NATURAL LANGUAGE PROCESSING LABORATORY

(21AI62)

LABORATORY MANUAL

VI Semester B.E.

(Academic Year: 2023-24)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
(Artificial Intelligence & Machine Learning)

SAHYADRI
College of Engineering & Management
Adyar, Mangaluru - 575007

Vision

To be a premier institution in Technology and Management by fostering excellence ineducation, innovation, incubation and values to inspire and empower the young minds.

Mission

M1. Creating an academic ambience to impart holistic education focusing on individual growth, integrity, ethical values and social responsibility.

M2. Develop skill based learning through industry-institution interaction to enhance competency and promote entrepreneurship.

M3. Fostering innovation and creativity through competitive environment with state-of-the-art infrastructure.

Program Outcomes:

PO1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Learning Objectives:

CO No.	Course Outcome Description	Bloom's Taxonomy Level
CO1	Discuss the concepts of NLP and demonstrate the statistical-based language models and smoothing techniques	CL3
CO2	Demonstrate the use of morphological analysis and parsing using Finite State Transducers, spelling error detection and correction, parts of speech tagging, context-free grammar, and different parsing approaches.	CL3
CO3	Apply the Naïve Bayes classifier and sentiment analysis for Natural language problems and text classifications.	CL3
CO4	Illustrate the use of Information Retrieval in the context of NLP and understand the concept of lexical semantics, lexical dictionaries such as WordNet, lexical computational semantics, distributional word similarity.	CL3
CO5	Develop the Machine Translation applications using Encoder and Decoder model.	CL3

CONTENTS

Exp. No.	Experiment Description
1	Consider the following Corpus of three sentences a. There is a big garden. b. Children play in a garden c. They play inside beautiful garden Calculate P for the sentence “They play in a big Garden” assuming a bi- gram language model.
2	Find the bigram count for the given corpus. Apply Laplace smoothing and find the bigram probabilities after add-one smoothing (up to 4 decimal places)
3	Implement rule-based tagger and stochastic tagger for the give corpus of sentences.
4	Implement top-down and bottom-up parsing using python NLTK.
5	Given the following short movie reviews, each labeled with a genre, either comedy or action: a. fun, couple, love, love : **comedy** b. fast, furious, shoot : **action** c. couple, fly, fast, fun, fun :**comedy** d. furious, shoot, shoot, fun :**action** e. 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	The dataset contains following 5 documents. D1: "Shipment of gold damaged in a fire" D2: "Delivery of silver arrived in a silver truck" D3: "Shipment of gold arrived in a truck" D4: “Purchased silver and gold arrived in a wooden truck” D5: “The arrival of gold and silver shipment is delayed.” Find the top two relevant documents for the query document with the content “gold silver truck " using the vector space model. Use the following similarity measure and analyze the result. a. Euclidean distance b. Manhattan distance c. Cosine similarity
7	The dataset contains following 4 documents. D1: " It is going to rain today " D2: " Today Rama is not going outside to watch rain" D3: “I am going to watch the movie tomorrow with Rama" D4: “Tomorrow Rama is going to watch the rain at sea shore " Find the top two relevant documents for the query document with the content “Rama watching the rain " using the latent semantic space model. Use the following similarity measure and show the result analysis using bar chart.

	a. Euclidean distance b. Cosine similarity c. Jaccard similarity d. Dice Similarity Coefficient.
8	Extract Synonyms and Antonyms for a given word using WordNet.
9	Implement a machine translator for 10 words using encoder-decoder model for any two languages.

EXPERIMENT DETAILS

1. **Consider the following Corpus of three sentences**
    - **a. There is a big garden.**
    - **b. Children play in a garden**
    - **c. They play inside beautiful garden**

   Calculate P for the sentence “They play in a big Garden” assuming a bi- gram language model.

