`

SHRI MADHWA VADIRAJA INSTITUTE OF TECHNOLOGY & MANAGEMENT

(A unit of Shri Sode Vadiraja Mutt Education Trust ®)
(Affiliated to Visvesvaraya Technological University, Belagavi)
Vishwothama Nagar, Bantakal – 574115, Udupi District, Karnataka

सर्वे भद्राणि पश्यन्तु

SMVITM

BAI601
NATURAL LANGUAGE PROCESSING LAB

LABORATORY MANUAL (2024-25)

6TH SEMESTER B.E.

NAME OF THE STUDENT : ______________________________

UNIVERSITY SEAT NUMBER : ______________________________

SECTION & BATCH : ______________________________

Prepared by:
Ms. Megha Rani R
Assistant Professor

DEPARTMENT OF
ARTIFICIAL INTELLIGENCE & MACHINE LEARNING

TABLE OF CONTENTS

	● Map Words to Properties Using Python Dictionaries ● Study Rule based tagger, Unigram Tagger Find different words from a given plain text without any space by comparing this text with a given corpus of words. Also find the score of words.	
7	Write a Python program to find synonyms and antonyms of the word "active" using WordNet.	
8	Implement the machine translation application of NLP where it needs to train a machine translation model for a language with limited parallel corpora. Investigate and incorporate techniques to improve performance in low-resource scenarios.	

NATURAL LANGUAGE PROCESSING		Semester	6
Course Code	BAI601	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100
Credits	04	Exam Hours	03
Examination nature (SEE)	Theory		

Course objectives:

This course will enable students to,

- Learn the importance of natural language modelling
- Understand the Applications of natural language processing
- Study spelling, error detection and correction methods and parsing techniques in NLP
- Illustrate the information retrieval models in natural language processing

Programs List:

1	Write a Python program for the following preprocessing of text in NLP: ● Tokenization ● Filtration ● Script Validation ● Stop Word Removal ● Stemming
2	Demonstrate the N-gram modeling to analyze and establish the probability distribution across sentences and explore the utilization of unigrams, bigrams, and trigrams in diverse English sentences to illustrate the impact of varying n-gram orders on the calculated probabilities.
3	Investigate the Minimum Edit Distance (MED) algorithm and its application in string comparison and the goal is to understand how the algorithm efficiently computes the minimum number of edit operations required to transform one string into another. ● Test the algorithm on strings with different type of variations (e.g., typos, substitutions, insertions, deletions) ● Evaluate its adaptability to different types of input variations
4	Write a program to implement top-down and bottom-up parser using appropriate context free grammar
5	Given the following short movie reviews, each labeled with a genre, either comedy or action: ● fun, couple, love, love comedy ● fast, furious, shoot action ● couple, fly, fast, fun, fun comedy ● furious, shoot, shoot, fun action ● fly, fast, shoot, love action and A new document D: fast, couple, shoot, fly Compute the most likely class for D. Assume a Naive Bayes classifier and use add-1 smoothing for the likelihoods.
6	Demonstrate the following using appropriate programming tool which illustrates the use of information retrieval in NLP: ● Study the various Corpus – Brown, Inaugural, Reuters, udhr with various methods like filelds, raw, words, sents, categories 3

	● Create and use your own corpora (plaintext, categorical) ● Study Conditional frequency distributions ● Study of tagged corpora with methods like tagged_sents, tagged_words ● Write a program to find the most frequent noun tags ● Map Words to Properties Using Python Dictionaries ● Study Rule based tagger, Unigram Tagger Find different words from a given plain text without any space by comparing this text with a given corpus of words. Also find the score of words.
7	Write a Python program to find synonyms and antonyms of the word "active" using WordNet.
8	Implement the machine translation application of NLP where it needs to train a machine translation model for a language with limited parallel corpora. Investigate and incorporate techniques to improve performance in low-resource scenarios.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together..

CIE for the theory component of the IPCC (maximum marks 50)

● IPCC means practical portion integrated with the theory of the course.

● CIE marks for the theory component are 25 marks and that for the practical component is 25 marks.

● 25 marks for the theory component are split into 15 marks for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and 10 marks for other assessment methods mentioned in 22OB4.2. The first test at the end of 40-50% coverage of the syllabus and the second test after covering 85-90% of the syllabus.

● Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for 25 marks).

CIE for the practical component of the IPCC

- 15 marks for the conduction of the experiment and preparation of laboratory record, and 10 marks for the test to be conducted after the completion of all the laboratory sessions.
- On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.

● The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to 15 marks.

