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

HIRA MOHAMMED K
(80521011)

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* ***HIRA MOHAMMED K (80521011)****, 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):
    pattern = r"""(?x)
                  (?:[A-Za-z]\.){2,}
                | (?:\w+(?:-\w+)+)
                | \b\w+'\w+\b
                | \b\w+\b
                | [\.,!?;:]
                | [^ \t\n]+
                """

    tokens = re.findall(pattern, text)
    processed_tokens = []
    for token in tokens:
        if re.match(r"\b\w+n't\b", token):
            processed_tokens.append(token[:-3])
            processed_tokens.append("n't")
        elif re.match(r"\b\w+'\w+\b", token):
            processed_tokens.append(token.split("'")[0])
            processed_tokens.append("'" + token.split("'")[1])
        else:
            processed_tokens.append(token)

    return processed_tokens

text = """It's a beautiful morning in October, and the air is crisp and refreshing.
U.S.A. is known for its diverse landscapes and vibrant cities. One of the country's
favorite treats is ice-cream, especially during these cool autumn months. Isn't it
amazing how something as simple as a scoop of ice-cream can bring so much joy? People
```

```
of all ages gather together to enjoy the chilly, creamy dessert. There's something
special about it, a perfect end to a lovely October day."""

tokens = tokenize(text)
print(tokens)
```

SAMPLE INPUT-OUTPUT

```
[
  'It', ''s', 'a', 'beautiful', 'morning',
  'in', 'October', ',', 'and', 'the',
  'air', 'is', 'crisp', 'and', 'refreshing',
  '.', 'U.S.A.', 'is', 'known', 'for',
  'its', 'diverse', 'landscapes', 'and', 'vibrant',
  'cities', '.', 'One', 'of', 'the',
  'country', ''s', 'favorite', 'treats', 'is',
  'ice-cream', ',', 'especially', 'during', 'these',
  'cool', 'autumn', 'months', '.', 'Is',
  'n't', 'it', 'amazing', 'how', 'something',
  'as', 'simple', 'as', 'a', 'scoop',
  'of', 'ice-cream', 'can', 'bring', 'so',
  'much', 'joy', '?', 'People', 'of',
  'all', 'ages', 'gather', 'together', 'to',
  'enjoy', 'the', 'chilly', ',', 'creamy',
  'dessert', '.', 'There', ''s', 'something',
  'special', 'about', 'it', ',', 'a',
  'perfect', 'end', 'to', 'a', 'lovely',
  'October', 'day', '.',
]
```

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).

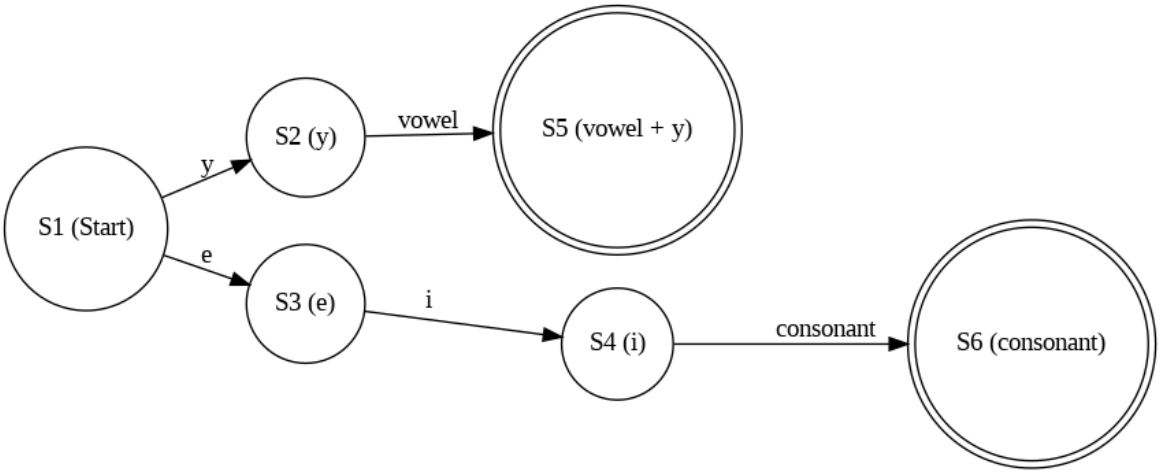
PROGRAM

