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

OMAL S
(80521015)

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

DECEMBER 2024

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* ***OMAL S (80521015)****, 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 seventh semester laboratory prescribed for the Master of Science (Five Year Integrated) in Computer Science degree.*

Faculty Member in-charge

Dr. Jeena Kleenankandy
Assistant Professor
Department of Computer Science
CUSAT

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

Table of Contents

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):
    # Regular expression
    pattern = re.compile(r"""
        (?:[A-Za-z]\.){2,}              #  U.S.A., e.g.
        | \w+(?:-\w+)+                  #  ice-cream
        | \b(?:[A-Za-z]+)(?:n’t|’re|’s|’d|’ll|’ve|’m)\b  # isn’t, I’m
        | \b\w+\b                       # regular words (single words)
        | [.,!?;:"\"’‘’()\[\]{}<>]      # punctuations and special symbols
    """, re.VERBOSE)

    # Extract tokens based on the pattern
    tokens = pattern.findall(text)

    # isn’t → is, n’t
    final_tokens = []
    for token in tokens:

        contraction_split = re.split(r"(n’t|’re|’s|’d|’ll|’ve|’m)", token)
        final_tokens.extend([t for t in contraction_split if t])

    return final_tokens

# Example usage
text = "The U.S.A. isn’t perfect, but ice-cream is delicious!
We’ve been there for a week."
tokens = tokenize(text)
print(tokens)
```

SAMPLE INPUT-OUTPUT

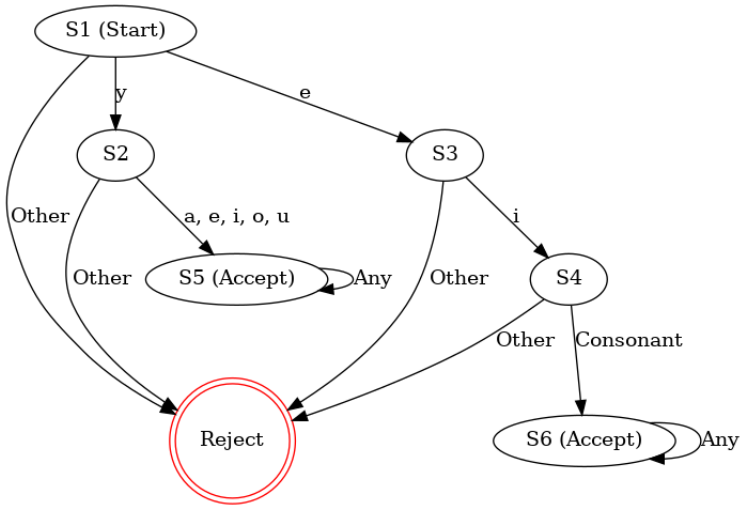
```
['The', 'U.S.A.', 'is', "n't", 'perfect', ',', 'but',
 'ice-cream', 'is', 'delicious', '!', 'We', "'ve",
'been', 'there', 'for', 'a', 'week', '.']
```

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):
    if len(word) < 2 or word[-1] != 's':
        return False

    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
        elif state == 'S6':
            continue

    return True

test_words = ["boys", "toys", "ponies", "skies", "puppies", "boies", "toies",
          "ponys", "girl"]
results = {word: is_plural_noun_accepted_fsa(word) for word in test_words}

print(results )


#using Transition Table
# Transition table
transition_table = {
    'S1': {'y': 'S2', 'e': 'S3'},
    'S2': {vowel: 'S5' for vowel in 'aeiou'},
    'S3': {'i': 'S4'},
    'S4': {consonant: 'S6' for consonant in 'bcdfghjklmnpqrtvwxyz'},
    'S5': {char: 'S5' for char in 'abcdefghijklmnopqrstuvwxyz'},
    'S6': {char: 'S6' for char in 'abcdefghijklmnopqrstuvwxyz'}
}


def is_plural_noun_accepted_fsa(word):
    if len(word) < 2 or word[-1] != 's':
        return False
```

```
    word = word[::-1]
    state = 'S1'

    for char in word[1:]:
        transitions = transition_table.get(state)

        if char not in transitions:
            return False

        state = transitions[char]

    return state in ['S5', 'S6']

test_words = ["boys", "toys", "ponies", "skies", "puppies", "boies",
    "toies", "ponys", "girl"]
results = {word: is_plural_noun_accepted_fsa(word) for word in test_words}

print(results)
```

