Natural Language Process

Journal

Submitted By
Tagadghar Shailesh Ashok

Roll No: 31031523034
FY MSc CS

Department Of Computer Science
Somaiya Vidyavihar University
SK Somaiya college

INDEX

Practical No. 01

Aim : Applying Tokenization, Stemming and Lemmatization on a given dataset.

Theory :

1. NLTK

The Natural Language Toolkit (NLTK) is a platform used for building Python programs that work with human language data for application in statistical natural language processing (NLP).
It contains text-processing libraries for tokenization, parsing, classification, stemming, tagging and semantic reasoning. It also includes graphical demonstrations and sample data sets as well as accompanied by a cookbook and a book which explains the principles behind the underlying language processing tasks that NLTK supports.

2. NLTK Corpus

The NLTK corpus is a massive dump of all kinds of natural language data sets that are worth taking a look at. Almost all of the files in the NLTK corpus follow the same rules for accessing them by using the NLTK module, but nothing is magical about them.

3. Porter Stemmer

The Porter stemming algorithm is a process for removing suffixes from words in English. Removing suffixes automatically is an operation which is especially useful in the field of information retrieval. In a typical IR environment a document is represented by a vector of words, or terms. Terms with a common stem will usually have similar meanings, for example:

CONNECT
CONNECTED
CONNECT**ING**
CONNECT**ION**
CONNECT**IONS**

Frequently, the performance of an IR system will be improved if term groups such as this are conflated into a single term. This may be done by removal of the various suffixes -ED, -ING, -ION, IONS to leave the single term CONNECT. In addition, the suffix stripping process will reduce the total number of terms in the IR system, and hence reduce the size and complexity of the data in the system, which is always advantageous.

4. WordNet

WordNet® is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked utilizing conceptual-semantic and lexical relations. The resulting network of meaningfully related words and concepts can be navigated with the browser. WordNet is also freely and publicly available for download. WordNet's structure makes it a useful tool for computational linguistics and natural language processing.

WordNet superficially resembles a thesaurus, in that it groups words together based on their meanings. However, there are some important distinctions. First, WordNet interlinks not just word forms—strings of letters—but specific senses of words. As a result, words that are found close to one another in the network are semantically disambiguated. Second, WordNet labels the semantic relations among words, whereas the groupings of words in a thesaurus do not follow any explicit pattern other than meaning similarity.

5. Stemming

Stemming is a technique used in Natural Language Processing (NLP) to reduce words to their base or root form by removing the endings of words. This process helps to consolidate different variations of the same word into a common base form, making it easier for machines to understand the text. However, stemming can sometimes be overly simplistic and aggressive, leading to incorrect results where the stem word loses its actual meaning.

6. Lemmatization

Lemmatization, on the other hand, is a more sophisticated process that also reduces words to their base or root form, known as the lemma. Unlike stemming, lemmatization takes into account the morphological analysis of the

words, considering the context and part of speech in which the word is used. This allows it to handle complex cases better and generally provides more accurate results but at the cost of being more computationally intensive.

Code :

```
import nltk
import nltk.corpus
from nltk.tokenize import word_tokenize
nltk.download('punkt')
nltk.download('wordnet')
from nltk.stem import PorterStemmer
from nltk.probability import FreqDist
from nltk.stem import WordNetLemmatizer

sample="""Natural language processing (NLP) refers to the branch of
computer science—and more specifically,
the branch of artificial intelligence or AI—concerned with giving
computers the ability to understand text
and spoken words in much the same way human beings can."""



#Apply Tokenization by passing the sample text to word tokenizer
Sample_Tokens=word_tokenize(sample)
print("Sample Tokens: \n", Sample_Tokens)

#Getting type of the data
print("\nType of Sample Tokens: \n", type(Sample_Tokens))

#Getting length of the data
print("\nLength of Sample Tokens: \n", len(Sample_Tokens))


#Finding frequency of the words coming in the sample text
FDist= FreqDist(Sample_Tokens)
print("\nFrequency Distribution: \n", FDist)

top5=FDist.most_common(5)
print("\nTop 5 Words: \n", top5)

# Stemming
pst=PorterStemmer()
print("\nStemming of Words: ")
print("Giving: ", pst.stem("Giving"))
print("Buying: ", pst.stem("Buying"))
print("Studying: ", pst.stem("Studying"))
```

```
print("Going: ", pst.stem("Going"))

# Lemmatization
lmt = WordNetLemmatizer()
print("\nLemmatization of Words: ")
print("Geese: ", lmt.lemmatize("geese"))
print("Cacti: ", lmt.lemmatize("cacti"))
```

