

Master of Science (Five Year Integrated) in
Computer Science (Artificial Intelligence & Data
Science)

Seventh Semester

Laboratory Record

21-805-0704: Computational Linguistics Lab

*Submitted in partial fulfillment
of the requirements for the award of degree in
Master of Science (Five Year Integrated)
in Computer Science (Artificial Intelligence & Data Science) of
Cochin University of Science and Technology (CUSAT)
Kochi*



Submitted by

ABHIN P.T
(80521002)

DEPARTMENT OF COMPUTER SCIENCE
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY (CUSAT)
KOCHI-682022

DECEMBER 2023

DEPARTMENT OF COMPUTER SCIENCE
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY (CUSAT)
KOCHI, KERALA-682022



*This is to certify that the software laboratory record for **21-805-0704: Computational Linguistics Lab** is a record of work carried out by **ABHIN P.T (80521002)**, in partial fulfillment of the requirements for the award of degree in **Master of Science (Five Year Integrated) in Computer Science (Artificial Intelligence & Data Science)** of Cochin University of Science and Technology (CUSAT), Kochi. The lab record has been approved as it satisfies the academic requirements in respect of the fifth semester laboratory prescribed for the Master of Science (Five Year Integrated) in Computer Science degree.*

Faculty Member in-charge

Dr. Jeena K
Assistant Professor
Department of Computer Science
CUSAT

Dr. Madhu S. Nair
Professor and Head
Department of Computer Science
CUSAT

Table of Contents

Sl.No.	Program	Pg.No.
1	Text Tokenizer	pg 1
2	Finite State Automata	pg 3
3	Finite State Transducer	pg 5
4	Minimum Edit Distance	pg 7
5	Spell Checker	pg 9
6	Sentiment Analysis	pg 12
7	POS Tagging	pg 15
8	Bigram Probability	pg 20
9	TF-IDF Matrix	pg 22
10	PPMI Matrix	pg 26
11	Naive Bayes Classifier	pg 30

Text Tokenizer

AIM

Implement a simple rule-based Text tokenizer for the English language using regular expressions. Your tokenizer should consider punctuations and special symbols as separate tokens. Contractions like "isn't" should be regarded as 2 tokens - "is" and "n't". Also identify abbreviations (eg, U.S.A) and internal hyphenation (eg. ice-cream), as single tokens.

PROGRAM

```
import re

def tokenize(text):
    # Updated pattern to match abbreviations, contractions, words,
    # and special symbols.
    pattern = r'''
        \b(?:[A-Za-z]\.){2,}[A-Za-z]?    # Match abbreviations like
        U.S.A
        | \b\w+\'\w+\b                  # Match contractions like isn't, can't
        | \b\w+(?:-\w+)\b                # Match hyphenated words
        like ice-cream
        | \b\w+\b                        # Match regular words
        | [.,!?:;'"(){}\[ \]<>@#$$%^&*~+=~'|\\/] # Match
        individual punctuation and special characters
    '''

    # Compile pattern with VERBOSE flag to allow comments
    tokenizer = re.compile(pattern, re.VERBOSE)

    # Find all tokens based on the pattern
    tokens = tokenizer.findall(text)

    # Use a set to eliminate duplicate tokens
    unique_tokens = set()

    # Post-process to further split contractions and hyphenated words if needed
    for token in tokens:
        # Split contractions like "isn't" into 'is' and "n't"
        if re.match(r"\b\w+'(?:t|ll|re|ve|d|m|s)\b", token): #
            # Matches common English contractions
            unique_tokens.add(token[:-3] if token.endswith("n't")
```

```
        else token[:-2]) # Base word
        unique_tokens.add(token[-3:] if token.endswith("n't")
        else token[-2:]) # Contraction part
    elif '-' in token and not token.startswith('-') and
    not token.endswith('-'): # Split hyphenated words
        split_parts = token.split('-')
        for part in split_parts[:-1]:
            unique_tokens.add(part)
            unique_tokens.add('-')
        unique_tokens.add(split_parts[-1]) # Add the
        last part without adding another hyphen
    else:
        unique_tokens.add(token)

# Convert set back to list and sort for consistent order
unique_tokens_list = sorted(unique_tokens)

# Create a list of lists for the table
table = [[i, unique_tokens_list[i]] for i in range(len(unique_tokens_list))]
return table

# Example usage
text = "The U.K. has world-class museums, doesn't it?
I can't wait to visit! Ice-cream and fun-filled adventures await."
tokens_table = tokenize(text)

# Print the table
print("Index\tToken")
for index, token in tokens_table:
    print(f"{index}\t{token}")
```

SAMPLE INPUT-OUTPUT

Index	Token
0	!
1	,
2	-
3	.
4	?
5	I
6	Ice
7	The
8	U.K.
9	adventures
10	and
11	await
12	ca
13	class
14	cream
15	does
16	filled
17	fun
18	has
19	it
20	museums
21	n't
22	to
23	visit
24	wait
25	world

Finite State Automata

AIM

Design and implement a Finite State Automata(FSA) that accepts English plural nouns ending with the character 'y', e.g. boys, toys, ponies, skies, and puppies but not boies or toies or ponys. (Hint: Words that end with a vowel followed by 'y' are appended with 's' and will not be replaced with "ies" in their plural form).