●The laboratory test (duration 02/03 hours) after completion of all the experiments shall be conducted for 50 marks and scaled down to 10 marks.

● Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for 25 marks.

●The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

LAB EXPERIMENTS

Program 1

1. Write a Python program for the following preprocessing of text in NLP:

- **Tokenization**
- **Filtration**
- **Script Validation**
- **Stop Word Removal**
- **Stemming**

Recommended Setup:

1. Install Anaconda from https://www.anaconda.com/
2. Open Anaconda Navigator → Launch Jupyter Notebook or Spyder.
3. Install Required Libraries: pip install nltk

CODE:

```
import nltk
import re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
```

```
nltk.download('punkt')
nltk.download('stopwords')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\Lenovo\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\Lenovo\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
True
```

```
text = "Python is amazing! こんにちは (Hello in Japanese), नमस्ते (Hello in Hindi), Привет (Hello in Russian). NLP is fun! 123 😊'


print("\nOriginal Text:\n", text)
```

```
Original Text:
 Python is amazing! こんにちは (Hello in Japanese), नमस्ते (Hello in Hindi), Привет (Hello in Russian). NLP is fun! 123 😊
```

```
tokens = word_tokenize(text)
print("\nStep 1 - Tokenization:\n", tokens)
```

```
Step 1 - Tokenization:
 ['Python', 'is', 'amazing', '!', 'こんにちは', '(', 'Hello', 'in', 'Japanese', ')', ',', 'नमस्ते', '(', 'Hello', 'in', 'Hindi',
')', ',', 'Привет', '(', 'Hello', 'in', 'Russian', ')', '.', 'NLP', 'is', 'fun', '!', '123', '😊']
```

```
filtered_tokens = [word for word in tokens if word.isalpha()]
print("\nStep 2 - Filtration (Removing non-alphabetic characters):\n", filtered_tokens)
```

```
Step 2 - Filtration (Removing non-alphabetic characters):
 ['Python', 'is', 'amazing', 'こんにちは', 'Hello', 'in', 'Japanese', 'Hello', 'in', 'Hindi', 'Привет', 'Hello', 'in', 'Russia
n', 'NLP', 'is', 'fun']
```

```
valid_tokens = [word for word in filtered_tokens if re.match(r"^[A-Za-z]+$", word)]
print("\nStep 3 - Script Validation (Ensuring valid words):\n", valid_tokens)
```

```
Step 3 - Script Validation (Ensuring valid words):
 ['Python', 'is', 'amazing', 'Hello', 'in', 'Japanese', 'Hello', 'in', 'Hindi', 'Hello', 'in', 'Russian', 'NLP', 'is', 'fun']
```

```
stop_words = set(stopwords.words('english'))
meaningful_words = [word.lower() for word in valid_tokens if word.lower() not in stop_words]
print("\nStep 4 - Stop Word Removal:\n", meaningful_words)
```

```
Step 4 - Stop Word Removal:
 ['python', 'amazing', 'hello', 'japanese', 'hello', 'hindi', 'hello', 'russian', 'nlp', 'fun']
```

```
stemmer = PorterStemmer()
stemmed_words = [stemmer.stem(word) for word in meaningful_words]
print("\nStep 5 - Stemming:\n", stemmed_words)
```

```
Step 5 - Stemming:
 ['python', 'amaz', 'hello', 'japanes', 'hello', 'hindi', 'hello', 'russian', 'nlp', 'fun']
```

Program 2

Demonstrate the N-gram modeling to analyze and establish the probability distribution across sentences and explore the utilization of unigrams, bigrams, and trigrams in diverse English sentences to illustrate the impact of varying n-gram orders on the calculated probabilities.