   Code :

```
#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <map>
using namespace std;

int main() {
    string corpus[10], test;
    int count;
    float probability = 1.0; //initial probability

    cout << "Enter the number of sentences in the corpus: ";
    cin >> count;
    cin.ignore();
    cout << "Enter the sentences for the corpus:" << endl;
    for (int i = 0; i < count; i++) {
        cout << "Sentence " << i+1 << ": ";
        getline(cin, corpus[i]);
        corpus[i] = "<s> " + corpus[i] + " </s>";// start and end with tags
    }

    cout << "Enter the test sentence: ";
    getline(cin, test);
    stringstream test_ss("<s> " + test + " </s>");

    string word;
```

```
    vector<string> test_words;// vector of words from test sentences
    while (test_ss >> word) test_words.push_back(word);

    vector<string> words;//vector of words for corpus
    map<string, int> unigram;//unigram
    map<pair<string, string>, int> bigram;//bigram
    for (int i = 0; i < count; i++) {
        stringstream ss(corpus[i]);
        string prev_word = "<s>", word;//previous tag ie beginning of sentence
        while (ss >> word) {
            words.push_back(word);
            unigram[word]++;//unigram
            bigram[{prev_word, word}]++;//bigram
            prev_word = word;//prev word for bigrams
        }
    }

    cout << "\nUnigram:" << endl;//display unigrams
    for (const auto& word : test_words)
        if (unigram.find(word) != unigram.end())
            cout << word << ": " << unigram[word] << endl;

    cout << "\nBigram:" << endl;//display bigrams
    for (int i = 0; i < test_words.size() - 1; i++) {
        auto current_bigram = make_pair(test_words[i], test_words[i + 1]);//current and next
word pairing for test sentence
        if (bigram.find(current_bigram) != bigram.end())//traverse till end.
            cout << "(" << current_bigram.first << ", " << current_bigram.second << "): " <<
bigram[current_bigram] << endl;
    }

    for (int i = 0; i < test_words.size() - 1; i++) {//calculate probability
        if (bigram.find({test_words[i], test_words[i + 1]}) != bigram.end()) {
            probability *= (float)bigram[{test_words[i], test_words[i + 1]}] /
```

```
unigram[test_words[i]];//probability calculation
        }
    }

    cout << "\nProbability:" <<probability<< endl;
    return 0;
}
```

2. **Find the bigram count for the given corpus. Apply Laplace smoothing and find the bigram probabilities after add-one smoothing (up to 4 decimal places)**

Code :

```
#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <map>
using namespace std;

int main() {
    int count;
    cout << "Enter the number of sentences in the corpus: ";
    cin >> count;
    cin.ignore();

    vector<string> corpus(count);//dataset (corpus)
    cout << "Enter the sentences for the corpus:" << endl;
    for (int i = 0; i < count; i++) {
        cout << "Sentence " << i + 1 << ": ";
        getline(cin, corpus[i]);
    }

    cout << "Enter the test sentence: ";
    string test;//test sentence
    getline(cin, test);

    stringstream test_ss(test);
    vector<string> test_words;//set of words in test sentence
    string word;
    while (test_ss >> word) test_words.push_back(word);//tokenisation

    map<string, int> unigram;
    map<pair<string, string>, int> bigram;
```

```
    for (const auto& sentence : corpus) {
        stringstream ss(sentence);
        string prev_word;
        ss >> prev_word; // Read the first word
        unigram[prev_word]++; // Count the first word of each sentence
        string current_word;
        while (ss >> current_word) {
            unigram[current_word]++;//unigram addition
            bigram[{prev_word, current_word}]++;//bigram addition
            prev_word = current_word;//prev word for the bigram
        }
    }

    // Compute vocab size
    int vocab_size = unigram.size();

    // Display unigram counts
    cout << "\nUnigram Counts:" << endl;
    for (const auto& entry : unigram) {
        cout << entry.first << ": " << entry.second << endl;
    }

    // Display bigram counts
    cout << "\nBigram Counts:" << endl;
    for (const auto& entry : bigram) {
        cout << "(" << entry.first.first << ", " << entry.first.second << "): " << entry.second <<
endl;
    }

    // Compute probabilities for test sentence
    float probability = 1.0;
    cout << "\nBigram Probabilities:" << endl;
    for (size_t i = 0; i < test_words.size() - 1; i++) {
```

```
        string w1 = test_words[i];
        string w2 = test_words[i + 1];
        int bigram_count = bigram[{w1, w2}];
        int unigram_count = unigram[w1];
        float bigram_probability = (float)(bigram_count + 1) / (unigram_count + vocab_size);
        probability *= bigram_probability;
        cout << "(" << w1 << ", " << w2 << "): " << bigram_probability << endl;
    }
    cout << "\nOverall Probability: " << probability << endl;
    return 0;
}
```