PROGRAM

```
def is_plural_noun_accepted_fsa(word):
    # Check if the word is too short or doesn't end with 's'
    if len(word) < 2 or word[-1] != 's':
        return False

    # Reverse the word for processing
    word = word[::-1]
    state = 'S1'

    for char in word[1:]:
        if state == 'S1':
            if char == 'y':
                state = 'S2'
            elif char == 'e':
                state = 'S3'
            else:
                return False
        elif state == 'S2':
            if char in 'aeiou':
                state = 'S5'
            else:
                return False
        elif state == 'S3':
            if char == 'i':
                state = 'S4'
            else:
                return False
        elif state == 'S4':
            if char.isalpha() and char not in 'aeiou':
                state = 'S6'
            else:
```

```
        return False
    elif state == 'S5':
        continue # Stay in S5 if we encounter a vowel
    elif state == 'S6':
        continue # Stay in S6 if we encounter a consonant

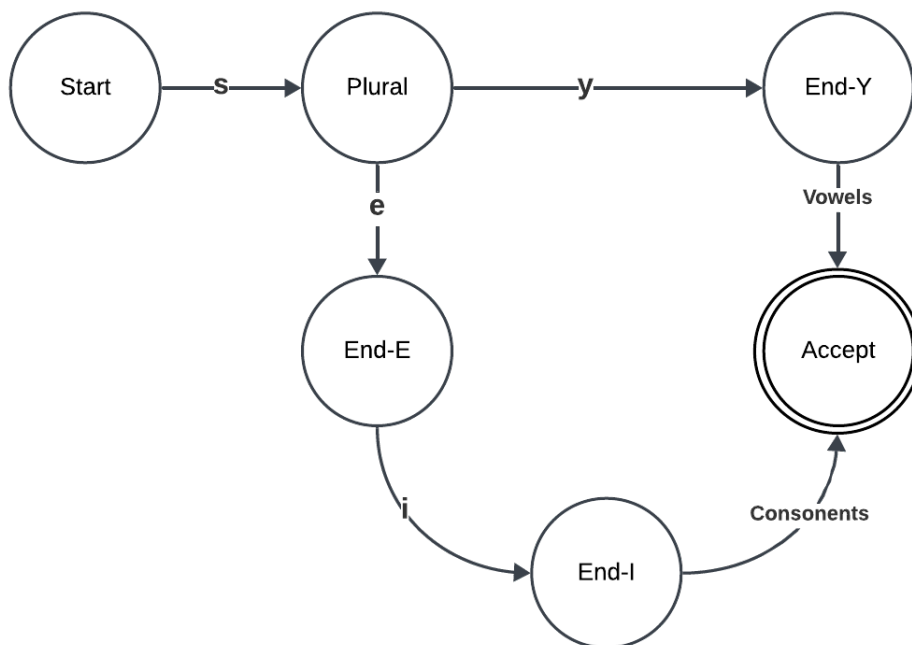
    return True

# List of test words
test_words = ['ladies' , 'boys', 'toys', 'ponies', 'skies', 'puppies', 'boies', 'toies',
              'ponys', 'carries', 'daisies']

# Create a dictionary to store the results of the test words
results = {word: is_plural_noun_accepted_fsa(word) for word in test_words}

# Print the results
print(results)
```

SAMPLE INPUT-OUTPUT



```
{'ladies': True, 'boys': True, 'toys': True, 'ponies': True, 'skies': True,
'puppies': True, 'boies': False, 'toies': False, 'ponys': False, 'carries': True, 'daisies': True}
```


Finite State Transducer

AIM

Design and implement a Finite State Transducer(FST) that accepts lexical forms of English words(e.g. shown below) and generates its corresponding plurals, based on the e-insertion spelling rule

=> e / {x,s,z}^ __ s#^

is the morpheme boundary and - word boundary

Input	Output
fox^s#	foxes
boy^s#	boys

PROGRAM

```
def pluralize(word):
    """
    Pluralizes an English word based on the e-insertion rule:
    - Inserts 'e' before 's' if the word ends with x^, s^, or z^ (indicating a
    morpheme boundary).
    - Otherwise, simply appends 's' to form the plural.

    Args:
    - word (str): The word in lexical form with a morpheme boundary symbol (^) at the
    end of the base word.

    Returns:
    - str: The pluralized form of the word.
    """
    if word.endswith(("x^", "s^", "z^")):
        # Rule: e-insertion for words ending in x, s, z with morpheme boundary (^)
        base_word = word[:-1] # Remove the morpheme boundary (^)
        plural_word = base_word + "es" # Insert 'e' before 's'
    elif word.endswith("^"):
        # For other cases with a morpheme boundary, just add 's'
        plural_word = word[:-1] + "s" # Remove ^ and add 's'
    else:
        # Default case (if ^ is not present, assume it's a regular plural form)
        plural_word = word + "s"
```

```
    return plural_word

# Test cases
test_words = ["fox^s#", "boy^s#"]

for word in test_words:
    print(f"{word} -> {pluralize(word)}")
from graphviz import Digraph

# Initialize the FST diagram
fst = Digraph("Finite State Transducer for Pluralization", format="png")
fst.attr(rankdir="LR") # Left to right orientation

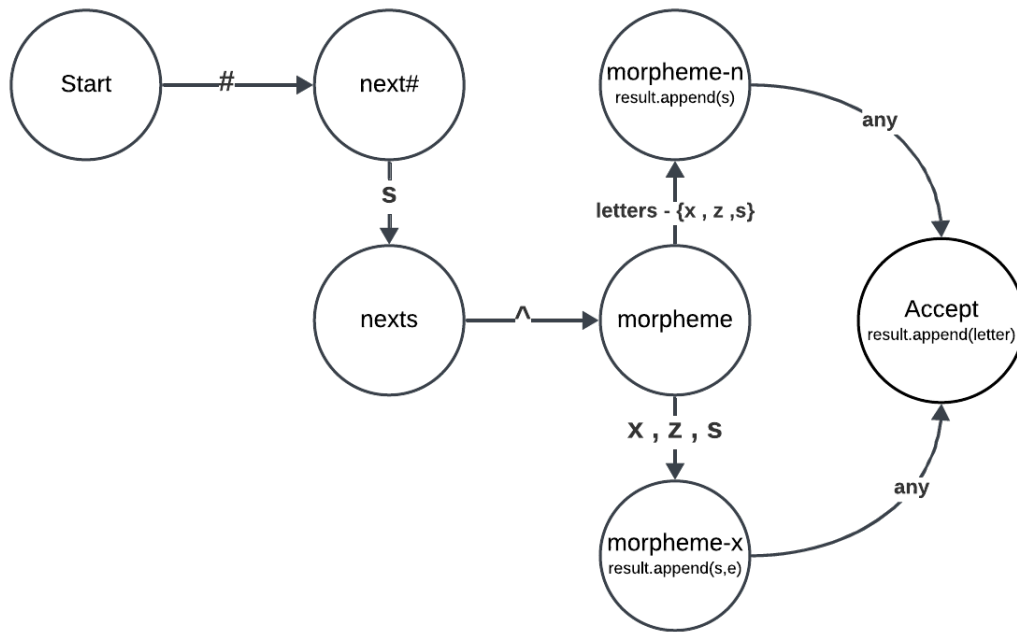
# Define states
fst.node("S0", "Start (S0)")
fst.node("S1", "Suffix Check (S1)")
fst.node("S2", "Insert 'e' (S2)")
fst.node("S3", "Final State (S3)")

# Define transitions
fst.edge("S0", "S1", label="x^, s^, z^") # From Start to Suffix Check
fst.edge("S1", "S2", label="Insert 'e'") # Insert "e" before "s"
fst.edge("S2", "S3", label="Add 's'") # Add "s" to form the plural
fst.edge("S0", "S3", label="Other endings, add 's'") # Direct pluralization for
other cases

# Save and render the diagram
fst.render("pluralization_fst")

print("FST diagram generated and saved as 'pluralization_fst.png'.")
```

SAMPLE INPUT-OUTPUT



fox^s#: foxes
boy^s#: boys

Minimum Edit Distance

AIM

Implement the Minimum Edit Distance algorithm to find the edit distance between any two given strings. Also, list the edit operations.