Output :

```
PS D:\Somaiya\NLP\NLP> & "D:/Python 3.12.0/python.exe" d:/Somaiya/NLP/NLP/Prac1.py
Sample Tokens:
 ['Natural', 'language', 'processing', '(', 'NLP', ')', 'refers', 'to', 'the', 'branch', 'of', '
'of', 'artificial', 'intelligence', 'or', 'AI—concerned', 'with', 'giving', 'computers', 'the',
much', 'the', 'same', 'way', 'human', 'beings', 'can', '.']

Type of Sample Tokens:
 <class 'list'>

Length of Sample Tokens:
 43

Frequency Distribution:
 <FreqDist with 37 samples and 43 outcomes>

Top 5 Words:
 [('the', 4), ('to', 2), ('branch', 2), ('of', 2), ('Natural', 1)]

Stemming of Words:
Giving:  give
Buying:  buy
Studying:  studi
Going:  go

Lemmatization of Words:
Geese:  goose
Cacti:  cactus
```

Practical No. 02

Aim :
1. Implementing POS Tagging.
2. Implementing Named Entity Recognition.

Theory :

1. Parts of Speech :

A PoS tag provides a considerable amount of information about a word and its neighbours. It can be used in various tasks such as sentiment analysis, text to speech conversion, etc.

2. POS Tagging :

Tagging means the classification of tokens into predefined classes. Parts of speech (POS) tagging is the process of marking each word in the given corpus with a suitable token i.e. part of speech based on the context. It is also known as grammatical tagging.

Tag	Description
ADJ	Adjective
ADV	Adposition
ADP	Adverb
AUX	Auxiliary
CCONJ	Coordinating Conjunction
DET	Determiner
INTJ	Interjection
NOUN	Noun
NUM	Numeral
PART	Particle
PRON	Pronoun
PROPN	Proper Noun
PUNCT	Punctuation
SCONJ	Subordinating Conjunction
SYM	Symbol
VERB	Verb
X	Other

Named Entity Recognition :

Named Entity Recognition (NER) is a crucial part of Natural Language Processing (NLP) and Information Retrieval (IR). It involves the process of identifying important named entities in the text such as names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc. and classifying them into predefined categories.

NER can be used in many applications including:

1. Information Extraction : To extract clear, concise information from large amounts of unstructured data.
2. Question Answering : To improve the accuracy of answers to user queries in a question answering system.
3. Machine Translation : To correctly translate named entities without changing their meaning.
4. Content Recommendation : To understand user behavior and recommend relevant content.
5. Sentiment Analysis : To identify the entities being discussed and determine the sentiment towards these entities.

There are different approaches to perform NER:

- Rule-based Approach
- Statistical Approach
- Machine Learning Approach

NER is a challenging task because it's language-dependent and requires a deep understanding of the context. For example, "Jordan" could refer to a person's name or a country depending on the context. Therefore, a good NER system needs to understand the context to accurately classify entities.

