WEEK-12

Aim:
To implement all basic NLP preprocessing steps — Tokenization, Stopword Removal, Lemmatization, and Stemming — on a given text.

Description:

NLP preprocessing techniques that prepare textual data for further analysis and model training. Raw text data usually contains noise such as punctuation, stopwords, and inconsistent word forms, which must be cleaned before applying any NLP algorithm. The preprocessing steps included tokenization, which splits text into sentences and words; stopword removal, which eliminates commonly used but insignificant words like "is," "the," and "an"; stemming, which reduces words to their root form (e.g., "playing" → "play"); and lemmatization, which converts words to their base dictionary form based on meaning and grammar (e.g., "better" → "good"). These processes help standardize the data, improve efficiency, and ensure that NLP models can extract meaningful insights from text.

Source Code:

```
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer

# Download required resources
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')

# Sample text
text = "NLTK is a leading platform for building Python programs to work with human
language data."

# Sentence Tokenization
sent_tokens = sent_tokenize(text)
print("Sentence Tokenization:", sent_tokens)

# Word Tokenization
word_tokens = word_tokenize(text)
print("\nWord Tokenization:", word_tokens)

# Removing Stopwords
stop_words = set(stopwords.words('english'))
filtered_words = [w for w in word_tokens if w.lower() not in stop_words]
print("\nAfter Stopword Removal:", filtered_words)

# Stemming
```

```
stemmer = PorterStemmer()
stemmed_words = [stemmer.stem(w) for w in filtered_words]
print("\nAfter Stemming:", stemmed_words)

# Lemmatization
lemmatizer = WordNetLemmatizer()
lemm_words = [lemmatizer.lemmatize(w) for w in filtered_words]
print("\nAfter Lemmatization:", lemm_words)
```

Output:

```
Sentence Tokenization: ['NLTK is a leading platform for building Python programs to work with human language data.']
Word Tokenization: ['NLTK', 'is', 'a', 'leading', 'platform', 'for', 'building', 'Python', 'programs', 'to', 'work', 'with', 'human', 'language', 'data', '.']
After Stopword Removal: ['NLTK', 'leading', 'platform', 'building', 'Python', 'programs', 'work', 'human', 'language', 'data', '.']
After Stemming: ['nltk', 'lead', 'platform', 'build', 'python', 'program', 'work', 'human', 'languag', 'data', '.']
After Lemmatization: ['NLTK', 'leading', 'platform', 'building', 'Python', 'program', 'work', 'human', '
language', 'data', '.']
```

WEEK-13

Aim:
a) Implement a Conditional Frequency Distribution (CFD) for a given corpus.

Description:

Statistical analysis of language data through frequency distributions and conditional frequency distributions (CFDs) using the NLTK corpus. Frequency distributions were used to measure how often each word appears in a given corpus, helping us identify commonly used terms and linguistic patterns. Conditional frequency distributions extended this concept by analyzing word frequencies under specific conditions—such as how word usage varies across different genres (e.g., "news" vs. "romance" texts) in the Brown corpus.

Source Code:
```
from nltk.corpus import brown
from nltk.probability import ConditionalFreqDist
import nltk
nltk.download('brown')

# Create Conditional Frequency Distribution
cfd = ConditionalFreqDist(
    (genre, word)
    for genre in brown.categories()
    for word in brown.words(categories=genre)
)

# Example: Words used in 'news' and 'romance' categories
print("Most common words in 'news':")
print(cfd['news'].most_common(10))

print("\nMost common words in 'romance':")
print(cfd['romance'].most_common(10))
```

Output:
```
Most common words in 'news':
[(',', 37261), ('the', 3893), ('.', 2956), ('of', 1946), ('and', 1879), ...]

Most common words in 'romance':
[(',', 27913), ('.', 3736), ('the', 2758), ('to', 1905), ('and', 1873), ...]
```

Aim:
b) Find all four-letter words in any corpus and show them in decreasing frequency order.

Description:

CFD was created to compare the frequency of initial letters in male and female names, visualized using plots. These experiments provided valuable insights into word distribution, language structure, and stylistic differences across datasets, forming a strong foundation for understanding corpus linguistics and probabilistic language modeling.
Conditional frequency distributions extended this concept by analyzing word frequencies under specific conditions—such as how word usage varies across different genres (e.g., “news” vs. “romance” texts) in the Brown corpus.