```
import graphviz

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', 'carries', 'daisies']
results = {word: is_plural_noun_accepted_fsa(word) for word in test_words}

items = list(results.items())
for i in range(0, len(items), 5):
    line = ', '.join([f"'{key}': {value}" for key, value in items[i:i+5]])
    print(line)

def draw_fsa():
    dot = graphviz.Digraph(comment='DFA for Plural Noun Detection', format='png')
    dot.attr(rankdir='LR')

    dot.node('S1', 'S1 (Start)', shape='circle')
    dot.node('S2', 'S2 (y)', shape='circle')
    dot.node('S3', 'S3 (e)', shape='circle')
    dot.node('S4', 'S4 (i)', shape='circle')
    dot.node('S5', 'S5 (vowel + y)', shape='doublecircle')
    dot.node('S6', 'S6 (consonant)', shape='doublecircle')

    dot.edge('S1', 'S2', label="y")
    dot.edge('S1', 'S3', label="e")
    dot.edge('S2', 'S5', label="vowel")
    dot.edge('S3', 'S4', label="i")
    dot.edge('S4', 'S6', label="consonant")

    return dot

fsa_graph = draw_fsa()
fsa_graph.render('plural_noun_fsa_lr', format='png', cleanup=True)

from IPython.display import Image
Image('plural_noun_fsa_lr.png')
```

SAMPLE INPUT-OUTPUT

```
'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.

PROGRAM

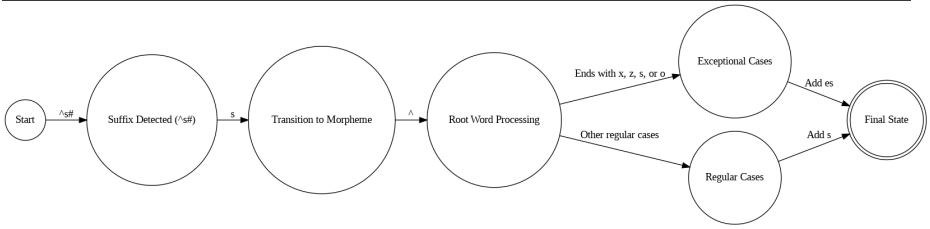
```
import graphviz

def pluralize_fst(word):
    state = 'S0'
    exceptionals = {'x', 'z', 's', 'o'}
    result = []

    for letter in word[::-1]:
        if state == 'S0':
            if letter == '#':
                state = 'S1'
            else:
                return None
        elif state == 'S1':
            if letter == 's':
                state = 'S2'
            else:
                return None
        elif state == 'S2':
            if letter == '^':
                state = 'S3'
            else:
                return None
        elif state == 'S3':
            if letter in exceptionals:
                state = 'S4'
                result.append('s')
                result.append('e')
                if letter == 'z':
                    result.append('z')
                result.append(letter)
            else:
```

```
                state = 'S5'
                result.append('s')
                result.append(letter)
        elif state == 'S4':
            result.append(letter)
        elif state == 'S5':
            result.append(letter)

    return ''.join(result[::-1])

test_words = [
    "fox^s#",
    "boy^s#",
    "bus^s#",
    "quiz^s#",
    "dog^s#",
    "hero^s#",
    "glass^s#",
    "tomato^s#",
]

results = {word: pluralize_fst(word) for word in test_words}

for word, plural in results.items():
    print(f"{word}  :  {plural}")

def draw_pluralization_fst():
    dot = graphviz.Digraph(comment='FST for Pluralization', format='png')
    dot.attr(rankdir='LR')

    dot.node('S0', 'Start', shape='circle')
    dot.node('S1', 'Suffix Detected (^s#)', shape='circle')
    dot.node('S2', 'Transition to Morpheme', shape='circle')
    dot.node('S3', 'Root Word Processing', shape='circle')
    dot.node('S4', 'Exceptional Cases', shape='circle')
    dot.node('S5', 'Regular Cases', shape='circle')
    dot.node('S6', 'Final State', shape='doublecircle')

    dot.edge('S0', 'S1', label='^s#')
    dot.edge('S1', 'S2', label='s')
    dot.edge('S2', 'S3', label='^')
```

```
    dot.edge('S3', 'S4', label='Ends with x, z, s, or o')
    dot.edge('S3', 'S5', label='Other regular cases')
    dot.edge('S4', 'S6', label='Add es')
    dot.edge('S5', 'S6', label='Add s')

    return dot

fst_graph = draw_pluralization_fst()
fst_graph.render('pluralization_fst_image', format='png', cleanup=True)
```