SAMPLE INPUT-OUTPUT

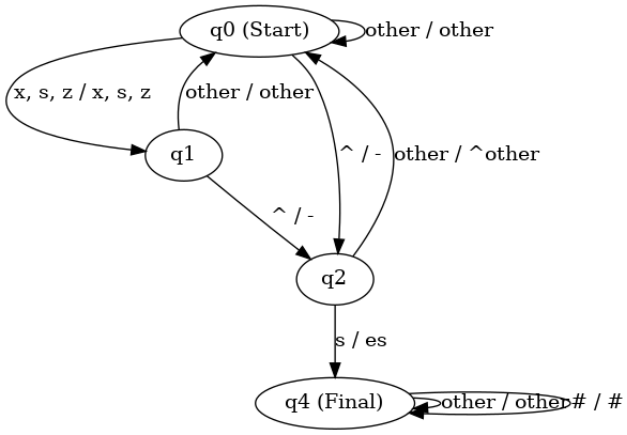
```
    {'boys': True,
      'toys': True,
      'ponies': True,
      'skies': True,
      'puppies': True,
      'boies': False,
      'toies': False,
      'ponys': False}
```

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.

PROGRAM




```
class PluralFST:
    def __init__(self):
        self.state = "q0"
        self.output = ""

    def reset(self):
        """Resets the FST to the initial state."""
        self.state = "q0"
        self.output = ""

    def process(self, input_word):
        """Processes an input lexical word to generate its plural form."""
        self.reset()
        i = 0

        while i < len(input_word):
            char = input_word[i]

            if self.state == "q0":
                if char in {'x', 's', 'z'}:
                    self.output += char
                    self.state = "q1"
```

```
                elif char == "^":
                    self.state = "q2"
                else:
                    self.output += char

            elif self.state == "q1":
                if char == "^":
                    self.state = "q2"
                else:
                    self.output += char
                    self.state = "q0"

            elif self.state == "q2":
                if char == "s":
                    if self.output[-1] in {'x', 's', 'z'}:
                        self.output += "e"
                    self.output += "s"
                    self.state = "q4"
                else:
                    self.output += "^" + char
                    self.state = "q0"

            elif self.state == "q4":
                if char == "#":
                    self.output += "#"
                else:
                    self.output += char

            i += 1

        return self.output


# Test the FST with provided examples
def test_fst():
    fst = PluralFST()

    # Test cases
    test_words = ["fox^s#", "boy^s#", "bus^s#", "buzz^s#", "class^s#",
         "cat^s#", "batch^s#"]
```

```
    for word in test_words:
        result = fst.process(word)
        result = result.replace('#', '')
        print(f"Input: {word} -> Output: {result}")

# Run the tests
test_fst()
```

SAMPLE INPUT-OUTPUT

```
Input: fox^s# -> Output: foxes
Input: boy^s# -> Output: boys
Input: bus^s# -> Output: buses
Input: buzz^s# -> Output: buzzes
Input: class^s# -> Output: classes
Input: cat^s# -> Output: cats
Input: batch^s# -> Output: batchs
```

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 minimum_edit_distance(source, target):
    rows, cols = len(source) + 1, len(target) + 1
    distance = [[0 for _ in range(cols)] for _ in range(rows)]

    for i in range(1, rows):
        distance[i][0] = i
    for j in range(1, cols):
        distance[0][j] = j

    for i in range(1, rows):
        for j in range(1, cols):
            if source[i - 1] == target[j - 1]:
                cost = 0
            else:
                cost = 2

            distance[i][j] = min(
                distance[i - 1][j] + 1,       # Deletion
                distance[i][j - 1] + 1,       # Insertion
                distance[i - 1][j - 1] + cost # Substitution
            )

    operations = []
    i, j = len(source), len(target)

    while i > 0 and j > 0:
        if source[i - 1] == target[j - 1]:
            i -= 1
            j -= 1
        else:
            current = distance[i][j]
            if current == distance[i - 1][j - 1] + 2:  # Substitution
                operations.append(f"Substitute '{source[i - 1]}' with
```