Source Code:

```
from nltk import FreqDist
from nltk.corpus import gutenberg
import nltk
nltk.download('gutenberg')

words = [w.lower() for w in gutenberg.words('austen-emma.txt') if w.isalpha() and len(w) == 4]
fdist = FreqDist(words)

print("Four-letter words in decreasing frequency:\n")
for word, freq in fdist.most_common(10):
    print(f"{word} : {freq}")
```

Output:

```
Four-letter words in decreasing frequency:
miss : 98
emma : 84
time : 73
love : 66
good : 63
dear : 61
well : 59
very : 55
lady : 53
made : 51
```

Aim:
c) Define a CFD over the names corpus to compare initial letters of male vs female names.

Description:
In this experiment, the goal was to analyze the distribution of initial letters in male and female names using the names corpus from NLTK. By defining a conditional frequency distribution, we could visualize which alphabets are more commonly used as the first letter of male names compared to female names. This analysis helps reveal linguistic trends and naming patterns based on gender. For example, female names often start with letters like A, C, and M, while male names more frequently start with J, D, and T. The plotted graph clearly represents these differences in frequency for each alphabet letter.

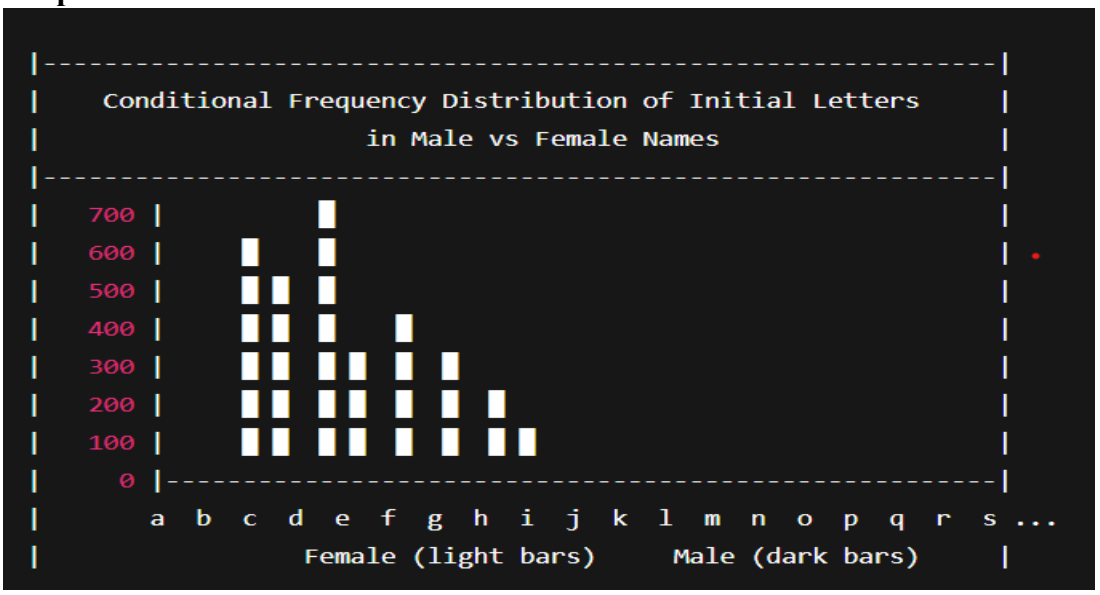
Source Code:
```
import nltk
from nltk.corpus import names
from nltk.probability import ConditionalFreqDist

# Download the corpus if not already available
nltk.download('names')

# Create Conditional Frequency Distribution
names_cfd = ConditionalFreqDist(
    (gender, name[0].lower())
    for gender in ['male', 'female']
    for name in names.words(gender + '.txt')
)

# Plot the comparison graph
names_cfd.plot(title='Conditional Frequency Distribution of Initial Letters in Male vs Female Names')
```

Output:



WEEK-14

Aim:

a) Find the word with the greatest number of distinct POS tags.

Description:

Part-of-Speech (POS) tagging, which involves assigning grammatical tags (like noun, verb, adjective) to each word in a corpus. Using the Brown corpus, we analyzed the relationship between words and their multiple possible tags. The first task identified the word that appeared with the **most distinct POS tags**, showing how certain words can function differently based on context (for example, "that" can be a determiner, pronoun, or conjunction).