SAMPLE INPUT-OUTPUT

```
fox^s#  :  foxes
boy^s#  :  boys
bus^s#  :  buses
quiz^s#  :  quizzes
dog^s#  :  dogs
hero^s#  :  heroes
glass^s#  :  glasses
tomato^s#  :  tomatoes
```



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(str1, str2):
    m = len(str1)
    n = len(str2)
    dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if str1[i - 1] == str2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = min(dp[i - 1][j] + 1,
                               dp[i][j - 1] + 1,
                               dp[i - 1][j - 1] + 2)

    i, j = m, n
    operations = []

    while i > 0 or j > 0:
        if i > 0 and j > 0 and str1[i - 1] == str2[j - 1]:
            i -= 1
            j -= 1
        elif i > 0 and j > 0 and dp[i][j] == dp[i - 1][j - 1] + 2:
            operations.append(f"Substitute '{str1[i - 1]}' with '{str2[j - 1]}' (cost 2)")
            i -= 1
            j -= 1
        elif i > 0 and dp[i][j] == dp[i - 1][j] + 1:
            operations.append(f"Delete '{str1[i - 1]}' (cost 1)")
            i -= 1
```

```
        else:
            operations.append(f"Insert '{str2[j - 1]}' (cost 1)")
            j -= 1

    return dp[m][n], operations[::-1]

str1 = "intention"
str2 = "execution"
distance, edit_operations = min_edit_distance(str1, str2)

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

SAMPLE INPUT-OUTPUT

```
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
import nltk
from nltk.util import ngrams
from nltk.tokenize import word_tokenize
from collections import Counter

nltk.download('punkt')

class BigramSpellChecker:
    def __init__(self, corpus):
        self.corpus = corpus.lower()
        self.vocabulary = set()
        self.unigrams = Counter()
        self.bigrams = Counter()
        self.create_vocabulary_and_bigrams()

    def create_vocabulary_and_bigrams(self):
        tokens = word_tokenize(self.corpus)
        self.vocabulary = set(tokens)
        self.unigrams = Counter(tokens)
        self.bigrams = Counter(ngrams(tokens, 2))

    def is_word_in_vocab(self, word):
```

```
        return word in self.vocabulary

    def edit_distance_one(self, 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 edit_distance_two(self, word):
        return set(e2 for e1 in
        self.edit_distance_one(word) for e2 in
        self.edit_distance_one(e1))

    def bigram_probability(self, w1, w2, alpha=1):
        bigram_count = self.bigrams[(w1, w2)]
        unigram_count = self.unigrams[w1]
        vocabulary_size = len(self.vocabulary)
        return (bigram_count + alpha) / (unigram_count + alpha * vocabulary_size)

    def get_candidates(self, word):
        candidates = [w for w in self.edit_distance_one(word) if
        self.is_word_in_vocab(w)]
        if not candidates:
            candidates = [w for w in self.edit_distance_two(word) if
            self.is_word_in_vocab(w)]
        return candidates

    def correct_article(self, prev_word, next_word, misspelled_word):
        if misspelled_word in ["a", "an"]:
            if next_word and next_word[0] in "aeiou":
                return "an"
            else:
                return "a"
        return misspelled_word

    def suggest_correction(self, prev_words, misspelled_word):
        candidates = self.get_candidates(misspelled_word)
        if not candidates:
```

```
            return misspelled_word
        best_candidate = max(
            candidates,
            key=lambda word: self.bigram_probability(prev_words[-2], word) +
            self.bigram_probability(prev_words[-1], word),
            default=misspelled_word
        )
        return best_candidate

    def correct_text(self, text):
        tokens = word_tokenize(text.lower())
        corrected_text = []

        for i, word in enumerate(tokens):
            if not self.is_word_in_vocab(word):
                prev_words = tokens[max(0, i-2):i]
                if i < len(tokens) - 1:
                    next_word = tokens[i + 1]
                else:
                    next_word = ""

                if word in ["a", "an"]:
                    corrected_word = self.correct_article(prev_words[-1]
                    if prev_words else "", next_word, word)
                else:
                    corrected_word = self.suggest_correction(prev_words, word)

                corrected_text.append(corrected_word)
            else:
                corrected_text.append(word)
        corrected_text[0] = corrected_text[0].capitalize()

        return ' '.join(corrected_text)

corpus = """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."""

spell_checker = BigramSpellChecker(corpus)
text_to_correct = "This is an exmple text with sme errors."
corrected_text = spell_checker.correct_text(text_to_correct)
```