Code :

```
# Implementing POS Tagging
import spacy

def pos_tagging(text):
  nlp = spacy.load("en_core_web_sm")
  doc = nlp(text)
  tagged = [(token.text, token.pos_) for token in doc]
  return tagged

# Example Usage
text = """Quickly, the very happy dog barked at the mailman because it
always loves to greet new people."""

tagged_text = pos_tagging(text)
print("Input: ", text)
print("Tagged Text:\n", tagged_text)
```

```
PS D:\Somaiya\NLP\NLP> & "D:/Python 3.12.0/python.exe" d:/Somaiya/NLP/NLP/Prac2.py
Input:  Quickly, the very happy dog barked at the mailman because it
always loves to greet new people.

Tagged Text:
 [('Quickly', 'ADV'), (',', 'PUNCT'), ('the', 'DET'), ('very', 'ADV'), ('happy', 'ADJ'), ('dog', 'NOUN'), ('barked', 'VER
'it', 'PRON'), ('\n', 'SPACE'), ('always', 'ADV'), ('loves', 'VERB'), ('to', 'PART'), ('greet', 'VERB'), ('new', 'ADJ'),
```

```
# Implementing Named Entity Recognition

import spacy
nlp = spacy.load("en_core_web_sm")

text = """Alphabet inc. is the parent company of Google located in
Mountain View,California whose CEO is Sundar Pichai since December
2019."""
doc = nlp(text)

# Tokens
print("Tokens: ")
for token in doc:
  print(token)

 # Entities
print("\nEntities: ")
for ent in doc.ents:
  print(ent.text + " | " + ent.label_ + " | " +
spacy.explain(ent.label_))
```

```
PS D:\Somaiya\NLP\NLP> & "D:/Python 3.12.0/python.exe" d:/Somaiya/
Tokens:
Alphabet
inc
.
is
the
parent
company
of
Google
located
in
Mountain
View
,
California
whose
CEO
is
Sundar
Pichai
since
December
2019
.

Entities:
Alphabet inc. | ORG | Companies, agencies, institutions, etc.
Google | ORG | Companies, agencies, institutions, etc.
Mountain View | GPE | Countries, cities, states
California | GPE | Countries, cities, states
Sundar Pichai | PERSON | People, including fictional
December 2019 | DATE | Absolute or relative dates or periods
```

Practical No. 03

Aim :
1. Working With Spacy.
2. Generating Unigrams, Bigrams, and Trigrams.

Theory :

spaCy :
spaCy which is developed by the software developers Matthew Honnibal and Ines Montani is a free, open-source Python library that provides advanced capabilities to conduct natural language processing (NLP) on large volumes of text at high speed. It helps you build models and production applications that can underpin document analysis, chatbot capabilities, and all other forms of text analysis.
It was developed with the goal of providing industrial-strength performance, while still being easy to use and integrated into existing workflows.
spaCy is built on the latest research and implements state-of-the-art techniques, making it an ideal choice for both beginners and experienced NLP practitioners.

Features :

1. Fast − spaCy is specially designed to be as fast as possible.
2. Accuracy − spaCy implementation of its labelled dependency parser makes it one of the most accurate frameworks (within 1% of the best available) of its kind.
3. Extensile − You can easily use spaCy with other existing tools like TensorFlow, Gensim, scikit-Learn, etc.
4. Deep learning integration − It has Thinc-a deep learning framework, which is designed for NLP tasks.

Matcher :
The Matcher in spaCy is a rule-based matching engine that operates over tokens, similar to regular expressions.

N-Gram :
N-grams are contiguous sequences of n items, such as words or characters, in a given text or speech. The "n" represents the number of items in the sequence, and n-grams are a fundamental concept in natural language processing and text analysis. Unigrams, bigrams, and trigrams are specific cases of n-grams with n equal to 1, 2, and 3, respectively. These sequences provide a way to capture local structures and dependencies within language data, aiding tasks like language modeling, machine translation, and text generation by revealing patterns and relationships between elements in a text or speech.

Unigram :
A unigram is the simplest form, representing a single word or token in a given text. It is the basic building block for analyzing and understanding the frequency and distribution of individual words within a corpus. For example, in the sentence "The quick brown fox," each word ("The," "quick," "brown," "fox") is considered a unigram.

Bigram :
A bigram consists of two consecutive words or tokens. It provides a bit more context than unigrams and helps capture relationships between adjacent words. In the same sentence, "The quick brown fox," examples of bigrams include "The quick," "quick brown," and "brown fox."

Trigram :
A trigram extends the concept further by considering three consecutive words or tokens in a sequence. Trigrams offer even more context and can help in capturing certain patterns of language usage. In the sentence, "The quick brown fox," a trigram example would be "The quick brown" or "quick brown fox.".

