wwwanswerscom showtimeewwwwwww")
  #str1 = input('Enter String to Search in Corpus')

  filtered_df = df[df['review'].str.contains('str1')]

  if filtered_df.empty:
    print('No MATCH FOUND')
  else:
   print("Below Rows found with search Data ",filtered_df)
```

In [ ]:

```
text_analyzer(df_cleaned)
```

```
wwwaaaaayyyyy downhillthe httpwwwinvocusne wwwanswerscom showtimeewwwwwww
No MATCH FOUND
```

NO EMOJI TEXT FOUND

SPELLING CORRECTION

- **pySpellchecker**
- **TextBlob**

There are two ways to do spelling correction

pySpellchecker

In [ ]:

```
!pip install pyspellchecker==0.5.6 -q
```

In [ ]:

```
# THis was taking too much time for all the words so i have applied for 1 review
from spellchecker import SpellChecker
from nltk import sent_tokenize , word_tokenize
# Assuming df['review'] contains the reviews
reviews = df_cleaned['review'][0].split()

spell = SpellChecker()
correct_words = []

for review in reviews:
    words = word_tokenize(review)  # Tokenize the review into words
    corrected_words = [spell.correction(word) for word in words]
    correct_words.append(corrected_words)
```

In [ ]:

```
# Initialize counter for the loop

def spell_verify(reviews, correct_words):
  counter = 0

  while counter < 5:
      for review, corrected_words in zip(reviews, correct_words):
          corrected_review = ''.join(corrected_words)   # Join corrected words back into
a sentence
          if review != corrected_review:
              print(f"Original: {review}\t Corrected: {corrected_review}")
      counter += 1
```

In [ ]:

```
spell_verify(reviews, correct_words)
```

```
Original: wordit  Corrected: word
Original: penitentary  Corrected: penitentiary
Original: awayi  Corrected: away
Original: romanceoz  Corrected: romance
Original: wholl  Corrected: whole
Original: wholl  Corrected: whole
Original: wordit  Corrected: word
Original: penitentary  Corrected: penitentiary
Original: awayi  Corrected: away
Original: romanceoz  Corrected: romance
Original: wholl  Corrected: whole
Original: wholl  Corrected: whole
Original: wordit  Corrected: word
Original: penitentary  Corrected: penitentiary
Original: awayi  Corrected: away
Original: romanceoz  Corrected: romance
Original: wholl  Corrected: whole
Original: wholl  Corrected: whole
Original: wordit  Corrected: word
Original: penitentary  Corrected: penitentiary
Original: awayi  Corrected: away
Original: romanceoz  Corrected: romance
Original: wholl  Corrected: whole
Original: wholl  Corrected: whole
Original: wordit  Corrected: word
Original: penitentary  Corrected: penitentiary
Original: awayi  Corrected: away
Original: romanceoz  Corrected: romance
Original: wholl  Corrected: whole
Original: wholl  Corrected: whole
```

TextBlob

In [ ]:

```
from textblob import TextBlob
from nltk.tokenize import sent_tokenize , word_tokenize# import sent tokenizer
correct_blob=[]
reviews = df_cleaned['review'][5].split()
for word in reviews:
  txt = TextBlob(word).correct()
  correct_blob.append(str(txt))
```

In [ ]:

```
spell_verify(reviews, correct_blob)
```

```
Original: alltime  Corrected: alliee
Original: selflessness  Corrected: helplessness
Original: dedication  Corrected: education
Original: preachy  Corrected: preach
Original: lukas  Corrected: lupus
Original: bette  Corrected: better
Original: kids  Corrected: kiss
Original: grandma  Corrected: grand
Original: dressedup  Corrected: dressed
Original: midgets  Corrected: midges
Original: whats  Corrected: what
Original: theyd  Corrected: they
Original: alltime  Corrected: alliee
Original: selflessness  Corrected: helplessness
Original: dedication  Corrected: education
Original: preachy  Corrected: preach
Original: lukas  Corrected: lupus
Original: bette  Corrected: better
Original: kids  Corrected: kiss
Original: grandma  Corrected: grand
Original: dressedup  Corrected: dressed
Original: midgets  Corrected: midges
Original: whats  Corrected: what
Original: theyd  Corrected: they
Original: alltime  Corrected: alliee
Original: selflessness  Corrected: helplessness
Original: dedication  Corrected: education
Original: preachy  Corrected: preach
Original: lukas  Corrected: lupus
Original: bette  Corrected: better
Original: kids  Corrected: kiss
Original: grandma  Corrected: grand
Original: dressedup  Corrected: dressed
Original: midgets  Corrected: midges
Original: whats  Corrected: what
Original: theyd  Corrected: they
Original: alltime  Corrected: alliee
Original: selflessness  Corrected: helplessness
Original: dedication  Corrected: education
Original: preachy  Corrected: preach
Original: lukas  Corrected: lupus
Original: bette  Corrected: better
Original: kids  Corrected: kiss
Original: grandma  Corrected: grand
Original: dressedup  Corrected: dressed
Original: midgets  Corrected: midges
Original: whats  Corrected: what
Original: theyd  Corrected: they
Original: alltime  Corrected: alliee
Original: selflessness  Corrected: helplessness
Original: dedication  Corrected: education
Original: preachy  Corrected: preach
Original: lukas  Corrected: lupus
Original: bette  Corrected: better
Original: kids  Corrected: kiss
Original: grandma  Corrected: grand
Original: dressedup  Corrected: dressed
```