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

SAMPLE INPUT-OUTPUT

```
Original Text: This is an exmple text with sme errors.
Corrected Text: This is an example text with some errors .
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

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

nltk.download('movie_reviews')

reviews = [(movie_reviews.raw(fileid), category)
           for category in movie_reviews.categories()
           for fileid in movie_reviews.fileids(category)]

data = pd.DataFrame(reviews, columns=['text', 'label'])

print("Dataset Preview:")
print(data.head())

data['label'] = data['label'].map({'pos': 1, 'neg': 0})

X_train, X_test, y_train, y_test = train_test_split(data['text'], data['label'],
test_size=0.2, random_state=42)

vectorizer = TfidfVectorizer()
X_train_vect = vectorizer.fit_transform(X_train)
X_test_vect = vectorizer.transform(X_test)

class MyNaiveBayes:
    def __init__(self, smoothing_factor=1.0):
        self.smoothing_factor = smoothing_factor
        self.class_probabilities = {}
```

```
        self.word_probabilities = {}
        self.vocab_size = 0

    def fit(self, X, y):
        num_documents = len(y)
        self.vocab_size = X.shape[1]

        unique_classes = np.unique(y)
        self.class_probabilities = {cls: np.log(np.sum(y == cls) / num_documents) for
        cls in unique_classes}

        word_counts_per_class = {cls: np.zeros(self.vocab_size) for cls in unique_classes}
        for idx in range(len(y)):
            current_class = y.iloc[idx]
            word_counts_per_class[current_class] += X[idx].toarray()[0]

        for cls in unique_classes:
            total_word_count = np.sum(word_counts_per_class[cls]) +
            self.smoothing_factor * self.vocab_size
            self.word_probabilities[cls] = (word_counts_per_class[cls] +
            self.smoothing_factor) / total_word_count

    def predict(self, X):
        log_probabilities = []
        for cls in self.class_probabilities:
            log_prob = self.class_probabilities[cls] +
            X.dot(np.log(self.word_probabilities[cls]))
            log_probabilities.append(log_prob)

        return np.argmax(log_probabilities, axis=0)

def evaluate_naive_bayes(k_value):
    model = MyNaiveBayes(smoothing_factor=k_value)
    model.fit(X_train_vect, y_train)

    predictions = model.predict(X_test_vect)

    accuracy = accuracy_score(y_test, predictions)
    report = classification_report(y_test, predictions)
    return accuracy, report
```

```
smoothing_factors = [0.25, 0.75, 1.0]
results = {}

for k in smoothing_factors:
    accuracy, report = evaluate_naive_bayes(k)
    results[k] = (accuracy, report)

for k, (accuracy, report) in results.items():
    print(f"\nResults for smoothing factor k={k}:")
    print(f"Accuracy: {accuracy:.2f}")
    print(f"Classification Report:\n{report}")
```

SAMPLE INPUT-OUTPUT

```
[nltk_data] Downloading package movie_reviews to /root/nltk_data...
[nltk_data]   Package movie_reviews is already up-to-date!
Dataset Preview:
                                                text label
0  plot : two teen couples go to a church party ,...   neg
1  the happy bastard's quick movie review \ndamn ...   neg
2  it is movies like these that make a jaded movi...   neg
3   " quest for camelot " is warner bros . ' firs...   neg
4  synopsis : a mentally unstable man undergoing ...   neg

Results for smoothing factor 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 smoothing factor 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 smoothing factor 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	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

def viterbi_algorithm(observations, states, start_probs, trans_probs, emit_probs):
    T = len(observations)
    N = len(states)
    viterbi = np.zeros((N, T))
    backpointer = np.zeros((N, T), dtype=int)
    state_to_idx = {state: idx for idx, state in enumerate(states)}
    obs_to_idx = {obs: idx for idx, obs in enumerate(observations)}

    for s in range(N):
        viterbi[s, 0] = start_probs[states[s]] *
        emit_probs[states[s]].get(observations[0], 1e-6)
        backpointer[s, 0] = 0

    for t in range(1, T):
        for s in range(N):
            max_prob, max_state = max(
                (viterbi[s_prev, t - 1] * trans_probs[states[s_prev]].get(states[s], 1e-6)
                 * emit_probs[states[s]].get(observations[t], 1e-6), s_prev) for
                 s_prev in range(N)
            )
            viterbi[s, t] = max_prob
            backpointer[s, t] = max_state

    best_last_state = np.argmax(viterbi[:, T - 1])
```

```
    best_path_prob = viterbi[best_last_state, T - 1]
    best_path = [best_last_state]
    for t in range(T - 1, 0, -1):
        best_last_state = backpointer[best_last_state, t]
        best_path.insert(0, best_last_state)

    best_path = [states[state] for state in best_path]
    return best_path, best_path_prob


states = ["NN", "VB", "JJ", "RB"]
observations = ["time", "flies", "fast"]
start_probability = {"NN": 0.5, "VB": 0.25, "JJ": 0.25, "RB": 0}
transition_probability = {
    "NN": {"NN": 0.25, "VB": 0.25, "JJ": 0.5, "RB": 0, "STOP": 0.1},
    "VB": {"NN": 0.25, "VB": 0.25, "JJ": 0.25, "RB": 0.25, "STOP": 0.1},
    "JJ": {"NN": 0, "VB": 0.75, "JJ": 0.25, "RB": 0, "STOP": 0.1},
    "RB": {"NN": 0.5, "VB": 0.25, "JJ": 0.25, "RB": 0, "STOP": 0.1},
}
emission_probability = {
    "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},
}

best_path, best_path_prob = viterbi_algorithm(observations, states, start_probability, tra
print("Most probable POS tag sequence:", best_path)
print("Probability of the sequence:", best_path_prob)
```

SAMPLE INPUT-OUTPUT

```
Most probable POS tag sequence: ['NN', 'VB', 'JJ']
Probability of the sequence: 3.125000000000001e-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 nltk import bigrams
from nltk.probability import FreqDist, ConditionalFreqDist

corpus = [
    "the sun is shining brightly today",
    "it is a beautiful day outside",
    "the weather is perfect for a walk",
    "i love sunny weather",
    "the sky is clear and the sun is shining"
]

corpus = [sentence.split() for sentence in corpus]
flat_corpus = [word.lower() for sentence in corpus for word in sentence]

unigram_freq = FreqDist(flat_corpus)
bigrams_list = list(bigrams(flat_corpus))
bigram_freq = FreqDist(bigrams_list)
bigram_cfd = ConditionalFreqDist(bigrams_list)

def calculate_bigram_probabilities():
    bigram_probabilities = {}
    for bigram in bigram_freq:
        bigram_probabilities[bigram] = bigram_freq[bigram] / unigram_freq[bigram[0]]
    return bigram_probabilities

def calculate_sentence_probability(sentence):
    sentence = sentence.lower().split()
    prob = 1.0
    for i in range(len(sentence) - 1):
        word_pair = (sentence[i], sentence[i + 1])
        word_prob = bigram_probabilities.get(word_pair, 1e-7)
        prob *= word_prob
    return prob
```

```
unigram_freq = FreqDist(flat_corpus)
bigram_probabilities = calculate_bigram_probabilities()

print("Unigram Counts:")
for word, count in unigram_freq.items():
    print(f"{word}: {count}")

print("\nBigram Counts:")
for bigram, count in bigram_freq.items():
    print(f"{bigram}: {count}")

print("\nBigram Probabilities:")
for bigram, prob in bigram_probabilities.items():
    print(f"{bigram}: {prob:.4f}")

test_sentence = "the sun is shining"
sentence_probability = calculate_sentence_probability(test_sentence)
print(f"\nProbability of the sentence '{test_sentence}': {sentence_probability:.4f}")
```

SAMPLE INPUT-OUTPUT

```
Unigram Counts:
the: 4
sun: 2
is: 5
shining: 2
brightly: 1
today: 1
it: 1
a: 2
beautiful: 1
day: 1
outside: 1
weather: 2
perfect: 1
for: 1
walk: 1
i: 1
love: 1
sunny: 1
sky: 1
clear: 1
and: 1
```

```
Bigram Counts:
('the', 'sun'): 2
('sun', 'is'): 2
('is', 'shining'): 2
('shining', 'brightly'): 1
('brightly', 'today'): 1
('today', 'it'): 1
('it', 'is'): 1
('is', 'a'): 1
('a', 'beautiful'): 1
('beautiful', 'day'): 1
('day', 'outside'): 1
('outside', 'the'): 1
('the', 'weather'): 1
('weather', 'is'): 1
('is', 'perfect'): 1
('perfect', 'for'): 1
('for', 'a'): 1
('a', 'walk'): 1
('walk', 'i'): 1
('i', 'love'): 1
('love', 'sunny'): 1
('sunny', 'weather'): 1
('weather', 'the'): 1
('the', 'sky'): 1
('sky', 'is'): 1
('is', 'clear'): 1
('clear', 'and'): 1
('and', 'the'): 1
```

```
Bigram Probabilities:
('the', 'sun'): 0.5000
('sun', 'is'): 1.0000
('is', 'shining'): 0.4000
('shining', 'brightly'): 0.5000
('brightly', 'today'): 1.0000
('today', 'it'): 1.0000
('it', 'is'): 1.0000
('is', 'a'): 0.2000
('a', 'beautiful'): 0.5000
('beautiful', 'day'): 1.0000
('day', 'outside'): 1.0000
('outside', 'the'): 1.0000
('the', 'weather'): 0.2500
('weather', 'is'): 0.5000
('is', 'perfect'): 0.2000
('perfect', 'for'): 1.0000
('for', 'a'): 1.0000
('a', 'walk'): 0.5000
('walk', 'i'): 1.0000
('i', 'love'): 1.0000
('love', 'sunny'): 1.0000
('sunny', 'weather'): 1.0000
('weather', 'the'): 0.5000
('the', 'sky'): 0.2500
('sky', 'is'): 1.0000
('is', 'clear'): 0.2000
('clear', 'and'): 1.0000
('and', 'the'): 1.0000
```

```
Probability of the sentence 'the sun is shining': 0.2000
```

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

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

documents = [
    "The quick brown fox jumps over the lazy dog",
    "A fast, brown fox leaps over the lazy dog",
    "A lazy dog sleeps peacefully in the sun",
]

vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(documents)
print("TF-IDF Matrix:\n", tfidf_matrix.toarray())

cosine_sim_doc = cosine_similarity(tfidf_matrix[0:1], tfidf_matrix[1:2])
print("Cosine Similarity between Document 1 and Document 2:", cosine_sim_doc[0][0])

def word_cosine_similarity(word1, word2, vectorizer, tfidf_matrix):
    try:
        idx1 = vectorizer.vocabulary_[word1]
        idx2 = vectorizer.vocabulary_[word2]
        word_vec1 = tfidf_matrix[:, idx1].toarray().flatten()
        word_vec2 = tfidf_matrix[:, idx2].toarray().flatten()
        return cosine_similarity([word_vec1], [word_vec2])[0][0]
    except KeyError:
        return 0

print("Cosine Similarity between 'fox' and 'dog':",
word_cosine_similarity('fox', 'dog', vectorizer, tfidf_matrix))
print("Cosine Similarity between 'quick' and 'lazy':",
word_cosine_similarity('quick', 'lazy', vectorizer, tfidf_matrix))
```

SAMPLE INPUT-OUTPUT

```
TF-IDF Matrix:
 [[0.31502724 0.24464675 0.         0.31502724 0.         0.41422296
  0.24464675 0.         0.31502724 0.         0.41422296 0.
  0.         0.4892935 ]
 [0.34779526 0.27009405 0.45730897 0.34779526 0.         0.
  0.27009405 0.45730897 0.34779526 0.         0.         0.
  0.         0.27009405]
 [0.         0.26291231 0.         0.         0.44514923 0.
  0.26291231 0.         0.         0.44514923 0.         0.44514923
  0.44514923 0.26291231]]
Cosine Similarity between Document 1 and Document 2: 0.5930054637135612
Cosine Similarity between 'fox' and 'dog': 0.810975720608166
Cosine Similarity between 'quick' and 'lazy': 0.5444320046508312
```

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
import math
from collections import Counter
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import CountVectorizer

def preprocess_text(corpus):
    return [doc.lower().split() for doc in corpus]

def compute_cooccurrence_matrix(corpus, window_size=2):
    word_counts = Counter()
    cooccurrence_matrix = Counter()

    for doc in corpus:
        for i, word in enumerate(doc):
            word_counts[word] += 1
            context_window = doc[max(i - window_size,
            0):min(i + window_size + 1, len(doc))]
            for context_word in context_window:
                if context_word != word:
                    cooccurrence_matrix[(word, context_word)] += 1

    return word_counts, cooccurrence_matrix

def compute_ppmi_matrix(word_counts, cooccurrence_matrix):
    total_words = sum(word_counts.values())
    ppmi_matrix = {}

    for (word1, word2), cooccurrence_count in cooccurrence_matrix.items():
        pmi = math.log((cooccurrence_count * total_words) / (word_counts[word1] *
        word_counts[word2])) if cooccurrence_count > 0 else 0
        ppmi_matrix[(word1, word2)] = max(0, pmi)
```

```
    return ppmi_matrix

def compute_cosine_similarity(vec1, vec2):
    return cosine_similarity([vec1], [vec2])[0][0]

def document_to_ppmi_vector(doc, ppmi_matrix, word_counts):
    vec = np.zeros(len(word_counts))
    word_to_idx = {word: idx for idx, word in enumerate(word_counts.keys())}

    for i, word1 in enumerate(doc):
        for word2 in doc[i+1:]:
            if (word1, word2) in ppmi_matrix:
                idx1, idx2 = word_to_idx[word1], word_to_idx[word2]
                vec[idx1] += ppmi_matrix.get((word1, word2), 0)
                vec[idx2] += ppmi_matrix.get((word1, word2), 0)

    return vec

corpus = [
    "time flies like an arrow",
    "fruit flies like a banana",
    "time is a great teacher"
]

processed_corpus = preprocess_text(corpus)

word_counts, cooccurrence_matrix =
compute_cooccurrence_matrix(processed_corpus, window_size=2)

ppmi_matrix = compute_ppmi_matrix(word_counts, cooccurrence_matrix)

ppmi_vectors = []
for doc in processed_corpus:
    ppmi_vectors.append(document_to_ppmi_vector(doc, ppmi_matrix, word_counts))

doc1_idx, doc2_idx = 0, 1
cosine_sim_doc1_doc2 = compute_cosine_similarity(ppmi_vectors[doc1_idx],
ppmi_vectors[doc2_idx])

word1 = 'time'
word2 = 'flies'
```

```
word_vector1 = np.zeros(len(word_counts))
word_vector2 = np.zeros(len(word_counts))

for i, word in enumerate(word_counts.keys()):
    if (word1, word) in ppmi_matrix:
        word_vector1[i] += ppmi_matrix.get((word1, word), 0)
    if (word2, word) in ppmi_matrix:
        word_vector2[i] += ppmi_matrix.get((word2, word), 0)

cosine_sim_word1_word2 = compute_cosine_similarity(word_vector1, word_vector2)

print(f"Cosine Similarity between Document 1 and Document 2: {cosine_sim_doc1_doc2}")
print(f"Cosine Similarity between '{word1}' and '{word2}': {cosine_sim_word1_word2}")
```

SAMPLE INPUT-OUTPUT

```
Cosine Similarity between Document 1 and Document 2: 0.5640385137792641
Cosine Similarity between 'time' and 'flies': 0.3652710903405736
```

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

```
import math

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"),
]

vocab = set()
for sentence, _ in training_data:
    vocab.update(sentence.lower().split())

word_counts = {"fish": {}, "guitar": {}}
sense_counts = {"fish": 0, "guitar": 0}

for sentence, sense in training_data:
    words = sentence.lower().split()
    sense_counts[sense] += len(words)
    for word in words:
```

```
        if word not in word_counts[sense]:
            word_counts[sense][word] = 0
        word_counts[sense][word] += 1

total_senses = sum(sense_counts.values())
prior_probs = {sense: sense_count / total_senses for
sense, sense_count in sense_counts.items()}

vocab_size = len(vocab)

def get_conditional_prob(word, sense):
    count_w_c = word_counts[sense].get(word, 0) + 1
    count_c = sense_counts[sense] + vocab_size
    return count_w_c / count_c

test_sentence = "He loves jazz. The bass line provided
the foundation for the guitar solo in the jazz piece"
test_word = "bass"

sentence_words = test_sentence.lower().split()
sense_probs = {}

for sense in prior_probs:
    prob = math.log(prior_probs[sense])
    for word in sentence_words:
        prob += math.log(get_conditional_prob(word, sense))
    sense_probs[sense] = prob

predicted_sense = max(sense_probs, key=sense_probs.get)
print(f"Test Word: {test_word}")
print(f"Predicted Sense: {predicted_sense}")
```

SAMPLE INPUT-OUTPUT

```
Test Word: bass
Predicted Sense: guitar
```