PROGRAM

```
def min_edit_distance(A, B):
    m = len(A)
    n = len(B)

    # Create a matrix of size (m+1) x (n+1)
    D = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize the first row and column
    for i in range(m + 1):
        D[i][0] = i # Deletion
    for j in range(n + 1):
        D[0][j] = j # Insertion

    # Fill the matrix
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if A[i - 1] == B[j - 1]:
                D[i][j] = D[i - 1][j - 1] # No cost
            else:
                D[i][j] = min(
                    D[i - 1][j] + 1,    # Deletion
                    D[i][j - 1] + 1,    # Insertion
                    D[i - 1][j - 1] + 2 # Substitution
                )

    # Backtrack to find the operations
    operations = []
    i, j = m, n

    while i > 0 or j > 0:
        if i > 0 and j > 0 and A[i - 1] == B[j - 1]:
            i -= 1
            j -= 1
```

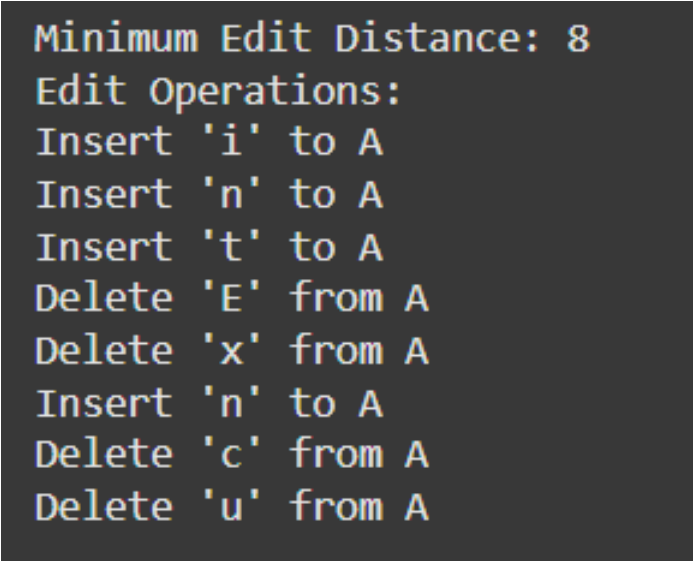
```
elif i > 0 and (j == 0 or D[i][j] == D[i - 1][j] + 1):
    operations.append(f"Delete '{A[i - 1]}' from A")
    i -= 1
elif j > 0 and (i == 0 or D[i][j] == D[i][j - 1] + 1):
    operations.append(f"Insert '{B[j - 1]}' to A")
    j -= 1
else:
    operations.append(f"Substitute '{A[i - 1]}' in A with '{B[j - 1]}'")
    i -= 1
    j -= 1

operations.reverse() # Reverse to get the order of operations from start to finish
return D[m][n], operations

# Example usage
A = "Execution"
B = "intention"
edit_distance, edit_operations = min_edit_distance(A, B)

print("Minimum Edit Distance:", edit_distance)
print("Edit Operations:")
for operation in edit_operations:
    print(operation)
```

SAMPLE INPUT-OUTPUT



```
Minimum Edit Distance: 8
Edit Operations:
Insert 'i' to A
Insert 'n' to A
Insert 't' to A
Delete 'E' from A
Delete 'x' from A
Insert 'n' to A
Delete 'c' from A
Delete 'u' from A
```

Spell Checker

AIM

Design and implement a statistical spell checker for detecting and correcting non-word spelling errors in English, using the bigram language model. Your program should do the following:

1. Tokenize the corpus and create a vocabulary of unique words.
2. Create a bi-gram frequency table for all possible bigrams in the corpus.
3. Scan the given input text to identify the non-word spelling errors
4. Generate the candidate list using 1 edit distance from the misspelled words
5. Suggest the best candidate word by calculating the probability of the given sentence using the bigram LM.