Code :

```
# Working With Spacy.
import spacy
from spacy.matcher import Matcher

# Initialize spacy word library
nlp = spacy.load("en_core_web_sm")
doc = nlp("2018 FIFA world cup : France won!!!")

# print token on a specific index
print("Token on index 2: ", doc[2])

# Print token ranging between indices
print("\nToken from index 2 to 5: ", doc[2:5])

# Print POS using Spacy
print("\nPOS Tagging: ")
for token in doc:
  print(token.i, " | ", token.text, " | ", token.pos, " | ",
token.pos_)

# Print the entities in the doc
print("\nEntities: ")
for ent in doc.ents:
  print(ent.text, " | ", ent.label, " | ", ent.label_)

# Getting all available entities in SpaCy
print("\nAll Entities: ")
print(nlp.get_pipe("ner").labels)

# Pattern Matching
pattern = [{"IS_DIGIT": True}, {"LOWER": "fifa"}, {"LOWER": "world"},
{"LOWER": "cup"}, {"IS_PUNCT": True}]
matcher2 = Matcher(nlp.vocab)
matcher2.add("fifa_pattern", [pattern])
matcher2 = matcher2(doc)
print()

for match_id, start, end in matcher2:
  print("Matched Pattern : ",doc[start:end])
```

```
PS D:\Somaiya\NLP\NLP> & "D:/Python 3.12.0/python.exe" d:/Somaiya/NLP/NLP/Prac
Token on index 2:  world

Token from index 2 to 5:  world cup :

POS Tagging:
0  |  2018  |  93  |  NUM
1  |  FIFA  |  96  |  PROPN
2  |  world  |  92  |  NOUN
3  |  cup  |  92  |  NOUN
4  |  :  |  97  |  PUNCT
5  |  France  |  96  |  PROPN
6  |  won  |  100  |  VERB
7  |  !  |  97  |  PUNCT
8  |  !  |  97  |  PUNCT
9  |  !  |  97  |  PUNCT

Entities:
2018  |  391  |  DATE
France  |  384  |  GPE

All Entities:
('CARDINAL', 'DATE', 'EVENT', 'FAC', 'GPE', 'LANGUAGE', 'LAW', 'LOC', 'MONEY',
T')

Matched Pattern :  2018 FIFA world cup :
```

```python
# Generating Unigrams, Bigrams, and Trigrams.
from nltk import ngrams

sentence = "Tony gave two $ to Peter, Bruce gave 500 € to Steve"
print("Sentence : ",sentence)

unigrams = ngrams(sentence.split(), 1)
print("\nUnigrams: ")
for grams in unigrams:
  print(grams)

bigrams = ngrams(sentence.split(), 2)
print("\nBigrams: ")
for grams in bigrams:
  print(grams)

trigrams = ngrams(sentence.split(), 3)
print("\nTrigrams: ")
for grams in trigrams:
  print(grams)
```

```
PS D:\Somaiya\NLP\NLP> & "D:/Python 3.12.0/python.exe" d:/Somaiya/
Sentence :  Tony gave two $ to Peter, Bruce gave 500 € to Steve

Unigrams:
('Tony',)
('gave',)
('two',)
('$',)
('to',)
('Peter,',)
('Bruce',)
('gave',)
('500',)
('€',)
('to',)
('Steve',)
```

```
Bigrams:
('Tony', 'gave')
('gave', 'two')
('two', '$')
('$', 'to')
('to', 'Peter,')
('Peter,', 'Bruce')
('Bruce', 'gave')
('gave', '500')
('500', '€')
('€', 'to')
('to', 'Steve')

Trigrams:
('Tony', 'gave', 'two')
('gave', 'two', '$')
('two', '$', 'to')
('$', 'to', 'Peter,')
('to', 'Peter,', 'Bruce')
('Peter,', 'Bruce', 'gave')
('Bruce', 'gave', '500')
('gave', '500', '€')
('500', '€', 'to')
('€', 'to', 'Steve')
```

Practical No. 04

Aim : Sentiment Analysis on Movie Reviews using Naive Bayes Classifier

Theory :

Sentiment Analysis :

Sentiment analysis is the process of analyzing digital text to determine if the emotional tone of the message is positive, negative, or neutral. Today, companies have large volumes of text data like emails, customer support chat transcripts, social media comments, and reviews. Sentiment analysis tools can scan this text to automatically determine the author's attitude towards a topic. Companies use the insights from sentiment analysis to improve customer service and increase brand reputation.