3. Implement rule-based tagger and stochastic tagger for the give corpus of sentences.

a)

Code :

```
#include <iostream>
#include <string>
#include <cstring>
#include <cctype>

using namespace std;

//word tags definition
string nouns[] = {"cat", "mat", "bat", "rat", "dad", "well"};
string verbs[] = {"sit", "sat", "run", "ran"};
string dets[] = {"a", "an", "the", "this"};
string adverb[]={"quickly","wisely","very"};
string prep[]={"on","at","in","from","is"};

//applying rules for taggers
void getTag(char word[], char tag[]) {
    int len = strlen(word);
    for (const string &noun : nouns) {
        if (strcmp(word, noun.c_str()) == 0) {//if noun
            strcpy(tag, "NN");
            return;
        }
    }
    for (const string &prp : prep) {
        if (strcmp(word, prp.c_str()) == 0) {//if preposition
            strcpy(tag, "PRP");
            return;
        }
    }
    for (const string &verb : verbs) {
        if (strcmp(word, verb.c_str()) == 0) {//if verb
```

```
            strcpy(tag, "VB");
            return;
        }
    }
    for (const string &det : dets) {
        if (strcmp(word, det.c_str()) == 0) {//if determiner
            strcpy(tag, "DET");
            return;
        }
    }
    for (const string &det : adverb) {
        if (strcmp(word, det.c_str()) == 0) {//if adverb
            strcpy(tag, "JJ");
            return;
        }
    }
    if (len >= 3 && strcmp(&word[len - 3], "ing") == 0) {//ends with ing
        strcpy(tag, "VBG");
    } else if (len >= 2 && strcmp(&word[len - 2], "ed") == 0) {//ends with ed
        strcpy(tag, "VBD");
    } else if (len >=1 && strcmp(&word[len - 1], "s") == 0) {//ends with s
        strcpy(tag, "NNS");
    } else if (len >= 2 && strcmp(&word[len - 2], "ly") == 0) {//ends with ly
        strcpy(tag, "RB");
    } else if (len >= 4 && strcmp(&word[len - 4], "able") == 0) {//ends with able
        strcpy(tag, "JJ");
    } else
    strcpy(tag, "NN");//else set as noun
}

void toLowerCase(char *str) {//lower case conversion
    for (int i = 0; str[i]; i++) {
        str[i] = tolower(str[i]);
    }
```

```
}

int main() {
    char sentence[] = "the cat is very fat sleeping on the mat";//test sentence
    char taggedSentence[1000] = "";//tagged sentence
    char word[50];
    char tag[10];
    char *token = strtok(sentence, " .");//tokenisation
    while (token != NULL) {
        strcpy(word, token);
        toLowerCase(word);
        getTag(word, tag);//get tag from rules
        char taggedWord[60];
        sprintf(taggedWord, "%s/%s ", word, tag);//assign tag to word in sentence
        if (strlen(taggedSentence) + strlen(taggedWord) < sizeof(taggedSentence)) {
            strcat(taggedSentence, taggedWord);//check for bit errors
        } else {
            cout << "Tagged sentence exceeds buffer size." << endl;
            return 1;
        }
        token = strtok(NULL, " .");//next token
    }

    cout << "Tagged Sentence:\n" << taggedSentence << endl;

    return 0;
}
```

3b)

Code :

