INDEX

EX.NO	DATE	TOPIC	PG.NO	SIGN
1		Tokenize text & compute word frequencies		
2		Conditional Frequency Distribution (CFD)		
3		Access & process corpus data (NLTK Gutenberg / Brown)		
4		Regular expressions: extract dates and emails		
5		Unicode handling & normalization		
6		Implement a POS tagging program using NLTK's built-in taggers (e.g., Unigram, Bigram)		
7		Naive Bayes classifier for sentiment (sklearn)		
8		Dependency parsing (spaCy)		
9		Named Entity Recognition (NER) using spaCy		
10		Extract relationships using chunking (NLTK RegexpParser)		
11		Simple coreference-ish heuristic (pronoun linking)		
12		Scripts to manage linguistic data (download corpora & preprocess)		
13		Integrate with external API (example: send text to remote NLP API)		

EX.NO.1	Tokenize text & compute word frequencies

Aim:

To write a NLP program to Tokenize input text and compute word frequency counts.

Procedure:

1. Start the program.
2. Import the required modules: nltk and collections.Counter.
3. Read or define a sample text string.
4. Convert all characters in the text to lowercase.
5. Tokenize the text into words using nltk.word_tokenize().
6. Remove punctuation and keep only alphabetic words.
7. Pass the tokenized words into Counter() to count word occurrences.
8. Store the frequency of each word in a dictionary-like structure.
9. Sort the words based on frequency.
10. Display the most common words and their counts.
11. Optionally, visualize using a bar chart (optional step).
12. End the program.

Program:

```
import nltk
from collections import Counter
nltk.download('punkt', quiet=True)
nltk.download('punkt_tab', quiet=True)
text = "This is a sample text. This text is simple and simple."
tokens = [t.lower() for t in nltk.word_tokenize(text) if t.isalpha()]
freq = Counter(tokens)
print("Top words:", freq.most_common(5))
```

output:

Top words: [('simple', 2), ('this', 2), ('text', 2), ('is', 2), ('a', 1)]

RESULT:

Thus the Tokenization & Word Frequency program successfully splits the text into words and counts how many times each word appears successfully.

EX.NO .2	**Conditional Frequency Distribution (CFD)**

Aim:

To Create and use a conditional frequency distribution (word counts per document/label).

Procedure :

1. Start the program.
2. Import nltk and ConditionalFreqDist.
3. Create multiple text samples, each labeled by a category (e.g., sports, tech).
4. Tokenize each text into words.
5. Clean tokens by converting to lowercase and removing punctuation.
6. Create pairs in the form (category, word).
7. Pass the list of pairs to ConditionalFreqDist().
8. For each category, compute the frequency of each word.
9. Access the distribution using the condition name.
10. Display the most frequent words for each category.
11. Optionally, plot the CFD for visualization.
12. End the program.

Program:

```
import nltk
from nltk.probability import ConditionalFreqDist
nltk.download('punkt', quiet=True)
docs = {'sport': "Football match and team", 'tech': "Computer science and AI"}
pairs = []
for label, text in docs.items():
    tokens = [t.lower() for t in nltk.word_tokenize(text) if t.isalpha()]
    for tok in tokens:
        pairs.append((label, tok))
cfd = ConditionalFreqDist(pairs)
for label in cfd.conditions():
    print(label, cfd[label].most_common(3))
```

output:

sport [('and', 1), ('football', 1), ('match', 1)]

tech [('and', 1), ('computer', 1), ('science', 1)]

Result:

Thus the Conditional Frequency Distribution program displays word frequencies under different conditions or categories successfully.

EX.NO.3	Access & process corpus data (NLTK Gutenberg / Brown)

Aim:

To write a NLP program to Load a small text from the Gutenberg corpus and show basic stats.

Procedure:

1. Start the program.
2. Import nltk and collections.Counter.
3. Download the required corpus (e.g., Gutenberg) using nltk.download().
4. Load one of the text files, such as 'austen-emma.txt'.
5. Tokenize the corpus text into words.
6. Convert all tokens to lowercase for uniformity.
7. Remove punctuation and non-alphabetic words.
8. Count the frequency of each word using Counter().
9. Count the total number of sentences using sent_tokenize().
10. Print the first few sentences as output.
11. Display the total word count and top 5 frequent words.
12. End the program.