CODE:

```
# Use an alternative approach without relying on NLTK's punkt tokenizer


from collections import Counter

import pandas as pd


# Sample sentences
sentences = [
    "The quick brown fox jumps over the lazy dog",
    "A journey of a thousand miles begins with a single step",
    "To be or not to be that is the question",
    "All that glitters is not gold",
    "An apple a day keeps the doctor away"
]
# Function to tokenize sentences (simple split-based tokenizer)
def tokenize(sentence):
    return sentence.lower().split()
# Function to generate n-grams
def generate_ngrams(tokens, n):
    return [tuple(tokens[i:i + n]) for i in range(len(tokens) - n + 1)]
# Collect all n-grams
unigrams = []
bigrams = []
trigrams = []



for sentence in sentences:
    tokens = tokenize(sentence)
```

```
    unigrams.extend(generate_ngrams(tokens, 1))
    bigrams.extend(generate_ngrams(tokens, 2))
    trigrams.extend(generate_ngrams(tokens, 3))

# Count occurrences
unigram_counts = Counter(unigrams)
bigram_counts = Counter(bigrams)
trigram_counts = Counter(trigrams)

# Function to calculate n-gram probabilities
def calculate_ngram_probabilities(ngram_counts, lower_order_counts=None):
    probabilities = {}
    for ngram, count in ngram_counts.items():
        if lower_order_counts:
            prefix = ngram[:-1]
            prefix_count = lower_order_counts[prefix] if prefix in lower_order_counts else 1  # Smoothing
            probabilities[ngram] = count / prefix_count
        else:
            probabilities[ngram] = count / sum(ngram_counts.values())
    return probabilities

# Compute probabilities
unigram_probs = calculate_ngram_probabilities(unigram_counts)
bigram_probs = calculate_ngram_probabilities(bigram_counts, unigram_counts)
trigram_probs = calculate_ngram_probabilities(trigram_counts, bigram_counts)

# Convert to DataFrame for visualization
df_unigrams = pd.DataFrame(unigram_probs.items(), columns=["Unigram", "Probability"])
df_bigrams = pd.DataFrame(bigram_probs.items(), columns=["Bigram", "Probability"])
df_trigrams = pd.DataFrame(trigram_probs.items(), columns=["Trigram", "Probability"])


# Display results
# Display the results in a structured format
```

```
print("Unigram Probabilities:")

print(df_unigrams.to_string(index=False))


print("\nBigram Probabilities:")

print(df_bigrams.to_string(index=False))


print("\nTrigram Probabilities:")

print(df_trigrams.to_string(index=False))
```

OUTPUT

```
Unigram Probabilities:
    Unigram  Probability
     (the,)     0.090909
   (quick,)     0.022727
   (brown,)     0.022727
     (fox,)     0.022727
   (jumps,)     0.022727
    (over,)     0.022727
    (lazy,)     0.022727
     (dog,)     0.022727
       (a,)     0.090909
 (journey,)     0.022727
      (of,)     0.022727
(thousand,)     0.022727
   (miles,)     0.022727
  (begins,)     0.022727
    (with,)     0.022727
  (single,)     0.022727
```

```
    (step,)     0.022727
      (to,)     0.045455
      (be,)     0.045455
      (or,)     0.022727
     (not,)     0.045455
    (that,)     0.045455
      (is,)     0.045455
(question,)     0.022727
     (all,)     0.022727
(glitters,)     0.022727
    (gold,)     0.022727
      (an,)     0.022727
   (apple,)     0.022727
     (day,)     0.022727
   (keeps,)     0.022727
  (doctor,)     0.022727
    (away,)     0.022727
```

```
Bigram Probabilities:
                Bigram  Probability
          (the, quick)         0.25
        (quick, brown)         1.00
          (brown, fox)         1.00
          (fox, jumps)         1.00
         (jumps, over)         1.00
           (over, the)         1.00
           (the, lazy)         0.25
           (lazy, dog)         1.00
          (a, journey)         0.25
         (journey, of)         1.00
               (of, a)         1.00
         (a, thousand)         0.25
     (thousand, miles)         1.00
       (miles, begins)         1.00
        (begins, with)         1.00
             (with, a)         1.00
           (a, single)         0.25
        (single, step)         1.00
              (to, be)         1.00
              (be, or)         0.50
             (or, not)         1.00
             (not, to)         0.50
            (be, that)         0.50
            (that, is)         0.50
             (is, the)         0.50
       (the, question)         0.25
           (all, that)         1.00
      (that, glitters)         0.50
        (glitters, is)         1.00
             (is, not)         0.50
           (not, gold)         0.50
           (an, apple)         1.00
            (apple, a)         1.00
              (a, day)         0.25
          (day, keeps)         1.00
          (keeps, the)         1.00
         (the, doctor)         0.25
        (doctor, away)         1.00
```

```
Trigram Probabilities:
                  Trigram  Probability
     (the, quick, brown)          1.0
     (quick, brown, fox)          1.0
     (brown, fox, jumps)          1.0
      (fox, jumps, over)          1.0
      (jumps, over, the)          1.0
       (over, the, lazy)          1.0
        (the, lazy, dog)          1.0
        (a, journey, of)          1.0
        (journey, of, a)          1.0
```