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <string>

using namespace std;

typedef unordered_map<string, int> StringIntMap;//unigram map
typedef vector<pair<string, string>> TaggedSentence;//bigram map

struct NaiveBayesModel {//define naive bayes model
    StringIntMap tagCounts;
    unordered_map<string, StringIntMap> wordTagCounts;
    int totalTags;
};

void trainNaiveBayes(NaiveBayesModel& model, const vector<TaggedSentence>&
taggedSentences) {//training function
    model.totalTags = 0;//initial tags =0
    for (const auto& sentence : taggedSentences) {
        for (const auto& wordTagPair : sentence) {
            const string& word = wordTagPair.first;//word
            const string& tag = wordTagPair.second;//tag
            model.tagCounts[tag]++;//tag increment
            model.wordTagCounts[tag][word]++;//word with tag occurence
            model.totalTags++;//total tags increment
        }
    }
}

vector<string> tagSentenceNaiveBayes(const NaiveBayesModel& model, const
vector<string>& sentence) {//testing function
```

```
    vector<string> tags(sentence.size());
    for (size_t i = 0; i < sentence.size(); ++i) {
        const string& word = sentence[i];
        double maxProb = -1;//initially -1(invalid)
        string bestTag;
        for (const auto& tagCount : model.tagCounts) {
            const string& tag = tagCount.first;
            int wordCount = 0;
            auto tagIt = model.wordTagCounts.find(tag);//get tag
            if (tagIt != model.wordTagCounts.end()) {//best match or not
                auto wordIt = tagIt->second.find(word);
                if (wordIt != tagIt->second.end()) {
                    wordCount = wordIt->second;
                }
            }
            double wordGivenTagProb = static_cast<double>(wordCount + 1) /
                                      (tagCount.second +
model.wordTagCounts.at(tag).size());//p(word|tag)
            double tagProb = static_cast<double>(tagCount.second) / model.totalTags;
            double prob = wordGivenTagProb * tagProb;//multiply
            if (prob > maxProb) {//if better, then update
                maxProb = prob;
                bestTag = tag;
            }
        }
        tags[i] = bestTag;
    }
    return tags;
}

int main() {
    vector<TaggedSentence> taggedSentences = {//dataset
        {{"John", "NNP"}, {"saw", "VBD"}, {"the", "DT"}, {"dog", "NN"}},
        {{"Mary", "NNP"}, {"walked", "VBD"}, {"to", "TO"}, {"the", "DT"}, {"park", "NN"}}
```

```
    };

    NaiveBayesModel model;//create model
    trainNaiveBayes(model, taggedSentences);//training phase

    vector<string> sentence = {"John", "walked", "to", "the", "park"};//test sentence
    vector<string> tags = tagSentenceNaiveBayes(model, sentence);//testing phse

    for (size_t i = 0; i < sentence.size(); ++i) {//output
        cout << sentence[i] << "/" << tags[i] << " ";
    }
    cout << endl;

    return 0;
}
```

4. **Implement top-down and bottom-up parsing using python NLTK.**

Code :

```
import nltk
from nltk import CFG
from nltk.tree import Tree

#defining the grammer
grammar = CFG.fromstring("""
    S -> NP VP
    VP -> V NP | V NP PP
    PP -> P NP
    V -> "saw" | "ate" | "walked"
    NP -> "Rahil"  | "Bob" | Det N | Det N PP
    Det -> "a" | "an" | "the" | "my"|"his"
    N -> "dog" | "cat" | "telescope" | "park"| "Moon"| "terrace"
    P -> "in" | "on" | "by" | "with"| "from"
""")

#define a proper sentence
sentence = "Rahil saw the Moon with the telescope from his terrace".split()

#bottom-up parser
print("Bottom-Up Parsing:")
bottom_up_parser = nltk.ChartParser(grammar)
bottom_up_trees = []
for tree in bottom_up_parser.parse(sentence):
    print(tree)
    tree.pretty_print()
    bottom_up_trees.append(tree)
if bottom_up_trees:#if parsing is possible, then print
    for tree in bottom_up_trees:
        tree.draw()
```