```
                    '{target[j - 1]}' (Cost: 2)")
                i -= 1
                j -= 1
            elif current == distance[i][j - 1] + 1:  # Insertion
                operations.append(f"Insert '{target[j - 1]}' (Cost: 1)")
                j -= 1
            else:  # Deletion
                operations.append(f"Delete '{source[i - 1]}' (Cost: 1)")
                i -= 1

    while i > 0:
        operations.append(f"Delete '{source[i - 1]}' (Cost: 1)")
        i -= 1
    while j > 0:
        operations.append(f"Insert '{target[j - 1]}' (Cost: 1)")
        j -= 1

    total_cost = distance[-1][-1]
    return total_cost, operations[::-1]  # Reverse the operations list


source_str = "intention"
target_str = "execution"

edit_distance, edit_operations = minimum_edit_distance(source_str, target_str)

print(f"Minimum Edit Distance: {edit_distance}")
print("Edit Operations:")
for operation in edit_operations:
    print(operation)
```

SAMPLE INPUT-OUTPUT

```
Minimum Edit Distance: 8
Edit Operations:
Delete 'i' (Cost: 1)
Substitute 'n' with 'e' (Cost: 2)
Substitute 't' with 'x' (Cost: 2)
Insert 'c' (Cost: 1)
Substitute 'n' with 'u' (Cost: 2)
```

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 Counter, defaultdict

# Step 1: Tokenize the Corpus and Create a Vocabulary
def tokenize_corpus(corpus):
    words = re.findall(r'\b[a-z]+\b', corpus.lower())
    return words

# Step 2: Create a Bigram Frequency Table
def create_bigram_freq_table(tokens):
    bigram_freq = defaultdict(int)
    for i in range(len(tokens) - 1):
        bigram = (tokens[i], tokens[i + 1])
        bigram_freq[bigram] += 1
    return bigram_freq

# Step 3: Identify Non-Word Spelling Errors
def detect_non_word_errors(words, vocabulary):
    return [word for word in words if word not in vocabulary]

# Step 4: Generate Candidates List Using 1 Edit Distance
def generate_candidates(word, vocabulary):
    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]

    candidates = set(deletes + transposes + replaces + inserts)
    return [candidate for candidate in candidates if candidate in vocabulary]

# Step 5: Suggest the Best Candidate Word Using Bigram Probabilities
def bigram_probability(prev_word, current_word, bigram_freq_table,
unigram_freq, vocab_size):
    bigram = (prev_word, current_word)
    bigram_count = bigram_freq_table.get(bigram, 0)
    prev_word_count = unigram_freq.get(prev_word, 0)
    return (bigram_count + 1) / (prev_word_count + vocab_size)

def suggest_best_candidate(input_words, misspelled_words,
bigram_freq_table, vocabulary):
    unigram_freq = Counter(vocabulary)
    vocab_size = len(vocabulary)

    corrected_words = input_words[:]
    for i, word in enumerate(input_words):
        if word in misspelled_words:
            prev_word = input_words[i - 1] if i > 0 else '<s>'
            candidates = generate_candidates(word, vocabulary)

            if not candidates:
                continue

            best_candidate = max(
                candidates,
                key=lambda candidate: bigram_probability(prev_word, candidate,
                bigram_freq_table, unigram_freq, vocab_size)
            )
            corrected_words[i] = best_candidate

    return corrected_words

def statistical_spell_checker(corpus, input_sentence):
```

```
    tokens = tokenize_corpus(corpus)
    vocabulary = set(tokens)

    bigram_freq_table = create_bigram_freq_table(tokens)

    input_words = tokenize_corpus(input_sentence)
    misspelled_words = detect_non_word_errors(input_words, vocabulary)
    print(f"Misspelled words: {misspelled_words}")

    corrected_sentence = suggest_best_candidate(input_words, misspelled_words,
    bigram_freq_table, vocabulary)

    return " ".join(corrected_sentence)

corpus = """
It is a spell checker that uses a statistical language model.
The language model is based on bigrams.
Bigrams are pairs of consecutive words in a sentence.
A bigram model considers the probability of each word given the previous word.
This is a simple example corpus with enough text to cover basic English words
and some example sentences to provide a context for bigrams and vocabulary.
Ideally, you would use a larger English corpus here."""

input_sentence = "It is a speel cheker that use a statistical langauge model."
corrected_sentence = statistical_spell_checker(corpus, input_sentence)

print("Corrected Sentence:", corrected_sentence)

input_sentence1 = "This is an exmple text with sme errors."
corrected_text = statistical_spell_checker(corpus, input_sentence1)

print("Corrected Text:", corrected_text)
```