```
Original: midgets  Corrected: midges
Original: whats  Corrected: what
Original: theyd  Corrected: they
```

Difference between PySpellChecker and TextBlob

PySpellChecker

- **PySpellChecker is a Python library specifically designed for spell checking.**
- **It provides functionalities to correct misspelled words in text.**
- **It uses pre-built language models and dictionaries to identify and correct misspelled words.**
- **PySpellChecker focuses solely on spell checking and does not offer other natural language processing (NLP) functionalities.**

TextBlob

- **TextBlob is a powerful Python library for processing textual data.**
- **It offers a wide range of NLP functionalities, including part-of-speech tagging, noun phrase extraction, sentiment analysis, translation, and spell checking.**
- **TextBlob wraps around NLTK and pattern libraries to provide an intuitive interface for performing various NLP tasks.**
- **In addition to spell checking, TextBlob can handle many other text processing tasks, making it suitable for comprehensive text analysis and manipulation.**

Key Differences

- **PySpellChecker is specialized for spell checking only, while TextBlob offers a broader range of NLP functionalities.**
- **TextBlob provides an easy-to-use interface for spell checking along with other text processing tasks, making it convenient for users who require multiple NLP functionalities in their projects.**
- **PySpellChecker may offer faster spell checking performance for large volumes of text due to its focused scope and efficient algorithms.**
- **TextBlob's spell checking functionality is part of its larger suite of NLP tools, making it suitable for projects requiring comprehensive text analysis and processing.**

Advance PreProcessing

STOPWORDS REMOVAL

In [ ]:

```
def count(text):
  from sklearn.feature_extraction.text import CountVectorizer
  cv= CountVectorizer()
  bow=cv.fit_transform(text)
  print(f"Total Unique Words in Movie Review Dataset : {len(cv.vocabulary_)}")
  # we can see the total word count before applying Stopwords
  total_word_count = text.str.split().apply(len).sum()
  print(f"total words in Corpus is : {total_word_count}")
```

In [ ]:

In [ ]:

```
count(df['review'])
```

```
Total Unique Words in Movie Review Dataset : 221180
total words in Corpus is : 11227623
```

In [ ]:

```
count(df_cleaned['review'])
```

```
Total Unique Words in Movie Review Dataset : 221180
total words in Corpus is : 11227623
```

In [ ]:

```
df_backup_02 = df_cleaned.copy()
```

In [ ]:

```
df_cleaned_copy = df_backup_02.copy()
```

In [ ]:

```
import nltk
from nltk.corpus import stopwords

# Download NLTK stopwords data if not already present
nltk.download('stopwords')

# Get the list of English stopwords
stop_words = set(stopwords.words('english'))

# Function to remove stopwords from text
def remove_stopwords(text):
    if isinstance(text, str):  # Check if text is a string
        tokens = nltk.word_tokenize(text)
        filtered_tokens = [word for word in tokens if word.lower() not in stop_words]
        return ' '.join(filtered_tokens)
    else:
        return text
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [ ]:

```
# Apply remove_stopwords function to the 'review' column
df_cleaned_copy['review'] = df_cleaned_copy['review'].apply(remove_stopwords)
```

In [ ]:

```
df_cleaned_copy['review']
```

Out[ ]:

```
0        one reviewers mentioned watching 1 oz episode ...
1        wonderful little production filming technique ...
2        thought wonderful way spend time hot summer we...
3        basically theres family little boy jake thinks...
4        petter matteis love time money visually stunni...
                               ...
49995    thought movie right good job wasnt creative or...
49996    bad plot bad dialogue bad acting idiotic direc...
49997    catholic taught parochial elementary schools n...
49998    im going disagree previous comment side maltin...
49999    one expects star trek movies high art fans exp...
Name: review, Length: 49582, dtype: object
```

In [ ]:

```
count(df_cleaned_copy['review'])
```

```
Total Unique Words in Movie Review Dataset : 221134
total words in Corpus is : 5946656
```