PROGRAM

```
import re
from collections import defaultdict

def tokenize(text):
    # Tokenize text into words using regex for simplicity
    return re.findall(r'\b\w+\b', text.lower())

def build_vocabulary_and_bigrams(corpus):
    words = tokenize(corpus)
    vocabulary = set(words)

    bigram_freq = defaultdict(int)
    for i in range(len(words) - 1):
        bigram = (words[i], words[i+1])
        bigram_freq[bigram] += 1

    print("Vocabulary:", vocabulary) # Print the vocabulary
    print("Bigram Frequency Table:", dict(bigram_freq)) # Print bigram frequency table

    return vocabulary, bigram_freq

def find_misspellings(text, vocabulary):
    words = tokenize(text)
```

```
misspellings = [word for word in words if word not in vocabulary]
print("Identified Misspellings:", misspellings) # Print identified misspellings
return misspellings

def edits1(word):
    # Generate all words one edit away from 'word'
    letters = 'abcdefghijklmnopqrstuvwxyz'
    splits = [(word[:i], word[i:]) for i in range(len(word) + 1)]
    deletes = [L + R[1:] for L, R in splits if R]
    transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R) > 1]
    replaces = [L + c + R[1:] for L, R in splits if R for c in letters]
    inserts = [L + c + R for L, R in splits for c in letters]
    return set(deletes + transposes + replaces + inserts)

def get_candidates(word, vocabulary):
    # Get candidate corrections for the misspelled word
    candidates = {w for w in edits1(word) if w in vocabulary}
    print(f"Candidates for '{word}':", candidates) # Print candidate corrections
    return candidates

def get_multigram_candidates(misspelled_word, words, vocabulary):
    # Generate multi-word candidates by combining single-word candidates
    candidates = set()
    for i in range(len(words)):
        for j in range(i + 1, len(words) + 1):
            candidate = ' '.join(words[i:j])
            if candidate not in vocabulary:
                continue
            candidates.add(candidate)
    return candidates

def bigram_probability(sentence, bigram_freq):
    words = tokenize(sentence)
    bigram_prob = 1.0
    for i in range(len(words) - 1):
        bigram = (words[i], words[i+1])
        # Add a small constant to avoid zero probability
        bigram_prob *= (bigram_freq[bigram] + 1) / sum(bigram_freq.values())
    return bigram_prob

def correct_spelling(text, vocabulary, bigram_freq):
```

```
words = tokenize(text)
misspellings = find_misspellings(text, vocabulary)

corrected_text = words[:]

for misspelled_word in misspellings:
    candidates = get_candidates(misspelled_word, vocabulary)
    multi_candidates = get_multigram_candidates
    (misspelled_word, words, vocabulary)

    # Combine single-word and multi-word candidates
    all_candidates = candidates.union(multi_candidates)

    if all_candidates:
        best_candidate = max(
            all_candidates,
            key=lambda candidate: bigram_probability(
                ' '.join(words).replace(misspelled_word, candidate), bigram_freq
            )
        )
        print(f"Replacing '{misspelled_word}' with '{best_candidate}'") # Print repla
        info
        corrected_text = [best_candidate if w ==
            misspelled_word else w for w in corrected_text]

    return ' '.join(corrected_text)

# Example corpus and test case
corpus = (
    "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. "
    "The only thing we have to fear is fear itself."
)

input_text = "The quik brown fox jumos ovr the lazy dg. A jurney
of a thosand miles begins wth a singel step."

# Building vocabulary and bigram frequency table
vocabulary, bigram_freq = build_vocabulary_and_bigrams(corpus)
```



```
# Correct the input text
corrected_text = correct_spelling(input_text, vocabulary, bigram_freq)
print("Corrected Text:", corrected_text)
```

SAMPLE INPUT-OUTPUT

Vocabulary: 'journey', 'over', 'not', 'only', 'itself', 'question', 'all', 'is', 'gold', 'lazy', 'be', 'that', 'fox', 'quick', 'we', 'dog', 'thousand', 'thing', 'begins', 'or', 'have', 'single', 'the', 'with', 'step', 'brown', 'jumps', 'a', 'of', 'to', 'fear', 'miles', 'glitters' Bigram Frequency Table: ('the', 'quick'): 1, ('quick', 'brown'): 1, ('brown', 'fox'): 1, ('fox', 'jumps'): 1, ('jumps', 'over'): 1, ('over', 'the'): 1, ('the', 'lazy'): 1, ('lazy', 'dog'): 1, ('dog', 'a'): 1, ('a', 'journey'): 1, ('journey', 'of'): 1, ('of', 'a'): 1, ('a', 'thousand'): 1, ('thousand', 'miles'): 1, ('miles', 'begins'): 1, ('begins', 'with'): 1, ('with', 'a'): 1, ('a', 'single'): 1, ('single', 'step'): 1, ('step', 'to'): 1, ('to', 'be'): 2, ('be', 'or'): 1, ('or', 'not'): 1, ('not', 'to'): 1, ('be', 'that'): 1, ('that', 'is'): 1, ('is', 'the'): 1, ('the', 'question'): 1, ('question', 'all'): 1, ('all', 'that'): 1, ('that', 'glitters'): 1, ('glitters', 'is'): 1, ('is', 'not'): 1, ('not', 'gold'): 1, ('gold', 'the'): 1, ('the', 'only'): 1, ('only', 'thing'): 1, ('thing', 'we'): 1, ('we', 'have'): 1, ('have', 'to'): 1, ('to', 'fear'): 1, ('fear', 'is'): 1, ('is', 'fear'): 1, ('fear', 'itself'): 1

Identified Misspellings: ['quik', 'jumos', 'ovr', 'dg', 'jurney', 'thosand', 'wth', 'singel']

Candidates for 'quik': 'quick'

Replacing 'quik' with 'quick'

Candidates for 'jumos': 'jumps'

Replacing 'jumos' with 'jumps'

Candidates for 'ovr': 'over', 'or'

Replacing 'ovr' with 'over'

Candidates for 'dg': 'dog'

Replacing 'dg' with 'dog'

Candidates for 'jurney': 'journey'

Replacing 'jurney' with 'journey'

Candidates for 'thosand': 'thousand'

Replacing 'thosand' with 'thousand'

Candidates for 'wth': 'with'

Replacing 'wth' with 'with'

Candidates for 'singel': 'single'

Replacing 'singel' with 'single'

Corrected Text: the quick brown fox jumps over the lazy dog a journey of a thousand miles begins with a single step

Sentiment Analysis

AIM

Implement a text classifier for sentiment analysis using the Naive Bayes theorem. Use Add-k smoothing to handle zero probabilities. Compare the performance of your classifier for k values 0.25, 0.75, and 1.

PROGRAM

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import accuracy_score, classification_report
from nltk.corpus import movie_reviews
import nltk

# Ensure that you have the nltk movie reviews corpus
nltk.download('movie_reviews')

# Load the IMDb dataset
documents = [(movie_reviews.raw(fileid), category)
              for category in movie_reviews.categories()
              for fileid in movie_reviews.fileids(category)]

# Create a DataFrame
df = pd.DataFrame(documents, columns=['text', 'label'])

# Map the labels to binary values: 'pos' -> 1 and 'neg' -> 0
df['label'] = df['label'].map({'pos': 1, 'neg': 0})

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df['text'], df['label'],
                                                    test_size=0.2, random_state=42)

# Vectorize the text data using TF-IDF
vectorizer = TfidfVectorizer()
X_train_vect = vectorizer.fit_transform(X_train)
X_test_vect = vectorizer.transform(X_test)

# Naive Bayes Classifier with Add-k smoothing
```

```
class NaiveBayes:
    def __init__(self, alpha=1.0):
        self.alpha = alpha # Add-k smoothing parameter
        self.class_priors = {}
        self.word_likelihoods = {}
        self.vocab_size = 0

    def fit(self, X, y):
        # Count the number of documents in each class
        n_documents = len(y)
        self.vocab_size = X.shape[1]