SAMPLE INPUT-OUTPUT

```
Misspelled words: ['speel', 'cheker', 'langauge']
Corrected Sentence: it is a spell checker that use a statistical language model
Misspelled words: ['an', 'exmple', 'sme', 'errors']
Corrected Text: this is a example text with some errors
```

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 numpy as np
import pandas as pd
from collections import defaultdict
from nltk.corpus import movie_reviews
import nltk
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# Download nltk data
nltk.download('movie_reviews')
nltk.download('punkt')

documents = [(list(movie_reviews.words(fileid)), category)
             for category in movie_reviews.categories()
             for fileid in movie_reviews.fileids(category)]

np.random.shuffle(documents)
train_data, test_data = train_test_split(documents, test_size=0.2, random_state=42)

def build_vocabulary(data):
    vocab = defaultdict(lambda: 0)
    for words, _ in data:
        for word in words:
            vocab[word] += 1
    return vocab

# Build vocabulary from training data
vocabulary = build_vocabulary(train_data)

# Count words by class
def count_words_by_class(data, vocabulary):
    word_counts = { "pos": defaultdict(lambda: 0), "neg": defaultdict(lambda: 0) }
```

```
    class_counts = { "pos": 0, "neg": 0 }
    for words, label in data:
        class_counts[label] += len(words)
        for word in words:
            word_counts[label][word] += 1
    return word_counts, class_counts

word_counts, class_counts = count_words_by_class(train_data, vocabulary)

# Calculate priors
total_docs = len(train_data)
prior_pos = sum(1 for _, label in train_data if label == 'pos') / total_docs
prior_neg = sum(1 for _, label in train_data if label == 'neg') / total_docs

# Helper function: Naive Bayes classifier with Add-k smoothing
def classify(text, k, vocabulary, word_counts, class_counts):
    words = nltk.word_tokenize(text)
    vocab_size = len(vocabulary)
    log_prob_pos = np.log(prior_pos)
    log_prob_neg = np.log(prior_neg)

    for word in words:
        # P(word|positive)
        word_prob_pos = (word_counts["pos"][word] + k) / (class_counts["pos"] +
             k * vocab_size)
        log_prob_pos += np.log(word_prob_pos)

        # P(word|negative)
        word_prob_neg = (word_counts["neg"][word] + k) / (class_counts["neg"] +
             k * vocab_size)
        log_prob_neg += np.log(word_prob_neg)

    return "pos" if log_prob_pos > log_prob_neg else "neg"

# Test classifier on test data with different k values
def test_classifier(k):
    predictions = []
    true_labels = []
    for words, label in test_data:
        text = " ".join(words)
        prediction = classify(text, k, vocabulary, word_counts, class_counts)
```

```
        predictions.append(prediction)
        true_labels.append(label)

    accuracy = accuracy_score(true_labels, predictions)
    report = classification_report(true_labels, predictions, output_dict=True)
    return accuracy, report

# Test with different k values
results = {}
for k in [0.25, 0.75, 1]:
    accuracy, report = test_classifier(k)
    results[k] = {
        "accuracy": accuracy,
        "precision": report['pos']['precision'],
        "recall": report['pos']['recall'],
        "f1-score": report['pos']['f1-score']
    }