Naive Bayes Classifier :

Naive Bayes classifier is a probabilistic machine learning model based on Bayes' theorem. It assumes independence between features and calculates the probability of a given input belonging to a particular class. It's widely used in text classification, spam filtering, and recommendation systems.

Naive Bayes Algorithm :

It is a classification technique based on Bayes' Theorem with an independence assumption among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.

The Naïve Bayes classifier is a popular supervised machine learning algorithm used for classification tasks such as text classification. It belongs to the family of generative learning algorithms, which means that it models the distribution of inputs for a given class or category. This approach is based on the assumption that the features of the input data are conditionally independent given the class, allowing the algorithm to make predictions quickly and accurately.

Code :

```
# pip install scikit-learn

import pandas as pd
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix
nltk.download("stopwords")

dataset = [["I liked the movie", "positive"],
["It's a good movie. Nice story", "positive"],
["Hero's acting is bad but the heroine looks good. Overall nice
movie.", "positive"],
["Nice songs. But sadly the ending is boring", "negative"],
["Sad and boring movie", "negative"]]

dataset = pd.DataFrame(dataset)
dataset.columns = ["Text", "Reviews"]

stop_words = set(stopwords.words('english'))

corpus = []
for i in range(0, 5):
  text = re.sub('[^a-zA-Z]', ' ', dataset['Text'][i])
  text = text.lower()
  text = text.split()
  ps = PorterStemmer()
  text = [ps.stem(word) for word in text if not word in stop_words]
  text = ' '.join(text)
  corpus.append(text)

print("Corpus: \n", corpus)

#Creating bag of words model
cv = CountVectorizer(max_features = 1500)
X = cv.fit_transform(corpus).toarray()
y = dataset.iloc[: , 1].values

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=0)
classifier = GaussianNB()
```

```
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix: \n", cm)
```

Output :

```
PS D:\Somaiya\NLP\NLP> & "D:/Python 3.12.0/python.exe" d:/Somaiya/NLP/NLP/Pac4.py
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\Shailesh\AppData\Roaming\nltk_data...
[nltk_data]   Unzipping corpora\stopwords.zip.
Corpus:
 ['like movi', 'good movi nice stori', 'hero act bad heroin look good overal nice movi', 'nice song sadli end bore', 'sad bore movi']

Confusion Matrix:
 [[0 0]
 [1 1]]
```

Practical No. 05

Aim : Implementing CYK Algorithm.

Theory :

CYK Algorithm :

The CYK (Cocke-Younger-Kasami) algorithm is a dynamic programming algorithm used for parsing context-free grammars, particularly in the context of parsing natural language sentences. It is named after its inventors: John Cocke, Daniel Younger, and Tadao Kasami.

The CYK algorithm is based on a bottom-up approach to parse a string and determine whether it belongs to a given context-free language defined by a grammar. It works by constructing a parse table that represents all possible ways to derive substrings of the input string from non-terminals in the grammar. The algorithm then fills in this table in a bottom-up manner, combining smaller substrings to build larger ones until the entire string is parsed.

The efficiency of the CYK algorithm is notable for its time complexity of O(n^3), where n is the length of the input string. This makes it a practical choice for certain parsing tasks, especially in applications like natural language processing and syntax analysis.

The CYK algorithm is particularly useful for parsing strings based on context-free grammars in Chomsky Normal Form (CNF). CNF is a restricted form of context-free grammars where each production rule has either two non-terminals or a terminal on the right-hand side.

Key steps of the CYK algorithm include:

1. Initialization : Fill in the table for substrings of length 1 based on the terminal symbols in the grammar.
2. Iteration : Iterate over all possible substrings of increasing lengths, combining smaller substrings according to the production rules of the grammar until the entire string is covered.
3. Parsing Decision : The top cell of the table represents the entire input string and indicates whether it can be derived from the start symbol of the grammar. If the start symbol is present in this cell, the string is accepted; otherwise, it is not.