(of, a, thousand)	1.0
(a, thousand, miles)	1.0
(thousand, miles, begins)	1.0
(miles, begins, with)	1.0
(begins, with, a)	1.0
(with, a, single)	1.0
(a, single, step)	1.0
(to, be, or)	0.5
(be, or, not)	1.0
(or, not, to)	1.0
(not, to, be)	1.0
(to, be, that)	0.5
(be, that, is)	1.0
(that, is, the)	1.0
(is, the, question)	1.0
(all, that, glitters)	1.0
(that, glitters, is)	1.0
(glitters, is, not)	1.0
(is, not, gold)	1.0
(an, apple, a)	1.0
(apple, a, day)	1.0
(a, day, keeps)	1.0
(day, keeps, the)	1.0
(keeps, the, doctor)	1.0
(the, doctor, away)	1.0

Program 3

Investigate the Minimum Edit Distance (MED) algorithm and its application in string comparison and the goal is to understand how the algorithm efficiently computes the minimum number of edit operations required to transform one string into another.

- **Test the algorithm on strings with different type of variations (e.g., typos, substitutions, insertions, deletions)**
- **Evaluate its adaptability to different types of input variations**

CODE:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def min_edit_distance(str1, str2):
    m, n = len(str1), len(str2)
    dp = np.zeros((m+1, n+1), dtype=int)

    for i in range(m+1):
        for j in range(n+1):
            if i == 0:
                dp[i][j] = j  # Insert all characters of str2
            elif j == 0:
                dp[i][j] = i  # Remove all characters of str1
            elif str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1]  # If last characters are same, ignore them
            else:
                dp[i][j] = 1 + min(dp[i-1][j],    # Remove
                                   dp[i][j-1],    # Insert
                                   dp[i-1][j-1])  # Replace

    return dp[m][n]
```

```
# Testing different variations
test_cases = [
    ("hello", "helo"),   # Deletion
    ("hello", "heloo"),  # Insertion
    ("hello", "hallo"),  # Substitution
    ("hello", "world"),  # Completely different
    ("cat", "cut"),      # One character substitution
    ("intention", "execution"),  # Complex example
]

results = []
for str1, str2 in test_cases:
    med = min_edit_distance(str1, str2)
    results.append((str1, str2, med))

# Creating a DataFrame to display results
df_results = pd.DataFrame(results, columns=["String 1", "String 2", "Min Edit Distance"])

# Display results
df_results
```

OUTPUT:

	String 1	String 2	Min Edit Distance
0	hello	helo	1
1	hello	heloo	1
2	hello	hallo	1
3	hello	world	4
4	cat	cut	1
5	intention	execution	5

Program 4

Write a program to implement top-down and bottom-up parser using appropriate context free grammar.

CODE:

```
import nltk
from nltk import CFG

# Define the context-free grammar (CFG)
grammar = CFG.fromstring("""
    S -> NP VP
    NP -> Det N | Det Adj N | PN
    VP -> V NP | V
    Det -> 'the' | 'a'
    N -> 'cat' | 'dog' | 'man' | 'park'
    Adj -> 'big' | 'small'
    V -> 'chased' | 'saw' | 'ate'
    PN -> 'John' | 'Mary'
""")

# List of test sentences
test_sentences = [
    "the cat chased the dog",
    "John saw the dog",
    "Mary ate",
    "the big dog saw a cat"
]

for sent in test_sentences:
    sentence = sent.split()
    print(f"\n=== Parsing: {' '.join(sentence)} ===")

    # **Top-Down Parsing (Recursive Descent)**
    print("\n**Top-Down Parsing (Recursive Descent)**")
    rd_parser = nltk.RecursiveDescentParser(grammar)
    found_parse = False
    for tree in rd_parser.parse(sentence):
        found_parse = True
        print(tree)
        tree.pretty_print()
    if not found_parse:
        print("No valid parse tree found using Top-Down Parsing.")
```

```
# **Bottom-Up Parsing (Chart Parser)**
print("\n**Bottom-Up Parsing (Chart Parser)**")
chart_parser = nltk.ChartParser(grammar)
found_parse = False
for tree in chart_parser.parse(sentence):
    found_parse = True
    print(tree)
    tree.pretty_print()
if not found_parse:
    print("No valid parse tree found using Bottom-Up Parsing.")
```

OUTPUT:

```
=== Parsing: the cat chased the dog ===

**Top-Down Parsing (Recursive Descent)**
(S (NP (Det the) (N cat)) (VP (V chased) (NP (Det the) (N dog))))
                S
       _________|_____
      |              VP
      |         _____|___
      NP        |        NP
  ___|___       |    ___|___
Det      N      V   Det      N
 |       |      |    |       |
the     cat  chased the     dog