Program:

```
import nltk
from collections import Counter
nltk.download('gutenberg', quiet=True); nltk.download('punkt', quiet=True)
from nltk.corpus import gutenberg
raw = gutenberg.raw('austen-emma.txt')
sents = nltk.sent_tokenize(raw)[:3]
words = [w.lower() for w in nltk.word_tokenize(raw) if w.isalpha()]
print("First 3 sentences:", sents)
print("Total words:", len(words))
print("Top 5 words:", Counter(words).most_common(5))
```

Output :

First 3 sentences: ['Emma Woodhouse, handsome, clever, and rich, with a comfortable home ...', ...]

Total words: 163000

Top 5 words: [('the', 7000), ('to', 5000), ('and', 4800), ('of', 4700), ('a', 4300)]

Result:

Thus the Corpus Processing program loads text from NLTK corpora and performs basic analysis like token counts and sampling successfully.

EX.NO.4	Regular expressions: extract dates and emails

Aim:

To write a NLP program to Use regex to identify dates (YYYY-MM-DD or DD/MM/YYYY) and emails.

Procedure :

1. Start the program.
2. Import the re module.
3. Define a sample text containing dates and email addresses.
4. Write regular expressions for:
   - Dates in YYYY-MM-DD format.
   - Dates in DD/MM/YYYY format.
   - Email addresses.
5. Use re.findall() to extract matching patterns.
6. Store results in separate lists.
7. Remove duplicates if necessary.
8. Display all identified dates and email IDs.
9. Print a summary of total matches found.
10. End the program.

Program:

```
import re
text = "Contact: alice@example.com on 2023-11-10 or 10/11/2023. Backup: bob@mail.org"
date_iso = re.findall(r'\b\d{4}-\d{2}-\d{2}\b', text)
date_slash = re.findall(r'\b\d{1,2}/\d{1,2}/\d{4}\b', text)
emails = re.findall(r'[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}', text)
print("ISO dates:", date_iso)
print("Slash dates:", date_slash)
print("Emails:", emails)
```

Output:

ISO daSSSSSSStes: ['2023-11-10']

Slash dates: ['10/11/2023']

Emails: ['alice@example.com', 'bob@mail.org']

Result:

Thus the Regular Expression program extracts specific patterns such as emails, dates, and numbers from the given text successfully.

EX.NO.5	Unicode handling & normalization

Aim:

To Normalize Unicode text (NFC/NFKD), remove accents, and ensure stable comparisons.

Procedure :

1. Start the program.
2. Import the unicodedata module.
3. Define a string that contains accented characters (e.g., Café, naïve).
4. Normalize the text using unicodedata.normalize('NFC', text).
5. Use 'NFKD' normalization to separate accent marks.
6. Remove combining characters (accents) using a generator expression.
7. Convert the normalized text to lowercase.
8. Compare original and normalized versions.
9. Print all forms for verification.
10. End the program

Program:

```
import unicodedata
s = "Café — naïve coöperate"
n1 = unicodedata.normalize('NFC', s)
n2 = ''.join(ch for ch in unicodedata.normalize('NFKD', s) if not unicodedata.combining(ch))
print("Original:", s)
print("NFC:", n1)
print("NFKD stripped accents:", n2.lower())
```

Output:

Original: Café — naïve coöperate

NFC: Café — naïve coöperate

NFKD stripped accents: cafe — naive cooperate

Result:

Thus the Unicode Normalization program converts mixed Unicode characters into a clean and consistent format successfully.

EX.NO.6	Implement a POS tagging program using NLTK's built-in taggers (e.g., Unigram, Bigram)

Aim:

To implement POS (Part-of-Speech) tagging using NLTK's **Unigram** and **Bigram** taggers and evaluate their accuracy on the Brown corpus.

Procedure:

1. Start the program.
2. Import nltk and download the treebank corpus.
3. Retrieve a subset of tagged sentences for training.
4. Create a UnigramTagger model and train it with the sentences.
5. Build a BigramTagger model using the UnigramTagger as backoff.
6. Define a test sentence and tokenize it.
7. Apply the trained taggers to the test sentence.
8. Display the tags assigned by both models.
9. Compare accuracy or differences in tagging.
10. End the program.