Code :

```
def is_in_cartesian_prod(x, y, r):
    return r in [i + j for i in x.split(',') for j in y.split(',')]

def accept_cyk(w, G, S):
    if w == ['ε']:  # Correctly handle the case when the input string
is empty
        return 'ε' in G[S]

    n = len(w)
    DP_table = [[''] * n for _ in range(n)]

    # Initialise DP Table with rules of the form A → a
    for i in range(n):
        for lhs in G.keys():
            for rhs in G[lhs]:
                if w[i] == rhs:
                    DP_table[i][i] = lhs if not DP_table[i][i] else
DP_table[i][i] + ',' + lhs

    # Fill in the DP Table for subsequences of length 2 to n
    for l in range(2, n + 1):
        for i in range(n - l + 1):
            j = i + l - 1
            for k in range(i, j):
                for lhs in G.keys():
                    for rhs in G[lhs]:
                        if is_in_cartesian_prod(DP_table[i][k],
DP_table[k + 1][j], rhs):
                            DP_table[i][j] = lhs if not DP_table[i][j]
else DP_table[i][j] + ',' + lhs

    # Check if Start Symbol is in the DP_Table[0][n-1]
    return S in DP_table[0][n - 1]

# Test Cases
if __name__ == "__main__":
    grammar = {
        'NP': ['Det', 'Nom'],
        'Nom': ['AP', 'Nom'],
        'AP': ['Adv', 'A'],
        'Det': ['a', 'an'],
        'Adv': ['very', 'extremely'],
        'A': ['heavy', 'orange', 'tall'],
        'N': ['book', 'man']
    }
```

```
input_string = "A very heavy book"
input_string1 = "a tall man"

start = 'NP'

if accept_cyk(input_string.split(), grammar, start):
    print(f"'{input_string}' is accepted by the grammar")
else:
    print(f"'{input_string}' is not accepted by the grammar")

if accept_cyk(input_string1.split(), grammar, start):
    print(f"'{input_string1}' is accepted by the grammar")
else:
    print(f"'{input_string1}' is not accepted by the grammar")
```

Output :

```
PS D:\Somaiya\NLP\NLP> & "D:/Python 3.12.0/python.exe" d:/Somaiya/NLP/NLP/Prac5.py
'A very heavy book' is not accepted by the grammar
'a tall man' is accepted by the grammar
```

Practical No. 06

Aim : Calculating Maximum Likelihood Estimation.

Theory :

Maximum Likelihood Estimation :

Maximum Likelihood Estimation (MLE) is a statistical method employed to estimate the parameters of a probability distribution or statistical model that best characterize a given dataset.

The core concept of MLE involves identifying the parameter values that maximize the likelihood of the observed data, assuming that the data are generated by the specified distribution or model. The likelihood function, which represents the probability of observing the given dataset under certain parameter values, is central to MLE and is optimized to obtain the most probable set of parameters.

Reuters :

The Reuters Corpus contains 10,788 news documents totaling 1.3 million words. The documents have been classified into 90 topics, and grouped into two sets, called "training" and "test"; thus, the text with fileid 'test/14826' is a document drawn from the test set. This split is for training and testing algorithms that automatically detect the topic of a document.

The Reuters Corpus, often referred to as the Reuters Corpus of News Documents, is a widely used collection of text documents that serves as a benchmark dataset in the field of natural language processing (NLP) and information retrieval.

The corpus consists of news articles published by the Reuters news agency, covering a diverse range of topics such as business, finance, technology, and world events. Originally created for research purposes, the Reuters Corpus is commonly employed for tasks like text classification, document clustering, and information retrieval.

It provides a standardized and representative set of documents for evaluating the performance of NLP algorithms and models.

Researchers and practitioners use the Reuters Corpus to develop and test various text processing and analysis techniques, contributing to advancements in the field of computational linguistics and information science.

Code :

```
import nltk
from nltk import FreqDist
from nltk.corpus import reuters
from nltk.tokenize import word_tokenize
#download NLTK resources if not already present

nltk.download('reuters')
nltk.download('punkt')

#Load the Reuters corpus
corpus = reuters.raw()

#Tokenize the corpus into words
tokens = word_tokenize(corpus)

#Calculate the frequency distribution of words
word_freq = FreqDist(tokens)

#Total number of words in the corpus
total_words = len(tokens)

#Calculate the MLE probabilities for each word
mle_probabilities = {word : count / total_words for word, count in
word_freq.items()}

#Example : Display the MLE probability of a specific word
example_word = "economy"
print(f"MLE Probability of '{example_word}' :
{mle_probabilities.get(example_word, 0)}")
```