**Bottom-Up Parsing (Chart Parser)**
(S (NP (Det the) (N cat)) (VP (V chased) (NP (Det the) (N dog))))

                S
       _________|_____
      |              VP
      |         _____|___
      NP        |        NP
  ___|___       |    ___|___
Det      N      V   Det      N
 |       |      |    |       |
the     cat  chased the     dog


=== Parsing: John saw the dog ===

**Top-Down Parsing (Recursive Descent)**
(S (NP (PN John)) (VP (V saw) (NP (Det the) (N dog))))
```

```
          S
 _________|___
|             VP
|     ________|___
NP   |            NP
|    |         ___|___
PN   V       Det      N
|    |        |       |
John saw     the     dog
```

```
**Bottom-Up Parsing (Chart Parser)**
(S (NP (PN John)) (VP (V saw) (NP (Det the) (N dog))))
```

```
          S
 _________|___
|             VP
|     ________|___
NP   |            NP
|    |         ___|___
PN   V       Det      N
|    |        |       |
John saw     the     dog
```

```
=== Parsing: Mary ate ===

**Top-Down Parsing (Recursive Descent)**
(S (NP (PN Mary)) (VP (V ate)))
```

```
      S
  ____|___
 NP       VP
 |        |
 PN       V
 |        |
Mary     ate
```

```
**Bottom-Up Parsing (Chart Parser)**
(S (NP (PN Mary)) (VP (V ate)))
```

```
        S
    ____|___
   NP       VP
   |        |
   PN       V
   |        |
  Mary     ate


=== Parsing: the big dog saw a cat ===

**Top-Down Parsing (Recursive Descent)**
(S
  (NP (Det the) (Adj big) (N dog))
  (VP (V saw) (NP (Det a) (N cat))))

          S
       ___|_______
      |          VP
      |       ___|___
     NP      |       NP
  ___|___    |    ___|___
Det Adj  N   V  Det      N
 |   |   |   |   |       |
the big dog saw  a      cat
```

```
**Bottom-Up Parsing (Chart Parser)**
(S
  (NP (Det the) (Adj big) (N dog))
  (VP (V saw) (NP (Det a) (N cat))))
         S
      ___|______
     |          VP
     |       ___|___
     NP     |       NP
  ___|___   |    ___|___
Det Adj  N   V  Det      N
 |   |   |   |   |       |
the big dog saw  a      cat
```

Program 5

Given the following short movie reviews, each labeled with a genre, either comedy or action:

● fun, couple, love, love comedy

● fast, furious, shoot action

● couple, fly, fast, fun, fun comedy

● furious, shoot, shoot, fun action

● fly, fast, shoot, love action and

A new document D: fast, couple, shoot, fly Compute the most likely class for D. Assume a Naive Bayes classifier and use add-1 smoothing for the likelihoods.

Code:

```
from collections import Counter
import numpy as np

def train_naive_bayes(docs, labels):
    vocab = set()
    word_counts = {"comedy": Counter(), "action": Counter()}
    class_counts = {"comedy": 0, "action": 0}

    for doc, label in zip(docs, labels):
        words = doc.split(', ')
        vocab.update(words)
        word_counts[label].update(words)
        class_counts[label] += 1

    vocab_size = len(vocab)
    total_docs = sum(class_counts.values())
    class_probs = {cls: np.log(class_counts[cls] / total_docs) for cls in class_counts}

    return word_counts, class_probs, vocab, vocab_size, class_counts

def compute_likelihood(word_counts, vocab_size, class_word_count, doc, smoothing=1):
    likelihood = 0
    for word in doc.split(', '):
        word_freq = word_counts[word] + smoothing
        likelihood += np.log(word_freq / (class_word_count + smoothing * vocab_size))
    return likelihood

def predict_naive_bayes(word_counts, class_probs, vocab, vocab_size, class_counts, doc):
    scores = {}
    for cls in class_probs:
```

```
        class_word_count = sum(word_counts[cls].values())
        likelihood = compute_likelihood(word_counts[cls], vocab_size, class_word_count, doc)
        scores[cls] = class_probs[cls] + likelihood
    return max(scores, key=scores.get)

# Training data
docs = [
    "fun, couple, love, love",
    "fast, furious, shoot",
    "couple, fly, fast, fun, fun",
    "furious, shoot, shoot, fun",
    "fly, fast, shoot, love"
]
labels = ["comedy", "action", "comedy", "action", "action"]

# Train Naive Bayes model
word_counts, class_probs, vocab, vocab_size, class_counts = train_naive_bayes(docs, labels)

# New document
doc_D = "fast, couple, shoot, fly"
predicted_class = predict_naive_bayes(word_counts, class_probs, vocab, vocab_size, class_counts, doc_D)

print(f"The most likely class for document D is: {predicted_class}")
```

Output :