```
print("Top-Down Parsing:")
top_down_parser = nltk.RecursiveDescentParser(grammar)
top_down_trees = []
try:
    for tree in top_down_parser.parse(sentence):
        print(tree)
        tree.pretty_print()
        top_down_trees.append(tree)
except ValueError as e:
    print(f"Error in parsing: {e}")
if top_down_trees:#if parsing is possible, then print
    for tree in top_down_trees:
        tree.draw()
```

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

a. fun, couple, love, love : comedy

b. fast, furious, shoot : action

c. couple, fly, fast, fun, fun :comedy

d. furious, shoot, shoot, fun :action

e. 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 defaultdict, Counter

import math


reviews = [

    ("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")

]#define reviews


D = "fast, couple, shoot, fly" #the query


def tokenize(text):

    return text.split(", ") #split into words


class_docs = defaultdict(list)#tokens based on class

vocabulary = set()#vocab set (no repeatations).

class_count=defaultdict(int)


for review, catgory in reviews: #classifying into classes

    tokens = tokenize(review)#tokenisation

    class_docs[catgory].extend(tokens)#add tokens to respective class

    class_count[catgory] += 1 #increment the class count

    vocabulary.update(tokens)#update the vocab set
```

```
vocab_size = len(vocabulary)
total_docs = len(reviews)
priors = {catgory: count / total_docs for catgory,count in class_count.items()} #prior
probability calculation for each class

likelihoods = {}
for catgory, tokens in class_docs.items(): #calculating likelihood probability for each class
    token_counts = Counter(tokens)
    total_words = len(tokens)
    likelihoods[catgory] = {word: (token_counts[word] + 1) / (total_words + vocab_size) for
word in vocabulary} # laplace smoothing

tokens = tokenize(D)#tokenise the query document
posteriors = {}

for catgory in priors:#calculating posterior probability for each class
    log_prob = (priors[catgory])
    for token in tokens:
        log_prob *= (likelihoods[catgory].get(token, 1 / (len(class_docs[catgory]) +
vocab_size)))
    posteriors[catgory] = log_prob

most_likely_class = max(posteriors, key=posteriors.get)
print('Posterior Probability:', posteriors)
print(f"The most likely class for the document '{D}' is: {most_likely_class}")
```

6. **The dataset contains following 5 documents.**
   D1: "Shipment of gold damaged in a fire"
   D2: "Delivery of silver arrived in a silver truck" D3: "Shipment of gold arrived in a truck"
   D4: "Purchased silver and gold arrived in a wooden truck" D5: "The arrival of gold and silver shipment is delayed."
   Find the top two relevant documents for the query document with the content "gold silver truck " using the vector space model.
   Use the following similarity measure and analyze the result.

   a. **Euclidean distance**
   b. **Manhattan distance**
   c. **Cosine similarity**

Code :

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from scipy.spatial.distance import euclidean, cityblock
from sklearn.metrics.pairwise import cosine_similarity

documents = [
    "Shipment of gold damaged in a fire",
    "Delivery of silver arrived in a silver truck",
    "Shipment of gold arrived in a truck",
    "Purchased silver and gold arrived in a wooden truck",
    "The arrival of gold and silver shipment is delayed."
]#Document set

query = "gold silver truck" #query sentence

vectorizer = CountVectorizer(stop_words="english")#initialise the tdf with no matrix
X = vectorizer.fit_transform(documents + [query])#push the document and query to tdf
vectors = X.toarray()#make it in array form
doc_vectors = vectors[:-1]#docs only
query_vector = vectors[-1]#query only

def compute_distances(doc_vectors, query_vector):#distance calculations
    euclidean_distances = [euclidean(doc, query_vector) for doc in doc_vectors]
    manhattan_distances = [cityblock(doc, query_vector) for doc in doc_vectors]
```

```
    cosine_similarities = cosine_similarity(doc_vectors, query_vector.reshape(1, -1)).flatten()
    return euclidean_distances, manhattan_distances, cosine_similarities

euclidean_distances, manhattan_distances, cosine_similarities =
compute_distances(doc_vectors, query_vector)

euclidean_ranking = np.argsort(euclidean_distances)#more distance, unlikely related
manhattan_ranking = np.argsort(manhattan_distances)#more distance, unlikely related
cosine_ranking = np.argsort(-cosine_similarities)#more cosine value, likely related