Output :

```
PS D:\Somaiya\NLP\NLP> & "D:/Python 3.12.0/python.exe" d:/Somaiya/NLP/NLP/Prac6.py
MLE Probability of 'economy' : 0.00036364481912707155
```

Practical No. 07

Aim : Naive Bayes Approach to Review - Sentiment Analysis

Theory :

Sentiment Analysis :

Sentiment analysis, also known as opinion mining, is the process of computationally identifying and categorizing opinions expressed in text data to determine whether the sentiment conveyed is positive, negative, or neutral. It involves analyzing subjective information, such as opinions, attitudes, and emotions, expressed in written language, such as reviews, social media posts, news articles, or customer feedback.

The primary goal of sentiment analysis is to understand the overall sentiment or attitude of individuals towards specific topics, products, services, events, or entities. By automatically extracting sentiment from large volumes of text data, sentiment analysis enables organizations to gain valuable insights into public opinion, customer satisfaction, market trends, and brand perception.

Libraries Used:

1. **nltk** : Natural Language Toolkit, used for natural language processing tasks.
2. **random** : Library for random number generation and sampling.
3. **SklearnClassifier** : Adapter class to use scikit-learn classifiers with NLTK.
4. **pickle** : Serialization and deserialization library in Python.
5. **MultinomialNB, BernoulliNB** : Naive Bayes classifiers from scikit-learn.
6. **word_tokenize** : Tokenization function from NLTK.
7. **re** : Regular expression library for string operations.
8. **rs** : Library for interacting with the operating system.
9. **pandas** : Data manipulation and analysis library.
10. **names** : Corpus of names, used for training.
11. **stopwords** : Collection of common stop words.
12. **matplotlib**.**pyplot** : Plotting library for data visualization.
13. **train_test_split** : Function for splitting datasets for training and testing14.
14. **metrics** : Module from scikit-learn for evaluating models.

Code :