```
The most likely class for document D is: action
```

Program 6

Demonstrate the following using appropriate programming tool which illustrates the use of information retrieval in NLP:

- **Study the various Corpus – Brown, Inaugural, Reuters, udhr with various methods like filelds, raw, words, sents, categories.**
- **Create and use your own corpora (plaintext, categorical)**
- **Study Conditional frequency distributions**
- **Study of tagged corpora with methods like tagged_sents, tagged_words .**
- **Write a program to find the most frequent noun tags**
- **Map Words to Properties Using Python Dictionaries**
- **Study Rule based tagger, Unigram Tagger.**

Find different words from a given plain text without any space by comparing this text with a given corpus of words. Also find the score of words.

Code:

```
import nltk
nltk.download('brown')
nltk.download('inaugural')
nltk.download('reuters')
nltk.download('udhr')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('universal_tagset')
nltk.download('tagsets')
nltk.download('treebank')
nltk.download('words')
```

Study the various Corpus – Brown, Inaugural, Reuters, udhr with various methods like filelds, raw, words, sents, categories.

```
from nltk.corpus import brown, inaugural, reuters, udhr

# Brown Corpus
print("BROWN Corpus:")
print("Categories:", brown.categories())#Lists all the categories (or genres) of texts in the Brown Corpus, like 'news', 'fiction', 'editorial', 'romance', etc.
print("Words:", brown.words(categories='news')[:10])#Fetches first 10 words from the 'news' category
print("Sents:", brown.sents(categories='news')[:2])#Displays the first 2 sentences from the 'news' category, where each sentence is a list of word tokens.

# Inaugural Corpus
print("\nINAUGURAL Corpus:")
```

```
print("File IDs:", inaugural.fileids()[:5])#List first 5 file IDs (each corresponds to a U.S. presidential inaugural address)
print("Words:", inaugural.words('2009-Obama.txt')[:10])

# Reuters Corpus
print("\nREUTERS Corpus:")
print("Categories:", reuters.categories()[:5])# news documents such as crude, trade, money-fx
print("Words:", reuters.words(categories='crude')[:10])
print("Sents:", reuters.sents(categories='crude')[:2])

# UDHR Corpus
print("\nUDHR Corpus:")
print("Languages:", udhr.fileids()[:5])#Universal Declaration of Human Rights in different languages.
print("Words (English):", udhr.words('English-Latin1')[:10])
```

Create and use your own corpora (plaintext, categorical)

```
from nltk.corpus import PlaintextCorpusReader

# Plaintext Corpus
corpus_root = 'my_corpus'# creates directory
import os
os.makedirs(corpus_root, exist_ok=True)
with open(os.path.join(corpus_root, 'sample.txt'), 'w') as f:
   f.write("This is a custom corpus file. It can be used for testing.")

custom_corpus = PlaintextCorpusReader(corpus_root, '.*\.txt')
print("\nCustom Corpus Words:", custom_corpus.words())
```

Output:

```
Custom Corpus Words: ['This', 'is', 'a', 'custom', 'corpus', 'file', '.', ...]
```

Study Conditional frequency distributions

```
import nltk
nltk.download('brown')  # Only needed once

from nltk.corpus import brown
from nltk import ConditionalFreqDist

# Conditional Frequency Distribution
word_category_pairs = [
   (word.lower(), cat)                    # Create (word, category) pairs
   for cat in brown.categories()           # Loop through each category (e.g., 'news', 'romance', 'fiction'...)
   for word in brown.words(categories=cat)    # Loop through each word in that category
]
cfd = ConditionalFreqDist(word_category_pairs)
```

print("\nCFD Example (word 'news'):", cfd['news'].most_common(3))

Output:

```
CFD Example (word 'news'): [('editorial', 18), ('news', 14), ('fiction', 13)]
```

Study of tagged corpora with methods like tagged_sents, tagged_words

```
from nltk.corpus import treebank

print("\nTagged Sents:", treebank.tagged_sents()[:2])
print("Tagged Words:", treebank.tagged_words()[:10])
```

OUTPUT:

```
Tagged Sents: [[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years', 'NNS'), ('old', 'JJ'), (',', ','), ('will', 'MD'), ('join', 'VB'), ('the', 'DT'), ('board', 'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'), ('director', 'NN'), ('Nov.', 'NNP'), ('29', 'CD'), ('.', '.')], [('Mr.', 'NNP'), ('Vinken', 'NNP'), ('is', 'VBZ'), ('chairman', 'NN'), ('of', 'IN'), ('Elsevier', 'NNP'), ('N.V.', 'NNP'), (',', ','), ('the', 'DT'), ('Dutch', 'NNP'), ('publishing', 'VBG'), ('group', 'NN'), ('.', '.')]]
Tagged Words: [('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years', 'NNS'), ('old', 'JJ'), (',', ','), ('will', 'MD'), ('join', 'VB'), ('the', 'DT')]
```

Write a program to find the most frequent noun tags

```
from collections import Counter

tags = [tag for (word, tag) in treebank.tagged_words()]
tag_freq = Counter(tags)

print("\nMost Frequent POS Tags:", tag_freq.most_common(5))

# Noun Tags
noun_tags = [tag for tag in tags if tag.startswith('NN')]
noun_freq = Counter(noun_tags)
print("\nMost Frequent NOUN Tags:", noun_freq.most_common(3))
```

OUTPUT:

```
Most Frequent POS Tags: [('NN', 13166), ('IN', 9857), ('NNP', 9410), ('DT', 8165), ('-NONE-', 6592)]

Most Frequent NOUN Tags: [('NN', 13166), ('NNP', 9410), ('NNS', 6047)]
```

Map Words to Properties Using Python Dictionaries

```
word_properties = {
  'cat': {'type': 'animal', 'sound': 'meow'},
  'car': {'type': 'vehicle', 'fuel': 'petrol'},
  'apple': {'type': 'fruit', 'color': 'red'}
}
print("\nWord Properties:")
for word, props in word_properties.items():
```

```
    print(f"{word.title()} → {props}")
```

OUTPUT:

```
Word Properties:
Cat → {'type': 'animal', 'sound': 'meow'}
Car → {'type': 'vehicle', 'fuel': 'petrol'}
Apple → {'type': 'fruit', 'color': 'red'}
```

Study Rule based tagger, Unigram Tagger.

```
from nltk.tag import DefaultTagger, UnigramTagger
from nltk.corpus import treebank

default_tagger = DefaultTagger('NN')
print("\nDefaultTagger Test:", default_tagger.tag(['Hello', 'world']))

train_data = treebank.tagged_sents()[:3000]
test_data = treebank.tagged_sents()[3000:]

unigram_tagger = UnigramTagger(train_data, backoff=default_tagger)
print("UnigramTagger Accuracy:", unigram_tagger.evaluate(test_data))
```

OUTPUT:

```
DefaultTagger Test: [('Hello', 'NN'), ('world', 'NN')]
C:\Users\Lenovo\AppData\Local\Temp\ipykernel_2336\2980264368.py:11: DeprecationWarning:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print("UnigramTagger Accuracy:", unigram_tagger.evaluate(test_data))
UnigramTagger Accuracy: 0.8741204403194475
```

Find different words from a given plain text without any space by comparing this text with a given corpus of words.

```
from nltk.corpus import words
nltk.download('words')
wordlist= set(words.words())

def segment_filtered(text, min_len=2):
    results = []

    def helper(text, sentence):
        if not text:
            results.append(sentence)
            return
        for i in range(1, len(text)+1):
            word = text[:i]
            if len(word) >= min_len and word in wordlist:
```

```
            helper(text[i:], sentence + [word])

    helper(text, [])
    return results
```

Also find the score of words

```
results = segment_filtered("themanrantosave", min_len=2)
def score_by_fewest_words(segmented_list):
    return[(words,len(words))for words in segmented_list]
scored = score_by_fewest_words(results)
scored.sort(key=lambda x: x[1])

print("\nFiltered & Scored Segmentations:")
for words, score in scored:
    print(" →", " ".join(words), "| Word Count:", score)
```

Output:

```
Filtered & Scored Segmentations:
 → the man ran to save | Word Count: 5
 → the man rant os ave | Word Count: 5
 → them an ran to save | Word Count: 5
 → them an rant os ave | Word Count: 5
 → th em an ran to save | Word Count: 6
 → th em an rant os ave | Word Count: 6
```

Program 7

Write a Python program to find synonyms and antonyms of the word "active" using WordNet.

CODE:

```
import nltk
nltk.download('wordnet')
nltk.download('omw-1.4')
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\Lenovo\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     C:\Users\Lenovo\AppData\Roaming\nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
True
```

```
from nltk.corpus import wordnet

def get_synonyms_antonyms(word):
    synonyms = set()
    antonyms = set()

    for syn in wordnet.synsets(word):
        for lemma in syn.lemmas():
            # Add synonym
            synonyms.add(lemma.name())