        # Calculate class prior probabilities
        classes = np.unique(y)
        self.class_priors = {c: np.log(np.sum(y == c) / n_documents) for c in classes}

        # Count word occurrences for each class
        word_counts = {c: np.zeros(self.vocab_size) for c in classes}
        for idx in range(len(y)):
            class_label = y.iloc[idx] # Use iloc to access the index
            word_counts[class_label] += X[idx].toarray()[0] # Access the vectorized data

        # Calculate the likelihood of each word given the class
        # with Add-k smoothing
        for c in classes:
            total_count = np.sum(word_counts[c]) + self.alpha * self.vocab_size #
            # Total count with
            # smoothing
            self.word_likelihoods[c] = (word_counts[c] + self.alpha)
            /
            total_count

    def predict(self, X):
        log_probabilities = []
        for c in self.class_priors:
            # Calculate log-probability for each class
            log_prob = self.class_priors[c] + X.dot(np.log(self.word_likelihoods[c]))
            log_probabilities.append(log_prob)

        # Choose the class with the highest probability
        return np.argmax(log_probabilities, axis=0)
```

```
def evaluate_model(k_value):  
    # Create and train the model  
    model = NaiveBayes(alpha=k_value)  
    model.fit(X_train_vect, y_train)  
  
    # Make predictions  
    predictions = model.predict(X_test_vect)  
  
    # Calculate accuracy and report  
    accuracy = accuracy_score(y_test, predictions)  
    report = classification_report(y_test, predictions)  
    return accuracy, report  
  
# Evaluate for different k values  
k_values = [0.25, 0.75, 1.0]  
results = {}  
  
for k in k_values:  
    accuracy, report = evaluate_model(k)  
    results[k] = (accuracy, report)  
  
# Display the results  
for k, (accuracy, report) in results.items():  
    print(f"Results for k={k}:")  
    print(f"Accuracy: {accuracy:.2f}")  
    print(f"Classification Report:\n{report}")
```

SAMPLE INPUT-OUTPUT

```
Results for k=0.25:
Accuracy: 0.80
Classification Report:
      precision    recall  f1-score   support

     0       0.77      0.86      0.81       199
     1       0.84      0.75      0.79       201

   accuracy          0.80       400
  macro avg       0.81      0.80      0.80       400
weighted avg       0.81      0.80      0.80       400

Results for k=0.75:
Accuracy: 0.81
Classification Report:
      precision    recall  f1-score   support

     0       0.76      0.89      0.82       199
     1       0.87      0.73      0.79       201

   accuracy          0.81       400
  macro avg       0.82      0.81      0.81       400
weighted avg       0.82      0.81      0.81       400

Results for k=1.0:
Accuracy: 0.81
Classification Report:
      precision    recall  f1-score   support

     0       0.76      0.90      0.82       199
     1       0.88      0.72      0.79       201

   accuracy          0.81       400
  macro avg       0.82      0.81      0.81       400
weighted avg       0.82      0.81      0.81       400
```

POS Tagging

AIM

Implement the Viterbi algorithm to find the most probable POS tag sequence for a given sentence, using the given probabilities:

a_{ij}	STOP	NN	VB	JJ	RB
START	0	0.5	0.25	0.25	0
NN	0.25	0.25	0.5	0	0
VB	0.25	0.25	0	0.25	0.25
JJ	0	0.75	0	0.25	0
RB	0.5	0.25	0	0.25	0

b_{ik}	time	flies	fast
NN	0.1	0.01	0.01
VB	0.01	0.1	0.01
JJ	0	0	0.1
RB	0	0	0.1

PROGRAM

```
import numpy as np

# Transition probabilities (a_ij)
transition_probs = {
    'START': {'NN': 0.5, 'VB': 0.25, 'JJ': 0.25, 'RB': 0, 'STOP': 0},
    'NN': {'NN': 0.25, 'VB': 0.5, 'JJ': 0, 'RB': 0.25, 'STOP': 0},
    'VB': {'NN': 0.25, 'VB': 0, 'JJ': 0.25, 'RB': 0.25, 'STOP': 0.25},
    'JJ': {'NN': 0.75, 'VB': 0, 'JJ': 0.25, 'RB': 0, 'STOP': 0},
    'RB': {'NN': 0.25, 'VB': 0, 'JJ': 0.25, 'RB': 0, 'STOP': 0.5},
}

# Emission probabilities (b_ik)
emission_probs = {
    'NN': {'time': 0.1, 'flies': 0.01, 'fast': 0.01},
    'VB': {'time': 0.01, 'flies': 0.1, 'fast': 0.01},
    'JJ': {'time': 0, 'flies': 0, 'fast': 0.1},
    'RB': {'time': 0, 'flies': 0, 'fast': 0.1},
}

# List of POS tags and words
tags = ['NN', 'VB', 'JJ', 'RB']
sentence = ['time', 'flies', 'fast']

# Initialize Viterbi and backpointer tables
viterbi = [{ } for _ in range(len(sentence))]
backpointer = [{ } for _ in range(len(sentence))]
```

```
# Print initial sentence
print("Sentence:", sentence)

# Initialize base case for t = 0
print("\nStep 1: Initialization")
for tag in tags:
    viterbi[0][tag] = transition_probs['START'][tag]
    * emission_probs[tag].get(sentence[0], 0)
    backpointer[0][tag] = None
    print(f"viterbi[0][tag] = START->{tag} *
    emission[{tag}][sentence[0]] = "
          f"{transition_probs['START'][tag]} * {emission_probs[tag].get(sentence[0],
          0)} =
          {viterbi[0][tag]}")

# Recursive case for t > 0
for t in range(1, len(sentence)):
    print(f"\nStep 2.{t}: Recursion for word '
    {sentence[t]}")
    for tag in tags:
        max_prob, prev_tag = max(
            (viterbi[t - 1][prev] * transition_probs[prev]
             [tag] * emission_probs[tag].get(sentence[t], 0)
             , prev)
            for prev in tags
        )
        viterbi[t][tag] = max_prob
        backpointer[t][tag] = prev_tag
        print(f"viterbi[{t}][tag] = max(prev_tag->{tag})
        * emission[{tag}][sentence[t]] = {max_prob}, "
              f"coming from {prev_tag}")