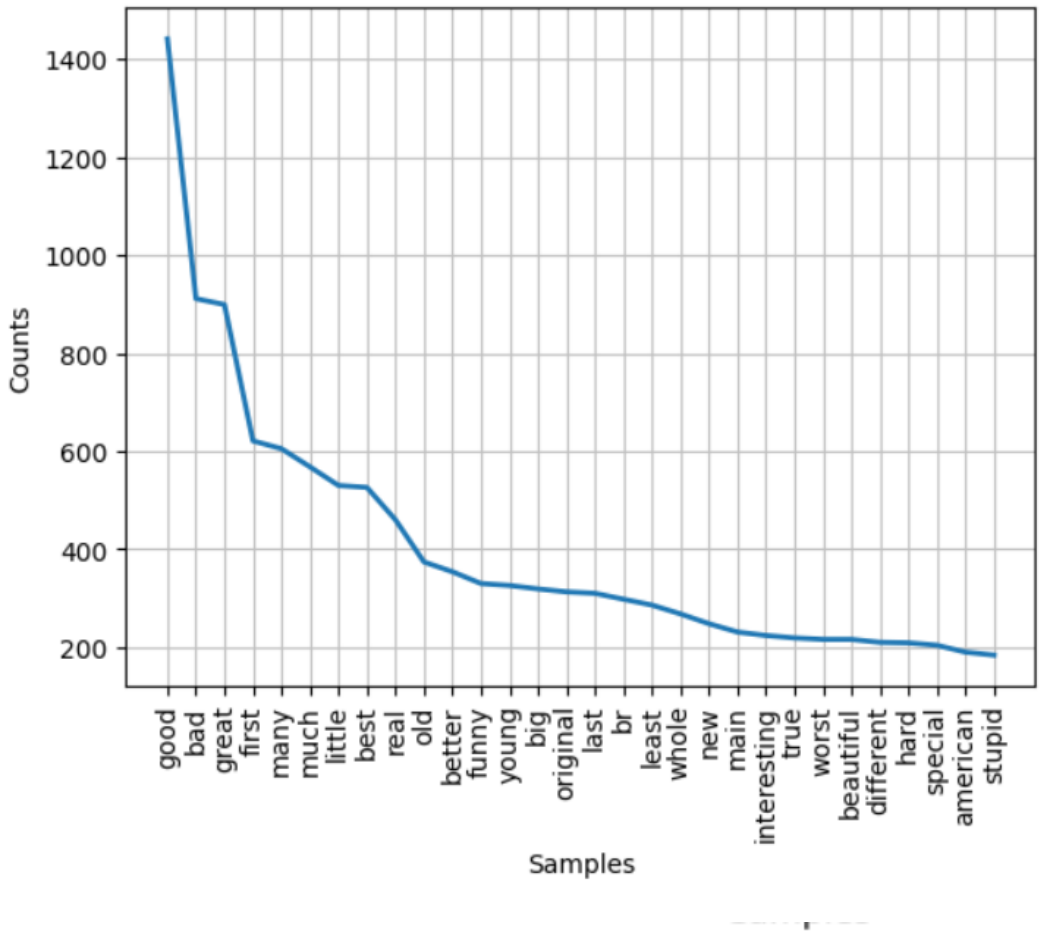
```
import nltk
import random
from nltk.classify.scikitlearn import SklearnClassifier
import pickle
from sklearn.naive_bayes import MultinomialNB, BernoulliNB
from nltk.tokenize import word_tokenize
import re
import os
import pandas as pd
from nltk.corpus import names, stopwords
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import metrics

# Download necessary NLTK resources
nltk.download("names")
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

# Importing the CSV file with the reviews
data = pd.read_csv('/content/File/dataset.csv')
dataset = data[:2499]

# Initialize lists to store words and documents
all_words = []
documents = []

# Define stopwords and allowed word types
stop_words = set(stopwords.words('english'))
allowed_word_types = ["J"]

# Define a function to preprocess each review
def preprocess_review(review):
    # Remove punctuations
    cleaned = re.sub(r'[^(a-zA-Z)\s]', '', review)
    # Tokenize
    tokenized = word_tokenize(cleaned)
    # Part-of-speech tagging
    pos_tags = nltk.pos_tag(tokenized)
    # Remove stopwords and extract adjectives
    adjectives = [w.lower() for w, pos in pos_tags if w.lower() not in
stop_words and pos[0].lower() == 'j']
    return adjectives

# Process reviews
for index, row in dataset.iterrows():
```

```
    adjectives = preprocess_review(row['review'])
    documents.append((adjectives, row['sentiment']))
    all_words.extend(adjectives)

# Create frequency distribution of words
all_words = nltk.FreqDist(all_words)
print("All words:", all_words)

# Plot the most common words
if all_words:
    all_words.plot(30, cumulative=False)
    plt.show()
else:
    print("No words to plot.")

# Extract the most common 1000 words as features
word_features = list(all_words.keys())

# Define a function to create feature sets
def find_features(document):
    words = set(document)
    features = {}
    for w in word_features:
        features[w] = (w in words)
    return features if features else {'contains_no_features': True}

# Create feature sets
featuresets = [(find_features(rev), category) for (rev, category) in
documents]

# Shuffle the feature sets
random.shuffle(featuresets)

# Split the data into training and testing sets
train_set, test_set = train_test_split(featuresets, test_size=0.2,
random_state=42)

# Train and test Naive Bayes classifier
classifier = nltk.NaiveBayesClassifier.train(train_set)
print("Naive Bayes Classifier accuracy:",
(nltk.classify.accuracy(classifier, test_set))*100)

# Train and test Multinomial Naive Bayes classifier
MNB_clf = SklearnClassifier(MultinomialNB())
mnb_cls = MNB_clf.train(train_set)
print("Multinomial Naive Bayes Classifier accuracy:",
(nltk.classify.accuracy(mnb_cls, test_set))*100)
```

```
# Train and test Bernoulli Naive Bayes classifier
BNB_clf = SklearnClassifier(BernoulliNB())
bnb_cls = BNB_clf.train(train_set)
print("Bernoulli Naive Bayes Classifier accuracy:",
(nltk.classify.accuracy(bnb_cls, test_set))*100)
```

Output :

```
All words: <FreqDist with 6805 samples and 46154 outcomes>
```




```
Naive Bayes Classifier accuracy: 79.60000000000001
Multinomial Naive Bayes Classifier accuracy: 78.8
Bernoulli Naive Bayes Classifier accuracy: 80.60000000000001
```