Stemming Vs Lemmatization

- **in stemming we have Porter Stemming and SnowBall Stemming**
- **But both will remove the suffixes or prefixes from the word and cut the words which may or may not be the actual words after stemming**
- **its a heuristic approch to deal with words**

Lemmatization

- **is the process of reducing words to their base or dictonary form which is called lemma it uses lexical knowledge and context to accuratly reduce words**

Lemmatization

In [ ]:

```
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize

nltk.download('punkt')
nltk.download('wordnet')

def lemmatize(text):
    lemmatizer = WordNetLemmatizer()
    lemma_words = []

    tokens = word_tokenize(text)

    for word in tokens:
        lemma = lemmatizer.lemmatize(word, pos='v')  # Lemmatize the word as a verb
        lemma_words.append(lemma)

    return ' '.join(lemma_words)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

In [ ]:

```
df_cleaned_copy['review'] = df_cleaned_copy['review'].apply(lemmatize)
```

In [ ]:

```
df_cleaned_copy['review']
```

Out[ ]:

```
0        one reviewers mention watch 1 oz episode youll...
1        wonderful little production film technique una...
2        think wonderful way spend time hot summer week...
3        basically theres family little boy jake think ...
4        petter matteis love time money visually stun f...
                               ...
49995    think movie right good job wasnt creative orig...
49996    bad plot bad dialogue bad act idiotic direct a...
49997    catholic teach parochial elementary school nun...
49998    im go disagree previous comment side maltin on...
49999    one expect star trek movies high art fan expec...
Name: review, Length: 49582, dtype: object
```

In [ ]:

```
count(df_cleaned_copy['review'])
```

```
Total Unique Words in Movie Review Dataset : 208007
total words in Corpus is : 5946656
```

In [ ]:

```
df_backup_03 = df_cleaned_copy.copy()
```

Frequency Distribution

In [ ]:

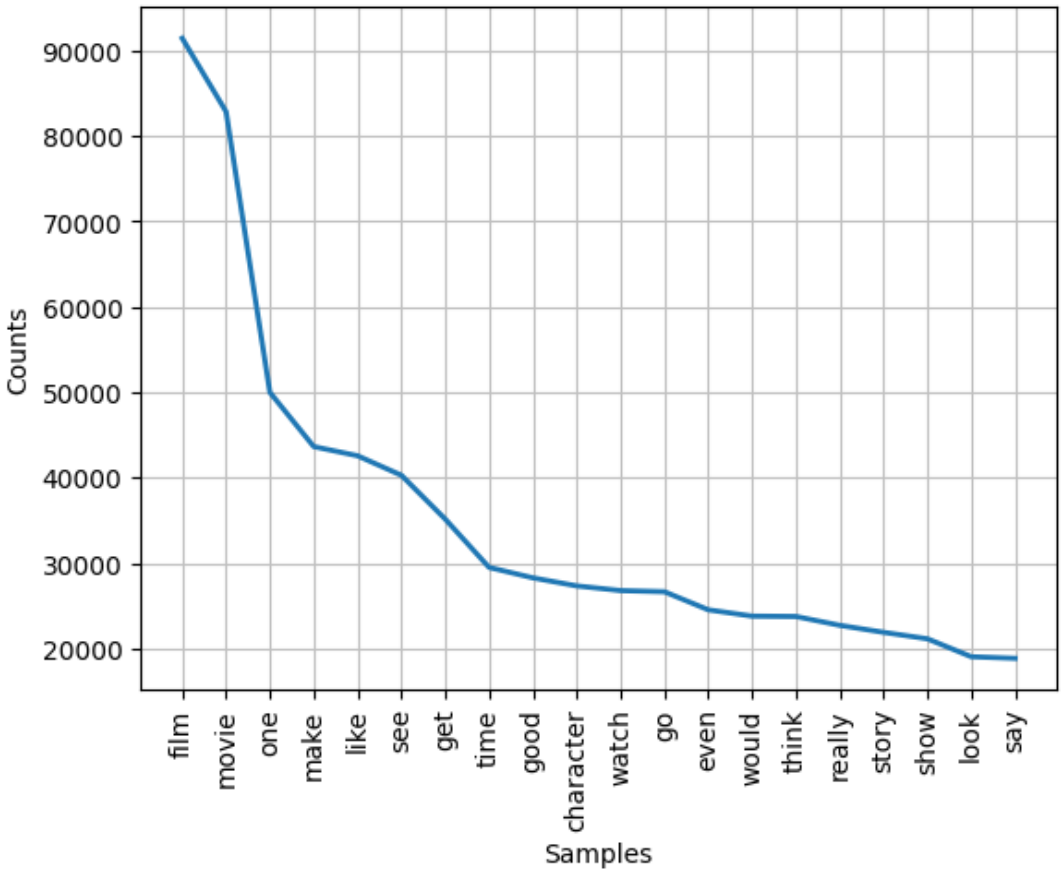
```
import nltk
from nltk.probability import FreqDist
from nltk.tokenize import word_tokenize

def freq_dist(text):
    tokens = word_tokenize(text)
    fd = FreqDist(tokens)
    print('Most Common Words:', fd.most_common(10))
    print(fd)
    fd.plot(20)
```

In [ ]:

```
freq_dist(df_cleaned_copy['review'].str.cat(sep=' '))
```

```
Most Common Words: [('film', 91432), ('movie', 82847), ('one', 49988), ('make', 43644), (
'like', 42570), ('see', 40285), ('get', 35154), ('time', 29505), ('good', 28283), ('chara
cter', 27339)]
<FreqDist with 208850 samples and 5946656 outcomes>
```



Language Detection

In [ ]:

In [ ]:

```
!pip install langdetect -q
```

In [ ]:

```
from langdetect import detect

# Function to detect language for a given text
def detect_language(text):
    try:
        return detect(text)
    except:
        return None  # Return None if language detection fails
```

In [ ]:

```
review_lang = detect_language(df_cleaned_copy['review'][45])
review_lang
```

Out[ ]:

```
'en'
```

1. **Feature Engineering**

   - **Bag of Words**
   - **TF-IDF (Term Frequency-Inverse Document Frequency)**
   - **Word embeddings (Word2Vec, GloVe, etc.)**
   - **n-grams**
   - **Topic modeling (LDA, LSA, etc.)**
   - **Feature selection**

Train Test Split The data

In [ ]:

```
X = df_cleaned_copy['review']
X
```

Out[ ]:

```
0        one reviewers mention watch 1 oz episode youll...
1        wonderful little production film technique una...
2        think wonderful way spend time hot summer week...
3        basically theres family little boy jake think ...
4        petter matteis love time money visually stun f...
                               ...
49995    think movie right good job wasnt creative orig...
49996    bad plot bad dialogue bad act idiotic direct a...
49997    catholic teach parochial elementary school nun...
49998    im go disagree previous comment side maltin on...
49999    one expect star trek movies high art fan expec...
Name: review, Length: 49582, dtype: object
```

LEBAL ENCODING FOR TARGET (y)

In [ ]:

```
df_cleaned_copy.head()
```

Out[ ]:

	review	sentiment
0	**one reviewers mention watch 1 oz episode youll...**	**positive**
1	**wonderful little production film technique una...**	**positive**

	review	sentiment
2	think wonderful way spend time hot summer week...	positive
3	basically theres family little boy jake think ...	negative
4	petter matteis love time money visually stun f...	positive

In [ ]:

```
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
y= encoder.fit_transform(df_cleaned_copy['sentiment'])
y
```

Out[ ]:

```
array([1, 1, 1, ..., 0, 0, 0])
```

Bag of Words

In [ ]:

```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer
cv=CountVectorizer()
x_vector=cv.fit_transform(X)
print(x_vector.shape)
```

```
(49582, 208007)
```

In [ ]:

```
cv.vocabulary_.get(u'algorithm')
```

Out[ ]:

```
8923
```

Train Test split

In [ ]:

```
from sklearn.model_selection import train_test_split
x_train , x_test, y_train, y_test = train_test_split(x_vector, y, test_size =0.20 , rand
om_state=42)
print(f"x_train {x_train.shape}")
print(f"x_test {x_test.shape}")
print(f"y_train {y_train.shape}")
print(f"y_test {y_test.shape}")
```

```
x_train (39665, 208007)
x_test (9917, 208007)
y_train (39665,)
y_test (9917,)
```

In [ ]:

```
# print(x_train.toarray()[:5])
```

MODEL FIT

In [ ]:

```
from sklearn.naive_bayes import MultinomialNB
clf = MultinomialNB().fit(x_train, y_train)
```

MODEL PREDICT

In [ ]:

```
y_pred = clf.predict(x_test)
```

MODEL SCORE

In [ ]:

```
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
```

In [ ]:

```
accuracy_score= accuracy_score(y_test, y_pred)
class_report = classification_report(y_test, y_pred)
confision_matrix = confusion_matrix(y_test , y_pred)
```

In [ ]:

```
accuracy_score
```

Out[ ]:

```
0.8570132096400122
```

In [ ]:

```
print(class_report)
```

```
              precision    recall  f1-score   support

           0       0.85      0.87      0.86      4939
           1       0.87      0.84      0.86      4978

    accuracy                           0.86      9917
   macro avg       0.86      0.86      0.86      9917
weighted avg       0.86      0.86      0.86      9917
```

In [ ]:

```
confision_matrix
```

Out[ ]:

```
array([[4311,  628],
       [ 790, 4188]])
```

Thanks a Lot