results_df = pd.DataFrame(results).transpose()
results_df
```

SAMPLE INPUT-OUTPUT

	accuracy	precision	recall	f1-score
0.25	0.8025	0.785714	0.829146	0.806846
0.75	0.8075	0.801980	0.814070	0.807980
1.00	0.8125	0.810000	0.814070	0.812030

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	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

# Define the transition probabilities (a_ij) for POS tags
transition_probs = {
    'START': {'NN': 0.5, 'VB': 0.25, 'JJ': 0.25, 'RB': 0.0},
    'NN': {'STOP': 0.25, 'NN': 0.25, 'VB': 0.5, 'JJ': 0.0, 'RB': 0.0},
    'VB': {'STOP': 0.25, 'NN': 0.25, 'VB': 0.25, 'JJ': 0.25, 'RB': 0.25},
    'JJ': {'STOP': 0.0, 'NN': 0.75, 'VB': 0.25, 'JJ': 0.0, 'RB': 0.25},
    'RB': {'STOP': 0.5, 'NN': 0.25, 'VB': 0.0, 'JJ': 0.25, 'RB': 0.0},
}

# Define the emission probabilities (b_ik) for words given POS tags
emission_probs = {
    'NN': {'time': 0.1, 'flies': 0.01, 'fast': 0.01},
    'VB': {'time': 0.01, 'flies': 0.1, 'fast': 0.01},
    'JJ': {'time': 0.0, 'flies': 0.0, 'fast': 0.1},
    'RB': {'time': 0.0, 'flies': 0.0, 'fast': 0.1},
}

# Define the sentence to tag
sentence = ['time', 'flies', 'fast']

# Define POS tags (excluding START and STOP)
tags = ['NN', 'VB', 'JJ', 'RB']

# Initialize Viterbi table and path pointers
```

```
viterbi = np.zeros((len(tags), len(sentence)))
backpointer = np.zeros((len(tags), len(sentence)), dtype=int)

# Initialization step
for i, tag in enumerate(tags):
    viterbi[i, 0] = transition_probs['START'][tag] * emission_probs[tag].
         get(sentence[0], 0)
    backpointer[i, 0] = 0

# Recursion step
for t in range(1, len(sentence)):
    for i, tag in enumerate(tags):
        max_prob = 0
        best_prev_state = 0
        for j, prev_tag in enumerate(tags):
            prob = viterbi[j, t-1] * transition_probs[prev_tag].get(tag, 0) *
                emission_probs[tag].get(sentence[t], 0)
            if prob > max_prob:
                max_prob = prob
                best_prev_state = j
        viterbi[i, t] = max_prob
        backpointer[i, t] = best_prev_state

# Termination step
best_last_tag = 0
best_last_prob = 0
for i, tag in enumerate(tags):
    prob = viterbi[i, len(sentence) - 1] * transition_probs[tag].get('STOP', 0)
    if prob > best_last_prob:
        best_last_prob = prob
        best_last_tag = i

# Backtrace to find the best path
best_path = [best_last_tag]
for t in range(len(sentence) - 1, 0, -1):
    best_path.insert(0, backpointer[best_path[0], t])

# Map tag indices back to tag names
best_tag_sequence = [tags[i] for i in best_path]

# Output the result
```

```
print("Most probable POS tag sequence:", best_tag_sequence)
print("Probability of the best sequence:", best_last_prob)
```

SAMPLE INPUT-OUTPUT

```
Most probable POS tag sequence: ['NN', 'VB', 'RB']
Probability of the best sequence: 3.1250000000000001e-05
```

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

def preprocess_text(text):
    # Simple preprocessing: lowercase and split by whitespace
    words = text.lower().split()
    return words

def build_bigram_model(corpus):
    # Preprocess corpus
    words = preprocess_text(corpus)

    # Count bigrams and single words
    bigram_counts = Counter((words[i], words[i+1]) for i in range(len(words) - 1))
    unigram_counts = Counter(words)

    # Calculate bigram probabilities
    bigram_probs = {}
    for (w1, w2), count in bigram_counts.items():
        bigram_probs[(w1, w2)] = count / unigram_counts[w1]

    return bigram_probs, unigram_counts

def calculate_sentence_probability(sentence, bigram_probs,
      unigram_counts, smoothing=1e-6):
    # Preprocess sentence
    words = preprocess_text(sentence)

    # Calculate probability of the sentence
    sentence_prob = 1.0
    for i in range(len(words) - 1):
        w1, w2 = words[i], words[i+1]
        # Get the probability of the bigram, apply smoothing if not found
```

```
        bigram_prob = bigram_probs.get((w1, w2), smoothing /
           (unigram_counts[w1] + smoothing))
        sentence_prob *= bigram_prob