# Termination step
print("\nStep 3: Termination")
max_final_prob, final_tag = max(
    (viterbi[len(sentence) - 1][tag] * transition_probs[tag]
     ['STOP'], tag) for tag in tags
)
print(f"Final probabilities multiplied with STOP: {[tag, viterbi[len(sentence) - 1][tag]
for tag in tags]}")
print(f"Max final probability is {max_final_prob}, with final
```

```
tag '{final_tag}')
```

```
# Traceback to find the best path
print("\nStep 4: Traceback")
best_path = []
current_tag = final_tag
for t in range(len(sentence) - 1, -1, -1):
    best_path.insert(0, current_tag)
    print(f"Backtracking at position {t}: Current tag =
    {current_tag}, Backpointer = {backpointer[t][current_tag]}")
    current_tag = backpointer[t][current_tag]

# Output the best path
print("\nMost probable POS tag sequence:", best_path)
```

SAMPLE INPUT-OUTPUT

```
Sentence: ['time', 'flies', 'fast']

Step 1: Initialization
viterbi[0][NN] = START->NN * emission[NN][time] = 0.5 * 0.1 = 0.05
viterbi[0][VB] = START->VB * emission[VB][time] = 0.25 * 0.01 = 0.0025
viterbi[0][JJ] = START->JJ * emission[JJ][time] = 0.25 * 0 = 0.0
viterbi[0][RB] = START->RB * emission[RB][time] = 0 * 0 = 0

Step 2.1: Recursion for word 'flies'
viterbi[1][NN] = max(prev_tag->NN) * emission[NN][flies] = 0.000125, coming from NN
viterbi[1][VB] = max(prev_tag->VB) * emission[VB][flies] = 0.0025000000000000005, coming from NN
viterbi[1][JJ] = max(prev_tag->JJ) * emission[JJ][flies] = 0.0, coming from VB
viterbi[1][RB] = max(prev_tag->RB) * emission[RB][flies] = 0.0, coming from VB

Step 2.2: Recursion for word 'fast'
viterbi[2][NN] = max(prev_tag->NN) * emission[NN][fast] = 6.250000000000001e-06, coming from VB
viterbi[2][VB] = max(prev_tag->VB) * emission[VB][fast] = 6.25e-07, coming from NN
viterbi[2][JJ] = max(prev_tag->JJ) * emission[JJ][fast] = 6.250000000000001e-05, coming from VB
viterbi[2][RB] = max(prev_tag->RB) * emission[RB][fast] = 6.250000000000001e-05, coming from VB

Step 3: Termination
Final probabilities multiplied with STOP: [('NN', 0.0), ('VB', 1.5625e-07), ('JJ', 0.0), ('RB', 3.125000000000001e-05)]
Max final probability is 3.125000000000001e-05, with final tag 'RB'

Step 4: Traceback
Backtracking at position 2: Current tag = RB, Backpointer = VB
Backtracking at position 1: Current tag = VB, Backpointer = NN
Backtracking at position 0: Current tag = NN, Backpointer = None

Most probable POS tag sequence: ['NN', 'VB', 'RB']
```


Bigram Probability

AIM

Write a Python code to calculate bigrams from a given corpus and calculate the probability of any given sentence

PROGRAM

```
from collections import defaultdict, Counter
import math

# Step 1: Define a sample corpus
corpus = [
    "Natural language processing is a fascinating field.",
    "It involves the interaction between computers and human language.",
    "One important aspect is understanding context.",
    "Another key element is generating coherent responses.",
    "Machine learning plays a crucial role in NLP.",
    "Deep learning models like transformers are widely used.",
    "Tokenization is an essential preprocessing step.",
    "Named entity recognition is another core task.",
    "Sentiment analysis helps understand emotions in text.",
    "Language modeling is fundamental to many NLP applications."
]

# Preprocessing: Tokenize sentences and add START and STOP tokens
def preprocess_corpus(corpus):
    tokenized_corpus = []
    for sentence in corpus:
        words = sentence.lower().replace('.', ' ').split()
        tokenized_corpus.append(["<START>"] + words + ["<STOP>"])
    return tokenized_corpus

tokenized_corpus = preprocess_corpus(corpus)

# Step 2: Create a bigram model
def build_bigram_model(tokenized_corpus):
    bigram_counts = defaultdict(Counter)
    unigram_counts = Counter()

    for sentence in tokenized_corpus:
```

```
    for i in range(len(sentence) - 1):
        bigram_counts[sentence[i]][sentence[i + 1]] += 1
        unigram_counts[sentence[i]] += 1
    unigram_counts[sentence[-1]] += 1 # For the last token (STOP)

    bigram_probs = defaultdict(dict)
    for word1 in bigram_counts:
        for word2 in bigram_counts[word1]:
            bigram_probs[word1][word2] = bigram_counts[word1][word2]
            / unigram_counts[word1]

    return bigram_probs

bigram_model = build_bigram_model(tokenized_corpus)

# Step 3: Calculate the probability of a given sentence
def calculate_sentence_probability(sentence, bigram_model):
    words = ["<START>"] + sentence.lower().replace('.', ' ').split() + ["<STOP>"]
    probability = 0 # Use log probabilities to prevent underflow

    for i in range(len(words) - 1):
        word1, word2 = words[i], words[i + 1]
        if word2 in bigram_model[word1]:
            probability += math.log(bigram_model[word1][word2])
        else:
            probability += math.log(1e-10) # Smoothing for unseen bigrams

    return math.exp(probability) # Convert back from log-probabilities

# Example sentence and its probability
test_sentence = "Natural language processing is fascinating."
sentence_probability = calculate_sentence_probability(test_sentence, bigram_model)