#best 2 docs (+1 since indexing is from 0)
top_2_euclidean = euclidean_ranking[:2]+1
top_2_manhattan = manhattan_ranking[:2]+1
top_2_cosine = cosine_ranking[:2]+1

print("Euclidean Distance:",euclidean_distances)
print("Manhattan Distance:",manhattan_distances)
print("Cosine Similarity:",cosine_similarities)

print("\nTop 2 documents using Euclidean distance:", top_2_euclidean)
print("Top 2 documents using Manhattan distance:", top_2_manhattan)
print("Top 2 documents using Cosine similarity:", top_2_cosine)
```

7. The dataset contains following 4 documents.

D1: " It is going to rain today "

D2: " Today Rama is not going outside to watch rain"

D3: "I am going to watch the movie tomorrow with Rama" D4: "Tomorrow Rama is going to watch the rain at sea shore "

Find the top two relevant documents for the query document with the content "Rama watching the rain " using the latent semantic space model.

Use the following similarity measure and show the result analysis using bar chart.

- **a. Euclidean distance**
- **b. Cosine similarity**
- **c. Jaccard similarity**
- **d. Dice Similarity Coefficient.**

Code :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD
from scipy.spatial.distance import euclidean, jaccard
from sklearn.metrics.pairwise import cosine_similarity

documents = [
    "It is going to rain today",
    "Today Rama is not going outside to watch rain",
    "I am going to watch the movie tomorrow with Rama",
    "Tomorrow Rama is going to watch the rain at sea shore"
]

query = "Rama watching the rain"

vectorizer = TfidfVectorizer(stop_words='english')
X_docs = vectorizer.fit_transform(documents).toarray()
X_query = vectorizer.transform([query]).toarray()

lsa = TruncatedSVD(n_components=4)
X_docs_lsa = lsa.fit_transform(X_docs)
X_query_lsa = lsa.transform(X_query)
```

```
def compute_similarity_measures(doc_vectors, query_vector):
    euclidean_distances = [euclidean(doc, query_vector) for doc in doc_vectors]
    cosine_similarities = cosine_similarity(doc_vectors, query_vector.reshape(1, -1)).flatten()

    jaccard_similarities = []
    dice_similarities = []

    for doc in doc_vectors:
        doc_binary = np.array(doc > 0, dtype=int)
        query_binary = np.array(query_vector > 0, dtype=int)

        jaccard_sim = 1 - jaccard(doc_binary, query_binary)
        dice_sim = 2 * np.sum(doc_binary & query_binary) / (np.sum(doc_binary) +
np.sum(query_binary))

        jaccard_similarities.append(jaccard_sim)
        dice_similarities.append(dice_sim)

    return euclidean_distances, cosine_similarities, jaccard_similarities, dice_similarities

euclidean_distances, cosine_similarities, jaccard_similarities, dice_similarities =
compute_similarity_measures(X_docs_lsa, X_query_lsa[0])

euclidean_ranking = np.argsort(euclidean_distances)
cosine_ranking = np.argsort(-cosine_similarities)
jaccard_ranking = np.argsort(-np.array(jaccard_similarities))
dice_ranking = np.argsort(-np.array(dice_similarities))

def plot_rankings(distances, measure_names):
    plt.figure(figsize=(12, 8))
    colors = ['b', 'g', 'r', 'c']
    bar_width = 0.2
    positions = np.arange(len(distances[0]))
```

```
    for i, dist in enumerate(distances):
        plt.bar(positions + i * bar_width, dist , bar_width, label=measure_names[i],
color=colors[i])

    plt.xlabel('Documents')
    plt.ylabel('Similarities')
    plt.title('Documents Comparison')
    plt.xticks(positions + bar_width, [f'D{i+1}' for i in range(len(distances[0]))])
    plt.legend()
    plt.tight_layout()
    plt.show()

print("Top 2 documents using Euclidean distance:", euclidean_ranking[:2] + 1)
print("Top 2 documents using Cosine similarity:", cosine_ranking[:2] + 1)
print("Top 2 documents using Jaccard similarity:", jaccard_ranking[:2] + 1)
print("Top 2 documents using Dice similarity coefficient:", dice_ranking[:2] + 1)

print("Euclidean distance:", euclidean_distances)
print("Cosine similarity:", cosine_similarities)
print("Jaccard similarity:", jaccard_similarities)
print("Dice similarity coefficient:", dice_similarities)

measure_names = ["Euclidean Distance", "Cosine Similarity", "Jaccard Similarity", "Dice
Similarity"]
dist = [euclidean_distances, cosine_similarities, jaccard_similarities, dice_similarities]

plot_rankings(dist, measure_names)
```