    # Convert to log probability if desired (for very low probabilities)
    log_sentence_prob = math.log(sentence_prob) if sentence_prob > 0
        else float('-inf')

    return sentence_prob, log_sentence_prob

# Example usage
corpus = "time flies like an arrow fruit flies like a banana"
sentence = "fruit flies like a banana"

# Build bigram model from corpus
bigram_probs, unigram_counts = build_bigram_model(corpus)

# Calculate probability of the sentence
sentence_prob, log_sentence_prob = calculate_sentence_probability(sentence,
     bigram_probs, unigram_counts)

print("Sentence Probability:", sentence_prob)
print("Log Sentence Probability:", log_sentence_prob)

}
```

SAMPLE INPUT-OUTPUT

```
Sentence Probability: 0.5
Log Sentence Probability: -0.6931471805599453
```

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

# Define a set of training documents
documents = [
    "time flies like an arrow",
    "fruit flies like a banana",
    "fast and furious",
    "time and tide wait for none"
]

# Step 1: Compute the TF-IDF matrix using sklearn
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(documents)
feature_names = vectorizer.get_feature_names_out()

# Display the TF-IDF matrix
print("TF-IDF Matrix:")
print(tfidf_matrix.toarray())

# Step 2: Function to calculate cosine similarity between two documents
def document_similarity(doc_index1, doc_index2):
    cosine_sim = cosine_similarity(tfidf_matrix[doc_index1],
        tfidf_matrix[doc_index2])[0][0]
    return cosine_sim

# Step 3: Function to calculate cosine similarity between two words
def word_similarity(word1, word2):
    # Check if both words are in the vocabulary
    if word1 not in feature_names or word2 not in feature_names:
        print("One or both words not in the vocabulary.")
        return None
```

```
    # Get the index of each word in the feature_names
    index1 = feature_names.tolist().index(word1)
    index2 = feature_names.tolist().index(word2)

    # Extract the word vectors (i.e., columns of the TF-IDF matrix)
    word_vector1 = tfidf_matrix[:, index1].toarray().flatten()
    word_vector2 = tfidf_matrix[:, index2].toarray().flatten()

    # Calculate cosine similarity between word vectors
    cosine_sim = cosine_similarity([word_vector1], [word_vector2])[0][0]
    return cosine_sim

# Example usage
doc_index1 = 0
doc_index2 = 1
print(f"Cosine Similarity between Document {doc_index1} and Document {doc_index2}:",
      document_similarity(doc_index1, doc_index2))

word1 = "time"
word2 = "flies"
print(f"Cosine Similarity between words '{word1}' and '{word2}':",
      word_similarity(word1, word2))
```

SAMPLE INPUT-OUTPUT

```
TF-IDF Matrix:
[[0.50867187 0.         0.50867187 0.         0.         0.40104275
  0.         0.         0.         0.40104275 0.         0.
  0.40104275 0.        ]
 [0.         0.         0.         0.55528266 0.         0.43779123
  0.         0.55528266 0.         0.43779123 0.         0.
  0.         0.        ]
 [0.         0.48693426 0.         0.         0.61761437 0.
  0.         0.         0.61761437 0.         0.         0.
  0.         0.        ]
 [0.         0.34431452 0.         0.         0.         0.
  0.43671931 0.         0.         0.         0.43671931 0.43671931
  0.34431452 0.43671931]]
Cosine Similarity between Document 0 and Document 1: 0.3511459953905124
Cosine Similarity between words 'time' and 'flies': 0.5125075431914927
```

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 collections import defaultdict
from sklearn.metrics.pairwise import cosine_similarity
def compute_ppmi_matrix(documents):
    # Build a vocabulary
    word_counts = defaultdict(int)
    co_occurrence_counts = defaultdict(int)
    total_count = 0
    for doc in documents:
        words = doc.split()
        total_count += len(words)
        for i, word in enumerate(words):
            word_counts[word] += 1
            for j in range(i + 1, len(words)):
                co_occurrence_counts[(word, words[j])] += 1
                co_occurrence_counts[(words[j], word)] += 1
    # Create a sorted list of vocabulary
    vocab = sorted(word_counts.keys())
    vocab_size = len(vocab)
    vocab_index = {word: i for i, word in enumerate(vocab)}
    # Initialize co-occurrence and PPMI matrices
    co_matrix = np.zeros((vocab_size, vocab_size))
    for (word1, word2), count in co_occurrence_counts.items():
        i, j = vocab_index[word1], vocab_index[word2]
        co_matrix[i, j] = count
    # Compute PPMI matrix
    ppmi_matrix = np.zeros_like(co_matrix)
    for i in range(vocab_size):
        for j in range(vocab_size):
            joint_prob = co_matrix[i, j] / total_count
            if joint_prob > 0:
                ppmi = max(0, np.log2(joint_prob / ((word_counts[vocab[i]] /
                total_count) * (word_counts[vocab[j]] / total_count))))
```