# Output results
print("Bigram Model (Partial):")
for word, transitions in list(bigram_model.items())[:5]:
    print(f"{word}: {transitions}")

print("\nTest Sentence:", test_sentence)
print("Probability of the sentence:", sentence_probability)
```

SAMPLE INPUT-OUTPUT

Bigram Model (Partial):

START: 'natural': 0.1, 'it': 0.1, 'one': 0.1, 'another': 0.1, 'machine': 0.1, 'deep': 0.1,
'tokenization': 0.1, 'named': 0.1, 'sentiment': 0.1, 'language': 0.1 natural: 'language': 1.0
language: 'processing': 0.3333333333333333, 'STOP': 0.3333333333333333, 'modeling':
0.3333333333333333
processing: 'is': 1.0
is: 'a': 0.16666666666666666, 'understanding': 0.16666666666666666, 'generating':
0.16666666666666666, 'an': 0.16666666666666666, 'another': 0.16666666666666666,
'fundamental': 0.16666666666666666

Test Sentence: Natural language processing is fascinating. Probability of the sentence:
3.33333333333333347e-22

TF-IDF Matrix

AIM

Write a program to compute the TF-IDF matrix given a set of training documents. Also, calculate the cosine similarity between any two given documents or two given words.

PROGRAM

```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Step 1: Define a corpus of training documents
documents = [
    "Natural language processing enables computers to understand and generate human language.",
    "Deep learning models like transformers have revolutionized natural language processing.",
    "Tokenization and preprocessing are critical steps in NLP pipelines.",
    "Named entity recognition identifies entities such as names, dates, and locations in text.",
    "Sentiment analysis detects emotions in customer reviews or social media data.",
    "Language modeling predicts the next word in a sequence, a fundamental task in NLP.",
    "Vector embeddings represent words and documents as dense numerical arrays.",
    "NLP applications range from chatbots to machine translation and summarization.",
    "Context is essential for understanding ambiguous language and idiomatic expressions.",
    "Grammar correction and text generation are other key areas of natural language processing."
]

# Step 2: Compute the TF-IDF matrix
def compute_tfidf_matrix(documents):
    vectorizer = TfidfVectorizer()
    tfidf_matrix = vectorizer.fit_transform(documents)
    feature_names = vectorizer.get_feature_names_out()
    return tfidf_matrix, feature_names
```

```
tfidf_matrix, feature_names = compute_tfidf_matrix(documents)

# Step 3: Cosine similarity between two documents
def calculate_document_similarity(doc1_index, doc2_index, tfidf_matrix):
    vector1 = tfidf_matrix[doc1_index]
    vector2 = tfidf_matrix[doc2_index]
    similarity = cosine_similarity(vector1, vector2)[0][0]
    return similarity

# Step 4: Cosine similarity between two words
def calculate_word_similarity(word1, word2, tfidf_matrix, feature_names):
    try:
        index1 = feature_names.tolist().index(word1)
        index2 = feature_names.tolist().index(word2)
        word_vector1 = tfidf_matrix[:, index1].toarray().flatten()
        word_vector2 = tfidf_matrix[:, index2].toarray().flatten()
        similarity = cosine_similarity([word_vector1], [word_vector2])[0][0]
        return similarity
    except ValueError:
        return "One or both words not found in the vocabulary."

# Example Usage
# Cosine similarity between two documents
doc1_index = 0 # First document
doc2_index = 1 # Second document
similarity_docs = calculate_document_similarity(doc1_index, doc2_index, tfidf_matrix)

# Cosine similarity between two words
word1 = "language"
word2 = "processing"
similarity_words = calculate_word_similarity(word1, word2, tfidf_matrix, feature_names)

# Output results
print("TF-IDF Matrix (Partial):")
print(tfidf_matrix.toarray()[:5, :5]) # Show partial matrix for brevity
print(f"\nFeature Names (Partial): {feature_names[:5]}")
print(f"\nCosine Similarity between documents {doc1_index} and {doc2_index}: {similarity_docs}")
print(f"Cosine Similarity between words '{word1}' and '{word2}': {similarity_words}")
```

SAMPLE INPUT-OUTPUT

```
PPMI Matrix (Partial):  
[[0. 0. 0. 0. 0.]  
 [0. 0. 0. 0. 0.]  
 [0. 0. 0. 0. 0.]  
 [0. 0. 0. 0. 0.]  
 [0. 0. 0. 0. 0.]  
 [0. 0. 0. 0. 0.]]  
  