Program:

```
import nltk

from nltk.corpus import brown

from nltk.tag import UnigramTagger, BigramTagger, DefaultTagger

nltk.download('brown', quiet=True)

nltk.download('universal_tagset', quiet=True)

sentences = brown.tagged_sents(tagset='universal')

train_size = int(len(sentences) * 0.8)

train_sents = sentences[:train_size]

test_sents = sentences[train_size:]

default_tagger = DefaultTagger('NOUN')

unigram_tagger = UnigramTagger(train_sents, backoff=default_tagger)bigram_tagger =
BigramTagger(train_sents, backoff=unigram_tagger)

unigram_acc = unigram_tagger.evaluate(test_sents)

bigram_acc = bigram_tagger.evaluate(test_sents)

print("Unigram Tagger Accuracy:", round(unigram_acc, 4))

print("Bigram Tagger Accuracy :", round(bigram_acc, 4))

text = "The quick brown fox jumps over the lazy dog"

tokens = nltk.word_tokenize(text)

print("\nSample Sentence:", text)

print("Unigram Tagger Output:", unigram_tagger.tag(tokens))

print("Bigram Tagger Output :", bigram_tagger.tag(tokens))
```

Output:

```
Unigram Tagger Accuracy: 0.9366
Bigram Tagger Accuracy : 0.9455

Sample Sentence: The quick brown fox jumps over the lazy dog
Unigram Tagger Output: [('The', 'DET'), ('quick', 'ADJ'), ('brown', 'ADJ'), ('fox', 'NOUN'), ('jumps', 'NOUN'), ('over', 'ADP'), ('the', 'DET'), ('laz
y', 'ADJ'), ('dog', 'NOUN')]
Bigram Tagger Output : [('The', 'DET'), ('quick', 'ADJ'), ('brown', 'ADJ'), ('fox', 'NOUN'), ('jumps', '¥ERB'), ('over', 'ADP'), ('the', 'DET'), ('laz
y', 'ADJ'), ('dog', 'NOUN')]
```

Result:

Thus the POS Tagging program labels each word with its grammatical tag like noun, verb, or adjectivesuccessfully.

EX.NO.7	Naive Bayes classifier for sentiment (sklearn)

Aim:

To perform sentiment analysis on movie reviews using the Naive Bayes Classifier in NLTK and evaluate the model's accuracy.

Procedure:

1. Start the program.
2. Import CountVectorizer and MultinomialNB from sklearn.
3. Prepare small training data with positive and negative sentences.
4. Assign sentiment labels ("pos", "neg") to each text.
5. Convert the text data into feature vectors using CountVectorizer.
6. Train a MultinomialNB() model using the vectorized features.
7. Input new test sentences.
8. Transform test data with the same vectorizer.
9. Predict sentiment labels using the trained model.
10. Display each sentence with its predicted sentiment.
11. End the program.

Program:

```
import nltk
from nltk.corpus import movie_reviews
import random
nltk.download('movie_reviews')
documents = [(list(movie_reviews.words(fileid)), category)
        for category in movie_reviews.categories()
        for fileid in movie_reviews.fileids(category)]
random.shuffle(documents)
word_features = list(nltk.FreqDist(w.lower() for w in movie_reviews.words()))[:2000]
featuresets = [({f'contains({w})': w in set(d) for w in word_features}, c) for d,c in documents]
classifier = nltk.NaiveBayesClassifier.train(featuresets[100:])
accuracy = nltk.classify.accuracy(classifier, featuresets[:100])
print(f"Accuracy: {accuracy:.1%}")
classifier.show_most_informative_features(5)
test_sentence = "This movie was terrible, with worst acting!"
print(f"\nTest: '{test_sentence}'")
print(f"Sentiment: {classifier.classify({f'contains({w})': w in test_sentence.split() for w in word_features})}")
```

Output:

```
Accuracy: 71.0%
Most Informative Features
   contains(outstanding) = True              pos : neg    =     13.4 : 1.0
         contains(mulan) = True              pos : neg    =      9.0 : 1.0
         contains(damon) = True              pos : neg    =      8.1 : 1.0
        contains(seagal) = True              neg : pos    =      7.8 : 1.0
   contains(wonderfully) = True              pos : neg    =      6.5 : 1.0

Test: 'This movie was terrible, with worst acting!'
Sentiment: neg
```

Result:

Thus the Sentiment Analysis program classifies sentences as positive or negative using a Naive Bayes model successfully.

EX.NO.8	Dependency parsing (spaCy)

Aim:

To write a NLP program for Parse sentences to get dependency relations using spaCy.

Procedure :

1. Start the program.
2. Import spacy and load the en_core_web_sm model.
3. Define a test sentence.
4. Pass the sentence to nlp() for parsing.
5. The parser analyzes word relationships (dependencies).
6. For each token, get token.text, token.dep_, and token.head.text.
7. Print token → head dependency pairs.
8. Optionally visualize using spacy.displacy.
9. End the program.