8. Extract Synonyms and Antonyms for a given word using WordNet.

Code :

```
import nltk
from nltk.corpus import wordnet

# Download WordNet data
nltk.download('wordnet')

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

    for syn in wordnet.synsets(word):# for all words in synset
        for lemma in syn.lemmas():
            synonyms.add(lemma.name())#append synonyms
            if lemma.antonyms():#if antonym exist
                for ant in lemma.antonyms():
                    antonyms.add(ant.name())#append all antonyms

    return synonyms, antonyms

word = input("Enter the word to get synonym and antonym: ")
synonyms, antonyms = get_synonyms_antonyms(word)
print(f"Synonyms of '{word}': {synonyms}")
print(f"Antonyms of '{word}': {antonyms}")
```

9. Implement a machine translator for 10 words using encoder-decoder model for any two languages.

Code :

```
import os
import sys
import transformers
import tensorflow as tf
from datasets import load_dataset
from transformers import AutoTokenizer
from transformers import TFAutoModelForSeq2SeqLM, DataCollatorForSeq2Seq
from transformers import AdamWeightDecay
from transformers import AutoTokenizer, TFAutoModelForSeq2SeqLM

model_checkpoint = "Helsinki-NLP/opus-mt-en-hi"

raw_datasets = load_dataset("cfilt/iitb-english-hindi")

print(raw_datasets)

print(raw_datasets['train'][1])

tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)

print(tokenizer("Hello, this is a sentence!"))

max_input_length = 128
max_target_length = 128

source_lang = "en"
target_lang = "hi"


def preprocess_function(examples):
    inputs = [ex[source_lang] for ex in examples["translation"]]
```

```
    targets = [ex[target_lang] for ex in examples["translation"]]
    model_inputs = tokenizer(inputs, max_length=max_input_length, truncation=True)

    # Setup the tokenizer for targets
    with tokenizer.as_target_tokenizer():
        labels = tokenizer(targets, max_length=max_target_length, truncation=True)

    model_inputs["labels"] = labels["input_ids"]
    return model_inputs

preprocess_function(raw_datasets["train"][:2])
tokenized_datasets = raw_datasets.map(preprocess_function, batched=True)
model = TFAutoModelForSeq2SeqLM.from_pretrained(model_checkpoint)

batch_size = 16
learning_rate = 2e-5
weight_decay = 0.01
num_train_epochs = 1
data_collator = DataCollatorForSeq2Seq(tokenizer, model=model, return_tensors="tf")
generation_data_collator = DataCollatorForSeq2Seq(tokenizer, model=model,
return_tensors="tf", pad_to_multiple_of=128)
train_dataset = model.prepare_tf_dataset(
    tokenized_datasets["test"],
    batch_size=batch_size,
    shuffle=True,
    collate_fn=data_collator,
)
validation_dataset = model.prepare_tf_dataset(
    tokenized_datasets["validation"],
    batch_size=batch_size,
    shuffle=False,
    collate_fn=data_collator,
)
generation_dataset = model.prepare_tf_dataset(
```

```
    tokenized_datasets["validation"],
    batch_size=8,
    shuffle=False,
    collate_fn=generation_data_collator,
)
optimizer = AdamWeightDecay(learning_rate=learning_rate,
weight_decay_rate=weight_decay)
model.compile(optimizer=optimizer)

model.fit(train_dataset, validation_data=validation_dataset, epochs=1)
model.save_pretrained("tf_model/")

tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
model = TFAutoModelForSeq2SeqLM.from_pretrained("tf_model/")

input_text  = "hello India"

tokenized = tokenizer([input_text], return_tensors='np')
out = model.generate(**tokenized, max_length=128)
print(out)

with tokenizer.as_target_tokenizer():
    print(tokenizer.decode(out[0], skip_special_tokens=True))
```