Cosine Similarity between words 'language' and 'processing': 0.2980892233002522  
Cosine Similarity between documents:  
'Natural language processing enables computers to understand and generate human language.'  
and  
'Deep learning models like transformers have revolutionized natural language processing.': 0.3750746564962848
```

PPMI Matrix

AIM

Write a program to compute the PPMI matrix given a set of training documents. Also, calculate the cosine similarity between any two given documents or two given words.

PROGRAM

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from collections import Counter, defaultdict

# Step 1: Define a corpus of long documents
documents = [
    "Natural language processing enables computers to understand\nand generate human language.",
    "Deep learning models like transformers have revolutionized natural\nlanguage processing.",
    "Tokenization and preprocessing are critical steps\nin NLP pipelines.",
    "Named entity recognition identifies entities such\nas names, dates, and locations in text.",
    "Sentiment analysis detects emotions in customer reviews\nor social media data.",
    "Language modeling predicts the next word in a sequence,\na fundamental task in NLP.",
    "Vector embeddings represent words and documents as dense numerical arrays.",
    "NLP applications range from chatbots to machine translation\nand summarization.",
    "Context is essential for understanding ambiguous language\nand idiomatic expressions.",
    "Grammar correction and text generation are other key\nareas of natural language processing."
]

# Preprocessing: Tokenize the corpus and count word\nco-occurrences
def tokenize_and_count(documents):
    vocab = set()
    word_counts = []
    for doc in documents:
```

```
        tokens = doc.lower().replace('.', '').split()
        vocab.update(tokens)
        word_counts.append(tokens)
    return list(vocab), word_counts

vocab, word_counts = tokenize_and_count(documents)
vocab_size = len(vocab)

# Create word-to-index mapping
word_to_index = {word: i for i, word in enumerate(vocab)}

# Step 2: Compute co-occurrence matrix
def build_cooccurrence_matrix(word_counts, vocab_size
, window_size=2):
    cooccurrence_matrix = np.zeros((vocab_size, vocab_size))
    for tokens in word_counts:
        for i, word in enumerate(tokens):
            word_index = word_to_index[word]
            for j in range(max(i - window_size, 0), min(i + window_size + 1, len(tokens))):
                if i != j:
                    context_word_index = word_to_index
                        [tokens[j]]
                    cooccurrence_matrix[word_index][context_word_index] += 1
    return cooccurrence_matrix

cooccurrence_matrix = build_cooccurrence_matrix
(word_counts, vocab_size)

# Step 3: Compute PPMI matrix
def compute_ppmi_matrix(cooccurrence_matrix):
    total_count = np.sum(cooccurrence_matrix)
    word_totals = np.sum(cooccurrence_matrix, axis=1)
    ppmi_matrix = np.zeros_like(cooccurrence_matrix)
    for i in range(cooccurrence_matrix.shape[0]):
        for j in range(cooccurrence_matrix.shape[1]):
            if cooccurrence_matrix[i][j] > 0:
                p_ij = cooccurrence_matrix[i][j] /
                    total_count
                p_i = word_totals[i] / total_count
                p_j = word_totals[j] / total_count
                ppmi = max(0, np.log2(p_ij / (p_i * p_j)))
```



```
        ppmi_matrix[i][j] = ppmi
    return ppmi_matrix

ppmi_matrix = compute_ppmi_matrix(cooccurrence_matrix)

# Step 4: Compute cosine similarity
def calculate_cosine_similarity(ppmi_matrix, word1, word2):
    index1, index2 = word_to_index[word1], word_to_index[word2]
    vector1, vector2 = ppmi_matrix[index1], ppmi_matrix[index2]
    return cosine_similarity([vector1], [vector2])[0][0]

# Step 5: Cosine similarity between documents
def document_vector(document, ppmi_matrix, word_to_index):
    vector = np.zeros(ppmi_matrix.shape[0])
    tokens = document.lower().replace('.', ' ').split()
    for token in tokens:
        if token in word_to_index:
            vector += ppmi_matrix[word_to_index[token]]
    return vector

def calculate_document_similarity(doc1, doc2, ppmi_matrix, word_to_index):
    vec1 = document_vector(doc1, ppmi_matrix, word_to_index)
    vec2 = document_vector(doc2, ppmi_matrix, word_to_index)
    return cosine_similarity([vec1], [vec2])[0][0]

# Example Usage
word1 = "language"
word2 = "processing"
similarity_words = calculate_cosine_similarity(ppmi_matrix, word1, word2)

doc1 = documents[0]
doc2 = documents[1]
similarity_docs = calculate_document_similarity(doc1, doc2, ppmi_matrix, word_to_index)

# Output
print("PPMI Matrix (Partial):")
print(ppmi_matrix[:5, :5]) # Display part of the matrix for brevity
print(f"\nCosine Similarity between words '{word1}' and '{word2}': {similarity_words}")
print(f"Cosine Similarity between documents:\n'{doc1}'\nand\n'{doc2}': {similarity_docs}")
```

SAMPLE INPUT-OUTPUT

```
TF-IDF Matrix (Partial):  
[[0.      0.      0.16742341 0.      0.      ]  
 [0.      0.      0.      0.      0.      ]  
 [0.      0.      0.18489537 0.      0.32244345]  
 [0.      0.      0.14617824 0.      0.      ]  
 [0.      0.30953339 0.      0.      0.      ]]  
  
Feature Names (Partial): ['ambiguous' 'analysis' 'and' 'applications' 'are']  
  
Cosine Similarity between documents 0 and 1: 0.21392570912457406  
Cosine Similarity between words 'language' and 'processing': 0.8252940664534878
```

Naive Bayes Classifier

AIM

Implement a Naive Bayes classifier with add-1 smoothing using a given test data and disambiguate any word in a given test sentence. Use Bag-of-words as the feature. You may define your vocabulary. Sample Input :

No.	Sentence	Sense
1	I love fish. The smoked <u>bass</u> fish was delicious.	fish
2	The <u>bass</u> fish swam along the line.	fish
3	He hauled in a big catch of smoked <u>bass</u> fish.	fish
4	The <u>bass</u> guitar player played a smooth jazz line.	guitar

Test Sentence: He loves jazz. The bass_line provided the foundation for the guitar solo in the jazz piece

PROGRAM

```
import numpy as np
from collections import defaultdict

# Training data and labels
train_data = [
    ("I love fish. The smoked bass fish was delicious.", "fish"),
    ("The bass fish swam along the line.", "fish"),
    ("He hauled in a big catch of smoked bass fish.", "fish"),
    ("The bass guitar player played a smooth jazz line.", "guitar")
]

# Vocabulary and preprocessing
vocabulary = set()
word_counts = {'fish': defaultdict(int), 'guitar': defaultdict(int)}
class_counts = {'fish': 0, 'guitar': 0}
total_words = {'fish': 0, 'guitar': 0}

# Preprocess the training data to build vocabulary and counts
for sentence, label in train_data:
    words = sentence.lower().split()
    class_counts[label] += 1
    total_words[label] += len(words)
```

```
    for word in words:
        word_counts[label][word] += 1
        vocabulary.add(word)

# Add-1 smoothing
V = len(vocabulary) # Vocabulary size
total_sentences = len(train_data) # Total number of sentences

# Calculate priors and conditional probabilities with add-1 smoothing
def calculate_probabilities(class_counts, word_counts, total_words, V):
    prior_probs = {label: class_counts[label] / total_sentences for label in class_counts}
    cond_probs = {label: {} for label in class_counts}

    for label in class_counts:
        total_word_count_in_class = total_words[label]

        for word in vocabulary:
            cond_probs[label][word] = (word_counts[label].get(word, 0) + 1) / (total_word_count_in_class + V)

    return prior_probs, cond_probs

prior_probs, cond_probs = calculate_probabilities(class_counts, word_counts, total_words, V)

# Test sentence and test word
test_sentence = "He loves jazz. The bass line provided the foundation for the guitar solo"
test_word = "bass"

# Process test sentence
test_words = test_sentence.lower().split()

# Calculate posterior probabilities for each class (fish, guitar)
def classify(test_words, prior_probs, cond_probs, V):
    post_probs = {}

    for label in prior_probs:
        prob = np.log(prior_probs[label])

        for word in test_words:
            prob += np.log(cond_probs[label].get(word, 1 / (V + total_words[label])))

        post_probs[label] = prob
```

```
# Return the class with the highest posterior probability
return max(post_probs, key=post_probs.get)

predicted_sense = classify(test_words, prior_probs, cond_probs, V)

print(f"Predicted sense for the word '{test_word}' in the test sentence: {predicted_sense}")
```

SAMPLE INPUT-OUTPUT

```
Predicted sense for the word 'bass' in the test sentence: guitar
```