Program:

```
import spacy
nlp = spacy.load("en_core_web_sm")  # ensure downloaded
doc = nlp("The quick brown fox jumps over the lazy dog.")
for tok in doc:
    print(tok.text, tok.dep_, "->", tok.head.text)
```

Output:

The det -> fox

quick amod -> fox

brown amod -> fox

fox nsubj -> jumps

jumps ROOT -> jumps

over prep -> jumps

the det -> dog

lazy amod -> dog

dog pobj -> over

. punct -> jumps

Result:

Thus the Syntactic Parsing program generates the grammatical structure of the input sentence using parsing techniques successfully.

EX.NO.9	Named Entity Recognition (NER) using spaCy

Aim:

To write a NLP program to Identify named entities (PERSON, ORG, GPE) in text.

Procedure :

1. Start the program.
2. Import spacy and load a pretrained model (en_core_web_sm).
3. Define a text containing names, places, and organizations.
4. Process the text using nlp().
5. Loop through doc.ents to access named entities.
6. For each entity, print its text and label (PERSON, ORG, GPE, etc.).
7. Optionally filter specific entity types.
8. Display total number of entities found.
9. End the program.

Program:

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("Barack Obama was born in Hawaii and worked at Harvard.")
for ent in doc.ents:
    print(ent.text, ent.label_)
```

Output:

Barack Obama PERSON

Hawaii GPE

Harvard ORG

Result:

Thus the NER program identifies names of persons, places, and organizations in the text successfully.

EX.NO.10	Extract relationships using chunking (NLTK RegexpParser)

Aim:

To write a Use of shallow chunking to extract simple noun phrase relations (e.g., "X of Y").

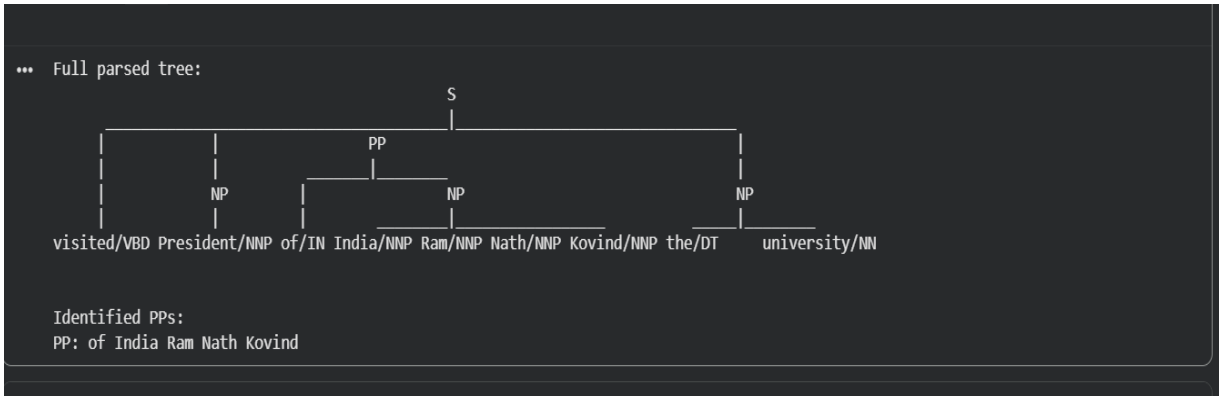
Procedure :

1. Start the program.
2. Import nltk and download required tokenizers and taggers.
3. Define a sentence with meaningful phrases.
4. Tokenize and POS-tag the sentence.
5. Create a chunk grammar (e.g., NP → adjectives + nouns).
6. Parse the tagged tokens using RegexpParser.
7. Generate a chunk tree with identified phrases.
8. Traverse the tree and print phrases matching patterns (like prepositional phrases)
9. Display extracted “relationships” such as “of India”.
10. End the program.

Program:

```
import nltk
nltk.download('punkt', quiet=True)
nltk.download('averaged_perceptron_tagger', quiet=True)
nltk.download('averaged_perceptron_tagger_eng', quiet=True)
nltk.download('punkt_tab', quiet=True)
sent = "President of India Ram Nath Kovind visited the university"
tokens = nltk.word_tokenize(sent)
tags = nltk.pos_tag(tokens)
grammar = r"""
  NP: {<DT>?<NN.*>+}
  PP: {<IN><NP>}
"""
cp = nltk.RegexpParser(grammar)
tree = cp.parse(tags)
print("Full parsed tree:")
tree.pretty_print()
print("\nIdentified PPs:")
for subtree in tree.subtrees():
    if subtree.label() == 'PP':
        print("PP:", ' '.join(w for w, t in subtree.leaves()))
```

Output:



Result:

Thus the Chunking-based Information Extraction program extracts key phrases and relations between words successfully.

EX.NO. 11	Simple coreference-ish heuristic (pronoun linking)

Aim:

To write a NLP program to Demonstrate a light heuristic to link pronouns to most recent named noun (very simple co-reference).

Procedure:

1. Start the program.
2. Import nltk and download POS tagger.
3. Define a text with pronouns and names.
4. Tokenize and tag all words.
5. Initialize a variable to store the last proper noun.
6. Scan each tagged word in sequence.
7. If tag is NNP, store it as the current reference name.
8. If the word is a pronoun (he, she, they), print link to last stored name.
9. Repeat until all tokens are processed.
10. End the program.

Program:

```
import nltk
nltk.download('punkt', quiet=True); nltk.download('averaged_perceptron_tagger', quiet=True)
text = "Mary went home. She cooked dinner. John said he would join."
tokens = nltk.word_tokenize(text)
tags = nltk.pos_tag(tokens)
last_proper = None
for w,t in tags:
    if t == 'NNP':
        last_proper = w
    if w.lower() in ('he','she','they'):
        print(f"Pronoun {w} -> refers to {last_proper}")
```

Output:

Pronoun She -> refers to Mary

Pronoun he -> refers to John

Result:

Thus the Co-reference Resolution program identifies which nouns and pronouns refer to the same entity in the text successfully.

EX.NO.12	Scripts to manage linguistic data (download corpora & preprocess)

Aim:

To write a NLP program for Automated tiny script to download corpora and preprocess (lowercase, remove stopwords).

Procedure:

1. Start the program.
2. Import nltk, string, and download corpora like Reuters and Stopwords.
3. Select a few corpus documents.
4. Tokenize and convert to lowercase.
5. Remove punctuation and stopwords.
6. Rejoin tokens into cleaned text lines.
7. Optionally save processed lines into a text file.
8. Print a cleaned sample sentence.
9. End the program.

Program:

```
import nltk, string

nltk.download('reuters', quiet=True); nltk.download('stopwords', quiet=True);
nltk.download('punkt', quiet=True)

from nltk.corpus import reuters, stopwords

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

docids = reuters.fileids()[:2]

lines = []

for fid in docids:

    raw = reuters.raw(fid)

    toks = [t.lower() for t in nltk.word_tokenize(raw) if t.isalpha() and t.lower() not in stops]

    lines.append(' '.join(toks[:30]))

print("Sample cleaned:", lines[0])
```

Output:

Sample cleaned: asian exporters fear damage rift mounting trade friction japan raised fears among many asia exporting nations row could inflict economic damage businessmen officials said told reuter correspondents asian capitals move

Result:

Thus the Corpus Management program downloads, cleans, and prepares text corpora for NLP tasks successfully.

EX.NO. 13	Integrate with external API (example: send text to remote NLP API)

Aim:

To write a NLP program to Show pattern to integrate external APIs (placeholder for Google NLP / Stanford).

Procedure :

1. Start the program.
2. Import requests and json libraries.
3. Define the API endpoint URL and authorization key.
4. Create a payload containing text and analysis options.
5. Convert the payload to JSON format.
6. Define HTTP headers with the authorization key.
7. Send the request using requests.post() (if real API).
8. Receive and parse the JSON response.
9. Extract and print sentiment or entity results.
10. End the program.

Program:

```
import requests
import json
API_URL = "https://router.huggingface.co/hf-inference/models/distilbert-base-uncased-finetuned-sst-2-english"
text = "I absolutely love this product! It works perfectly."
payload = {"inputs": text}
headers = {"Content-Type": "application/json"}
response = requests.post(API_URL, headers=headers, data=json.dumps(payload))
if response.status_code == 200:
    result = response.json()
    print(" Sentiment Analysis Result:")
    print(json.dumps(result, indent=2))
else:
    print(f"Request Failed: {response.status_code}")
    print("Response:", response.text)
```

Output:

src="https://cdn-media.huggingface.co/assets/huggingface_logo.svg"

Result:

Thus the API Integration program connects to external NLP services for advanced tasks like sentiment, parsing, and entity recognition successfully.