            # Check and add antonym if exists
            if lemma.antonyms():
                for ant in lemma.antonyms():
                    antonyms.add(ant.name())

    return synonyms, antonyms
# Test word
word = "active"
synonyms, antonyms = get_synonyms_antonyms(word)

print(f"Synonyms of '{word}':")
print(", ".join(sorted(synonyms)))

print(f"\nAntonyms of '{word}':")
print(", ".join(sorted(antonyms)))
```

Output:

```
Synonyms of 'active':
active, active_agent, active_voice, alive, combat-ready, dynamic, fighting, participating

Antonyms of 'active':
dormant, extinct, inactive, passive, passive_voice, quiet, stative
```

Program-8:

Implement the machine translation application of NLP where it needs to train a machine translation model for a language with limited parallel corpora. Investigate and incorporate techniques to improve performance in low-resource scenarios.

pip install torch

```
import torch
import torch.nn as nn
import torch.optim as optim

# Data & vocab
data = [("hello", "namaste"), ("thank you", "dhanyavaad"), ("bye", "alvida")]

def build_vocab(sentences):
    vocab = {'<pad>': 0, '<sos>': 1, '<eos>': 2}
    for s in sentences:
        for w in s.split():
            if w not in vocab:
                vocab[w] = len(vocab)
    return vocab

SRC = build_vocab([s for s, _ in data])
TGT = build_vocab([t for _, t in data])
IDX2TGT = {i: w for w, i in TGT.items()}

def tensorize(text, vocab):
    return torch.tensor([vocab['<sos>']] + [vocab[w] for w in text.split()] + [vocab['<eos>']])

# Models
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim):
        super().__init__()
        self.emb = nn.Embedding(input_dim, emb_dim)
        self.rnn = nn.GRU(emb_dim, hid_dim, batch_first=True)

    def forward(self, x):
        x = self.emb(x)
        return self.rnn(x)

class Attention(nn.Module):
    def __init__(self, hid_dim):
        super().__init__()
        self.attn = nn.Linear(hid_dim * 2, 1)

    def forward(self, hidden, enc_outs):
        hidden = hidden.permute(1, 0, 2).repeat(1, enc_outs.shape[1], 1)
        energy = torch.tanh(self.attn(torch.cat((hidden, enc_outs), dim=2)))
```

```
        weights = torch.softmax(energy.squeeze(2), dim=1)
        return weights

class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim, attention):
        super().__init__()
        self.emb = nn.Embedding(output_dim, emb_dim)
        self.rnn = nn.GRU(emb_dim + hid_dim, hid_dim, batch_first=True)
        self.fc = nn.Linear(hid_dim * 2, output_dim)
        self.attn = attention

    def forward(self, x, hidden, enc_outs):
        x = self.emb(x).unsqueeze(1)
        a = self.attn(hidden, enc_outs).unsqueeze(1)
        c = torch.bmm(a, enc_outs)
        rnn_input = torch.cat((x, c), dim=2)
        output, hidden = self.rnn(rnn_input, hidden)
        out = self.fc(torch.cat((output.squeeze(1), c.squeeze(1)), dim=1))
        return out, hidden

# Init
E = Encoder(len(SRC), 16, 32)
A = Attention(32)
D = Decoder(len(TGT), 16, 32, A)
optE = optim.Adam(E.parameters(), lr=1e-2)
optD = optim.Adam(D.parameters(), lr=1e-2)
loss_fn = nn.CrossEntropyLoss()

# Training
for epoch in range(100):
    for src, tgt in data:
        se = tensorize(src, SRC).unsqueeze(0)
        te = tensorize(tgt, TGT)
        enc_out, hidden = E(se)
        loss = 0
        x = te[0]
        for i in range(1, len(te)):
            output, hidden = D(torch.tensor([x]), hidden, enc_out)
            loss += loss_fn(output, te[i].unsqueeze(0))
            x = te[i]
        optE.zero_grad()
        optD.zero_grad()
        loss.backward()
        optE.step()
        optD.step()

# Translate
def translate(text):
    se = tensorize(text, SRC).unsqueeze(0)
    enc_out, hidden = E(se)
```

```
    x = torch.tensor([TGT['<sos>']])
    result = []
    for _ in range(10):
        o, hidden = D(x, hidden, enc_out)
        x = o.argmax(1)
        if x == TGT['<eos>']:
            break
        result.append(IDX2TGT[x.item()])
    return ' '.join(result)

print(translate("hello"))  # ➜ "namaste"
print(translate(" thank you"))  # ➜ "dhanyavaad"
```

Output:

```
namaste
dhanyavaad
```