```
                ppmi_matrix[i, j] = ppmi
    return ppmi_matrix, vocab_index
def cosine_similarity_between_entities(matrix, index, entity1, entity2):
    vec1 = matrix[index[entity1]]
    vec2 = matrix[index[entity2]]
    return cosine_similarity([vec1], [vec2])[0][0]
# Define training documents
documents = [
    "the quick brown fox jumps over the lazy dog",
    "the quick brown fox is very quick",
    "the lazy dog sleeps all day",
]
# Compute the PPMI matrix
ppmi_matrix, vocab_index = compute_ppmi_matrix(documents)

# Example: Cosine similarity between two words
word1 = "quick"
word2 = "lazy"
similarity = cosine_similarity_between_entities
     (ppmi_matrix, vocab_index, word1, word2)
print(f"Cosine Similarity between '{word1}' and '{word2}': {similarity}")

# Example: Cosine similarity between two documents (convert documents into
PPMI-based vectors)
doc_ppmi_matrix = np.zeros((len(documents), ppmi_matrix.shape[0]))
for i, doc in enumerate(documents):
    words = doc.split()
    for word in words:
        if word in vocab_index:
            doc_ppmi_matrix[i] += ppmi_matrix[vocab_index[word]]

doc_similarity = cosine_similarity([doc_ppmi_matrix[0]],
       [doc_ppmi_matrix[1]])[0][0]
print(f"Cosine Similarity between Document 1 and Document 2: {doc_similarity}")
```

SAMPLE INPUT-OUTPUT

```
Cosine Similarity between 'quick' and 'lazy': 0.6015694425758082
Cosine Similarity between Document 1 and Document 2: 0.9335628509370792
```

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 **bass** fish was delicious.	fish
2	The **bass** fish swam along the line.	fish
3	He hauled in a big catch of smoked **bass** fish.	fish
4	The **bass** 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
- Test word: bass
- Output: guitar

PROGRAM

```
from collections import defaultdict, Counter
import numpy as np

# Training data
training_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")
]

# Preprocess training data
def tokenize(text):
    return text.lower().replace('.', '').replace(',', '').split()

# Step 1: Build vocabulary and calculate word frequencies by sense
vocabulary = set()
sense_word_counts = defaultdict(Counter)
sense_counts = Counter()
```

```
for sentence, sense in training_data:
    words = tokenize(sentence)
    vocabulary.update(words)
    sense_word_counts[sense].update(words)
    sense_counts[sense] += 1

vocab_size = len(vocabulary)

# Step 2: Calculate priors for each sense
total_senses = sum(sense_counts.values())
priors = {sense: count / total_senses for sense, count in sense_counts.items()}

# Step 3: Calculate likelihoods with add-1 smoothing
def word_likelihood(word, sense):
    word_count = sense_word_counts[sense][word]
    total_count = sum(sense_word_counts[sense].values())
    return (word_count + 1) / (total_count + vocab_size)  # Add-1 smoothing

# Step 4: Classify the test sentence
def classify(sentence, target_word):
    words = tokenize(sentence)
    max_prob = -np.inf
    best_sense = None

    for sense in sense_counts:
        # Calculate log-probability to avoid underflow
        log_prob = np.log(priors[sense])

        # Only consider the target word's probability and surrounding words
        for word in words:
            log_prob += np.log(word_likelihood(word, sense))

        # Select the sense with the maximum probability
        if log_prob > max_prob:
            max_prob = log_prob
            best_sense = sense

    return best_sense

# Test data
test_sentence = "He loves jazz. The bass line provided the foundation for the
```

```
 guitar solo in the jazz piece"
target_word = "bass"

# Classify the target word in the test sentence
predicted_sense = classify(test_sentence, target_word)
print(f"Test word: {target_word}")
print(f"Predicted sense: {predicted_sense}")
```

SAMPLE INPUT-OUTPUT

```
Test word: bass
Predicted sense: guitar
```