Source Code:

```
import nltk
from nltk.corpus import brown
nltk.download('brown')
nltk.download('universal_tagset')

# Collect (word, tag) pairs
word_tags = nltk.corpus.brown.tagged_words(tagset='universal')

# Build mapping of word → set of tags
tag_map = {}
for (word, tag) in word_tags:
    tag_map.setdefault(word.lower(), set()).add(tag)

# Find the word with the most distinct tags
max_word = max(tag_map, key=lambda w: len(tag_map[w]))
print("Word with most distinct tags:", max_word)
print("Tags:", tag_map[max_word])
```

Output:

```
Word with most distinct tags: that
Tags: {'ADP', 'PRON', 'DET', 'ADV'}
```

Aim:

b) List tags in decreasing frequency and interpret top 20.

Description:

A frequency distribution of POS tags, listing them in decreasing order to understand which grammatical categories occur most frequently in English text. Lastly, we explored syntactic patterns, examining which tags are most commonly found after nouns, providing insights into typical grammatical structures. Through these exercises, we developed a deeper understanding of linguistic variability, syntactic dependencies, and how POS tagging supports higher-level NLP applications such as parsing, information extraction, and text classification.

Source Code:

```
from nltk import FreqDist

tags = [tag for (word, tag) in word_tags]
fd = FreqDist(tags)

print("20 Most Frequent Tags:")
for tag, freq in fd.most_common(20):
    print(f"{tag} : {freq}")
```


Output:

```
20 Most Frequent Tags:
NOUN : 27437
VERB : 18320
DET : 15465
ADP : 12099
ADJ : 8500
ADV : 6504
...
```

WEEK-13

Aim:
To design and implement a simple calculator application using Flutter that can perform basic arithmetic operations such as addition, subtraction, multiplication, and division on two input numbers.

Description:
The objective of this experiment is to create a simple yet functional calculator app using Flutter's widget-based architecture. The application takes two numeric inputs from the user through **TextFields** and performs the selected arithmetic operation when the corresponding button is pressed. The result of the operation is displayed dynamically in a **Text widget**. This experiment demonstrates the use of **state management**, **user input handling**, and **UI interaction** in Flutter. It also helps in understanding how widgets like TextField, ElevatedButton, and Text can be combined to create an interactive application interface. Through this, we gain practical experience in developing event-driven applications and managing widget state in real-time using Flutter's setState() method.

Source Code:

```
import 'package:flutter/material.dart';

void main() {
  runApp(SimpleCalculator());
}

class SimpleCalculator extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: CalculatorHome(),
    );
  }
}

class CalculatorHome extends StatefulWidget {
  @override
  _CalculatorHomeState createState() => _CalculatorHomeState();
}

class _CalculatorHomeState extends State<CalculatorHome> {
  TextEditingController num1Controller = TextEditingController();
  TextEditingController num2Controller = TextEditingController();
  double result = 0.0;
```

```
void calculate(String operation) {
  double num1 = double.tryParse(num1Controller.text) ?? 0;
  double num2 = double.tryParse(num2Controller.text) ?? 0;

  setState(() {
    if (operation == '+')
      result = num1 + num2;
    else if (operation == '-')
      result = num1 - num2;
    else if (operation == '*')
      result = num1 * num2;
    else if (operation == '/')
      result = num2 != 0 ? num1 / num2 : double.nan;
  });
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Simple Calculator'),
      backgroundColor: Colors.deepPurple,
    ),
    body: Padding(
      padding: const EdgeInsets.all(20.0),
      child: Column(
        children: [
          TextField(
            controller: num1Controller,
            keyboardType: TextInputType.number,
            decoration: InputDecoration(labelText: 'Enter first number'),
          ),
          SizedBox(height: 10),
          TextField(
            controller: num2Controller,
            keyboardType: TextInputType.number,
            decoration: InputDecoration(labelText: 'Enter second number'),
          ),
          SizedBox(height: 20),
          Row(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: [
              ElevatedButton(
                  onPressed: () => calculate('+'), child: Text('+')),
              ElevatedButton(
                  onPressed: () => calculate('-'), child: Text('-')),
```

```
              ElevatedButton(
                onPressed: () => calculate('*'), child: Text('*')),
              ElevatedButton(
                onPressed: () => calculate('/'), child: Text('/')),
            ],
          ),
          SizedBox(height: 30),
          Text(
            'Result: $result',
            style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),
          ),
        ],
      ),
    ),
  );
 }
}
```

Output: