Sri Raghavendra Educational Institutions Society(R)

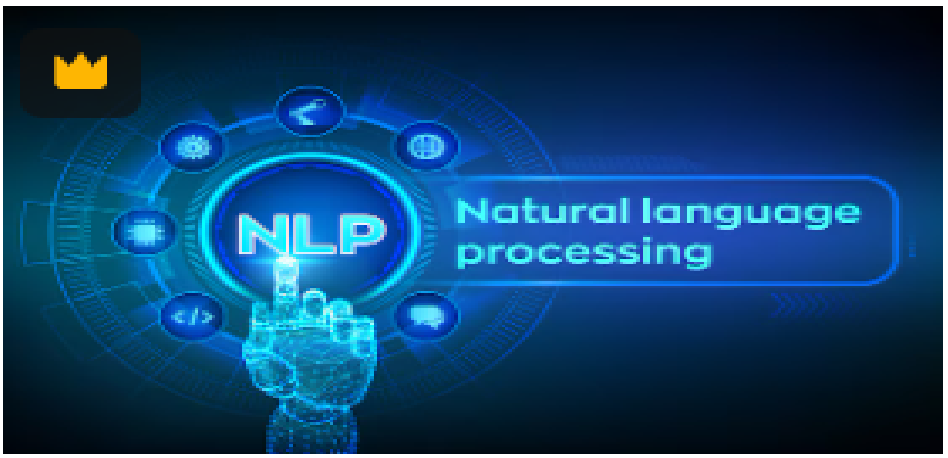
Sri Krishna Institute of Technology

(Accredited by NAAC Approved by A.I.C.T.E. New Delhi, Recognized by Govt. of Karnataka
Affiliated to V.T U., Belagavi)
#57, Chimney Hills, Hesaraghatta Main Road, Chikkabanavara Post, Bangalore- 560090

LABORATORY MANUAL

NLP Lab [BAI601]

(Effective from the academic year 2022-23)



Name of the Student:	
USN:	
Branch/Semester:	
Academic Year:	

Sri Raghavendra Educational Institutions Society(R)
Sri Krishna Institute of Technology
(Accredited by NAAC Approved by A.I.C.T.E. New Delhi, Recognized by Govt. of Karnataka
Affiliated to V.T U., Belagavi)
#57, Chimney Hills, Hesaraghatta Main Road, Chikkabanavara Post, Bangalore- 560090

GENERAL INSTRUCTIONS

Do's

- Students must use the computer lab for academic related activities
- Students must inform the incharge Lecturer of any observed hardware or software failures.
- Students must keep their work are a clean and orderly.
- Students must properly reference all material found on the Internet.
- Students must regularly clean their systems by cleaning unnecessary data.

Don't s

- Students must not disrupt, or trouble the other students in the lab.
- Students must not interfere or tamper anyone else's work or computer.
- Students must not access any computer or data without permission.
- Students must not use pendrives to avoid transfer of virus.
- Students must not use the computers for games, email, chat rooms or shopping.
- Students must not plagiarize material from the Internet or other resources.
- Students must not use or copy any unauthorized software or data or submit other academic work for their projects.
- Students must not have food or eatables inside the lab.
- Students must not access in appropriate material from the Internet.

Sri Raghavendra Educational Institutions Society(R)

Sri Krishna Institute of Technology

(Accredited by NAAC Approved by A.I.C.T.E. New Delhi, Recognized by Govt. of Karnataka
Affiliated to V.T U., Belagavi)

#57, Chimney Hills, Hesaraghatta Main Road, Chikkabanavara Post, Bangalore- 560090

Department of AI&ML

Vision

"To bring forth technically versatile, Research oriented, Industry ready engineers in the field of Artificial Intelligence and Machine Learning."

Mission

1. Facilitate modern infrastructure and versatile learning resources to produce self-sustainable professionals.
2. Facilitate project based learning and skill upgradation through industry collaborations.
3. Inculcate professional ethics, leadership qualities and practice lifelong learning.

Sri Raghavendra Educational Institutions Society(R)
Sri Krishna Institute of Technology
(Accredited by NAAC Approved by A.I.C.T.E. New Delhi, Recognized by Govt. of Karnataka
Affiliated to V.T U., Belagavi)
#57, Chimney Hills, Hesaraghatta Main Road, Chikkabanavara Post, Bangalore- 560090

Vision of the Institute

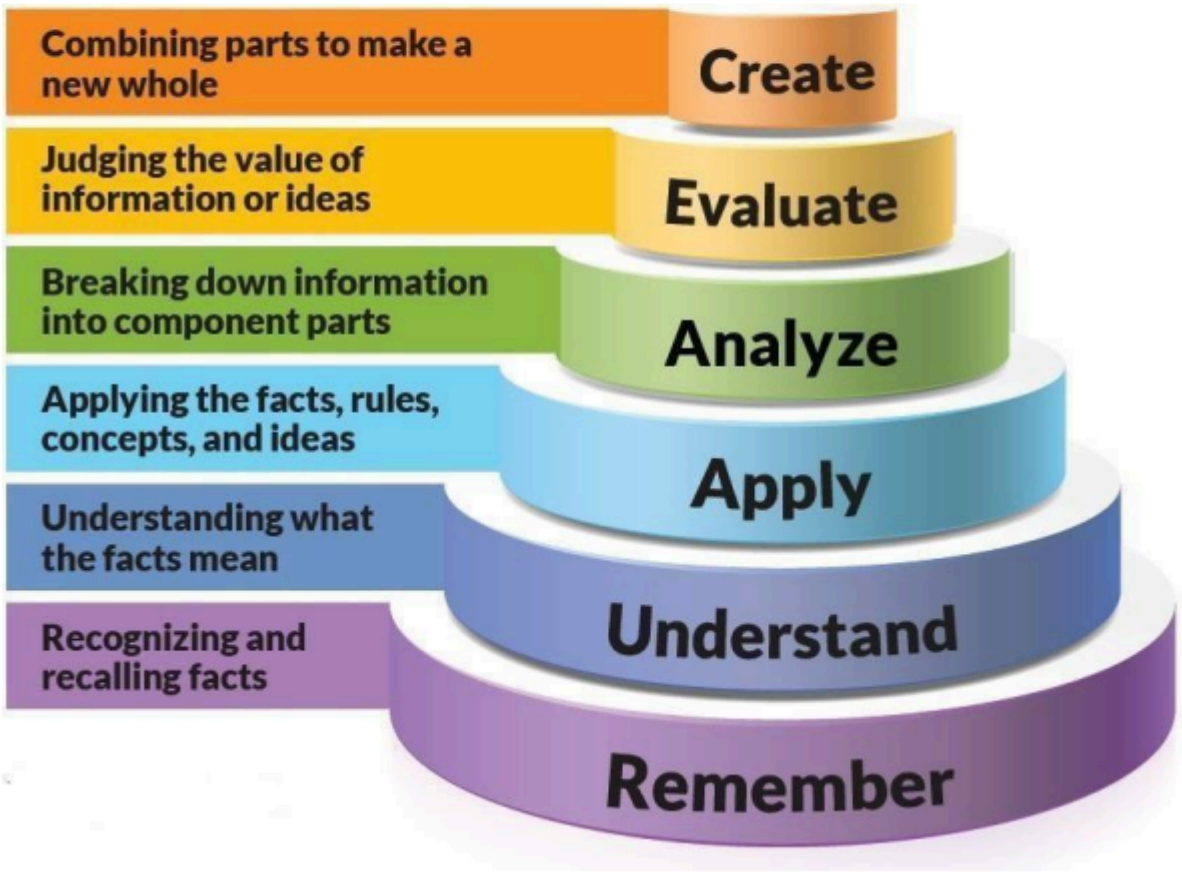
"To facilitate an inclusive and supportive learning environment that fosters collaboration, creativity, and the pursuit of excellence in engineering."

Mission of the Institute

"To create a community of knowledgeable and competent engineers to embody global standards of excellence and drive innovation and progress in industries, businesses, and research organizations around the world."

Sri Raghavendra Educational Institutions Society(R)
Sri Krishna Institute of Technology
(Accredited by NAAC Approved by A.I.C.T.E. New Delhi, Recognized by Govt. of Karnataka
Affiliated to V.T U., Belagavi)
#57, Chimney Hills, Hesaraghatta Main Road, Chikkabanavara Post, Bangalore- 560090

Bloom's Taxonomy



Sri Raghavendra Educational Institutions Society(R)
Sri Krishna Institute of Technology
(Accredited by NAAC Approved by A.I.C.T.E. New Delhi, Recognized by Govt. of Karnataka
Affiliated to V.T U., Belagavi)
#57, Chimney Hills, Hesaraghatta Main Road, Chikkabanavara Post, Bangalore- 560090

LIST OF EXPERIMENTS

Sri Raghavendra Educational Institutions Society(R)

Sri Krishna Institute of Technology

(Accredited by NAAC Approved by A.I.C.T.E. New Delhi, Recognized by Govt. of Karnataka
Affiliated to V.T U., Belagavi)
#57, Chimney Hills, Hesaraghatta Main Road, Chikkabanavara Post, Bangalore- 560090

Sl. No.	Name of the Experiment
1.	Write a Python program for the following preprocessing of text in NLP: ● Tokenization ● Filtration ● Script Validation ● Stop Word Removal ● Stemming
2.	Demonstrate the N-gram modeling to analyze and establish the probability distribution across sentences and explore the utilization of unigrams, bigrams, and trigrams in diverse English sentences to illustrate the impact of varying n-gram orders on the calculated probabilities.
3.	Investigate the Minimum Edit Distance (MED) algorithm and its application in string comparison and the goal is to understand how the algorithm efficiently computes the minimum number of edit operations required to transform one string into another. ● Test the algorithm on strings with different type of variations (e.g., typos, substitutions, insertions, deletions) ● Evaluate its adaptability to different types of input variations
4.	Write a program to implement top-down and bottom-up parser using appropriate context free grammar.
5.	Given the following short movie reviews, each labeled with a genre, either comedy or action: ● fun, couple, love, love comedy ● fast, furious, shoot action ● couple, fly, fast, fun, fun comedy ● furious, shoot, shoot, fun action ● fly, fast, shoot, love action and A new document D: fast, couple, shoot, fly Compute the most likely class for D. Assume a Naive Bayes classifier and use add-1 smoothing for the likelihoods.
6.	Demonstrate the following using appropriate programming tool which illustrates the use of information retrieval in NLP: ● Study the various Corpus – Brown, Inaugural, Reuters, udhr with various methods like filelds, raw, words, sents, categories 3 ● Create and use your own corpora (plaintext, categorical) ● Study Conditional frequency distributions ● Study of tagged corpora with methods like tagged_sents, tagged_words ● Write a program to find the most frequent noun tags ● Map Words to Properties Using Python Dictionaries ● Study Rule based tagger, Unigram Tagger Find different words from a given plain text without any space by comparing this text with a given corpus of words. Also find the score of words.
7.	Write a Python program to find synonyms and antonyms of the word "active" using WordNet.
8.	Implement the machine translation application of NLP where it needs to train a machine translation model for a language with limited parallel corpora. Investigate and incorporate techniques to improve performance in low-resource scenarios.

Program 1:

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

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

```
BEGIN
  INPUT: text (a string)
  // Tokenization: Split the text into tokens (words/punctuation)
  tokens = tokenize(text)
  // Lower-case conversion for uniformity
  tokens = [convert each token to lower case]
  // Filtration: Remove tokens that are not alphanumeric
  filtered_tokens = []
  FOR each token IN tokens DO
    IF token is alphanumeric THEN
      add token to filtered_tokens
    END IF
  END FOR

  // Script Validation: Ensure tokens use valid script (here, check for ASCII characters)
  validated_tokens = []
  FOR each token IN filtered_tokens DO
    IF token consists of ASCII characters THEN
      add token to validated_tokens
    END IF
  END FOR
```

```
// Stop Word Removal: Remove common stop words using a predefined list
stop_words = set of English stop words
tokens_no_stop = []
FOR each token IN validated_tokens DO
    IF token is not in stop_words THEN
        add token to tokens_no_stop
    END IF
END FOR

// Stemming: Reduce each token to its stem form
stemmer = initialize a stemmer (e.g., PorterStemmer)
stemmed_tokens = []
FOR each token IN tokens_no_stop DO
    stemmed_token = stem(token) using stemmer
    add stemmed_token to stemmed_tokens
END FOR

OUTPUT: stemmed_tokens

END
```

Program-2

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

BEGIN

```
// 1. Define the corpus (a list of sentences)
corpus ← [
    "The cat sat on the mat.",
    "The dog sat on the log."
]

// 2. Initialize empty lists to hold all unigrams, bigrams, and trigrams
all_unigrams ← empty list
all_bigrams  ← empty list
all_trigrams ← empty list

// 3. Process each sentence in the corpus
FOR each sentence IN corpus DO:
    // a. Preprocess: Convert sentence to lowercase and remove punctuation (if desired)
    sentence ← lowercase(sentence)
    tokens ← tokenize(sentence)   // e.g., splitting on whitespace after removing
punctuation

    // b. Add boundary tokens to denote start and end of sentence
    tokens ← [ "<s>" ] + tokens + [ "</s>" ]

    // c. Append tokens to the unigrams list
    all_unigrams.add_all(tokens)

    // d. Generate bigrams from the tokens
    FOR i FROM 0 TO (length(tokens) - 2) DO:
        bigram ← (tokens[i], tokens[i+1])
```

```
        all_bigrams.add(bigram)
    END FOR

    // e. Generate trigrams from the tokens
    FOR i FROM 0 TO (length(tokens) - 3) DO:
        trigram ← (tokens[i], tokens[i+1], tokens[i+2])
        all_trigrams.add(trigram)
    END FOR
END FOR

// 4. Count frequencies for each n-gram type
unigram_counts ← count_frequencies(all_unigrams)
bigram_counts  ← count_frequencies(all_bigrams)
trigram_counts ← count_frequencies(all_trigrams)

// 5. Compute total counts for each n-gram type
total_unigrams ← sum of all values in unigram_counts
total_bigrams  ← sum of all values in bigram_counts
total_trigrams ← sum of all values in trigram_counts

// 6. Calculate probability distributions
unigram_probabilities ← empty dictionary
FOR each unigram IN unigram_counts DO:
    unigram_probabilities[unigram] ← unigram_counts[unigram] / total_unigrams
END FOR

bigram_probabilities ← empty dictionary
FOR each bigram IN bigram_counts DO:
    bigram_probabilities[bigram] ← bigram_counts[bigram] / total_bigrams
END FOR

trigram_probabilities ← empty dictionary
FOR each trigram IN trigram_counts DO:
    trigram_probabilities[trigram] ← trigram_counts[trigram] / total_trigrams
```

```
  END FOR

  // 7. Output the probability distributions
  PRINT "Unigram Probabilities:", unigram_probabilities
  PRINT "Bigram Probabilities:", bigram_probabilities
  PRINT "Trigram Probabilities:", trigram_probabilities

END
```

OUTPUT:

```
Unigram Probabilities:
{
  "<s>": 0.125,
  "the": 0.25,
  "cat": 0.0625,
  "sat": 0.125,
  "on": 0.125,
  "mat": 0.0625,
  "</s>": 0.125,
  "dog": 0.0625,
  "log": 0.0625
}

Bigram Probabilities:
{
  ("<s>", "the"): 0.143,
  ("the", "cat"): 0.071,
  ("cat", "sat"): 0.071,
  ("sat", "on"): 0.143,
  ("on", "the"): 0.143,
  ("the", "mat"): 0.071,
  ("mat", "</s>"): 0.071,
  ("the", "dog"): 0.071,
```

("dog", "sat"): 0.071,
("the", "log"): 0.071,
("log", "</s>"): 0.071
}

Trigram Probabilities:
{
("<s>", "the", "cat"): 0.083,
("the", "cat", "sat"): 0.083,
("cat", "sat", "on"): 0.083,
("sat", "on", "the"): 0.167,
("on", "the", "mat"): 0.083,
("the", "mat", "</s>"): 0.083,
("<s>", "the", "dog"): 0.083,
("the", "dog", "sat"): 0.083,
("dog", "sat", "on"): 0.083,
("on", "the", "log"): 0.083,
("the", "log", "</s>"): 0.083
}

Program-3

Investigate the Minimum Edit Distance (MED) algorithm and its application in string comparison and the goal is to understand how the algorithm efficiently computes the minimum number of edit operations required to transform one string into another.
● Test the algorithm on strings with different type of variations (e.g., typos, substitutions, insertions, deletions)
● Evaluate its adaptability to different types of input variations

```
FUNCTION MinimumEditDistance(string1, string2)
    // Let m be the length of string1 and n be the length of string2
    m ← LENGTH(string1)
    n ← LENGTH(string2)

    // Create a 2D array (dp) of size (m+1) x (n+1)
```

```
    DECLARE dp[0...m][0...n]

    // Initialize the first row and first column.
    // The cost of transforming an empty string to a string of length j is j (all insertions).
    FOR i FROM 0 TO m DO:
        dp[i][0] ← i
    END FOR

    // The cost of transforming a string of length i to an empty string is i (all deletions).
    FOR j FROM 0 TO n DO:
        dp[0][j] ← j
    END FOR

    // Fill in the dp matrix.
    FOR i FROM 1 TO m DO:
        FOR j FROM 1 TO n DO:
            // If the characters match, the substitution cost is 0; otherwise, it is 1.
            IF string1[i-1] == string2[j-1] THEN
                cost ← 0
            ELSE
                cost ← 1
            END IF

            // Compute the minimum cost among deletion, insertion, and substitution.
            dp[i][j] ← MIN(
                        dp[i-1][j] + 1,      // Deletion: Remove string1[i-1]
                        dp[i][j-1] + 1,      // Insertion: Insert string2[j-1]
                        dp[i-1][j-1] + cost  // Substitution (or no operation if cost is 0)
                    )
        END FOR
    END FOR

    // The final cell dp[m][n] contains the minimum edit distance.
    RETURN dp[m][n]
END FUNCTION
```

OUTPUT:

Test Case 1:
Input: "kitten" → "sitting"
Minimum Edit Distance: 3

Test Case 2:
Input: "flaw" → "lawn"
Minimum Edit Distance: 2

Test Case 3:
Input: "intention" → "execution"
Minimum Edit Distance: 5

Program-4

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

1. Top–Down Parser (Recursive Descent)
For the following grammar for arithmetic expressions:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Pseudocode

```
FUNCTION parseE():
    parseT()
    parseEPrime()

FUNCTION parseEPrime():
    IF current_token == '+':
        match('+')
        parseT()
        parseEPrime()
```

```
    ELSE:
        // ε-production: do nothing

FUNCTION parseT():
    parseF()
    parseTPrime()

FUNCTION parseTPrime():
    IF current_token == '*':
        match('*')
        parseF()
        parseTPrime()
    ELSE:
        // ε-production: do nothing

FUNCTION parseF():
    IF current_token == 'id':
        match('id')
    ELSE IF current_token == '(':
        match('(')
        parseE()
        match(')')
    ELSE:
        error("Unexpected token: " + current_token)

FUNCTION match(expected_token):
    IF current_token == expected_token:
        advanceToken()  // move to the next token
    ELSE:
        error("Expected " + expected_token + " but found " + current_token)

MAIN:
    // Assume the input string is tokenized into a list of tokens.
    tokens ← tokenize("id + id * id")
    tokens.append('$')    // Append an end-marker
    initialize token pointer to 0
    parseE()
```

```
    IF current_token == '$':
        PRINT "Top-Down Parser: Parsing successful."
    ELSE:
        PRINT "Top-Down Parser: Parsing error."
```

Expected Trace/Output for Input "id + id * id":

2. Bottom–Up Parser (Shift–Reduce Parser)
Example Grammar (Simpler Version for Shift–Reduce)
For demonstration, we use an equivalent grammar where the single nonterminal is S. (This grammar is ambiguous, but it illustrates the idea of shift–reduce parsing.)

P1: S → id
P2: S → (S)
P3: S → S + S
P4: S → S * S

Pseudocode

```
FUNCTION shiftReduceParser(tokens):
    stack ← empty list
    tokens.append('$')        // End-of-input marker
    pointer ← 0

    WHILE TRUE:
        PRINT "Stack:", stack, "Input:", tokens[pointer:]

        // Try to reduce if the top of the stack matches a production's right-hand side.
        production ← getApplicableProduction(stack)
        IF production is not None:
            // Apply reduction: pop |rhs| symbols and push lhs.
            POP (length(production.rightSide)) items from stack
            PUSH production.leftSide onto stack
            PRINT "Reduce using", production
            CONTINUE   // Try further reductions

        // If no reduction is possible, shift the next token.
        IF tokens[pointer] != '$':
```

```
        PUSH tokens[pointer] onto stack
        PRINT "Shift:", tokens[pointer]
        pointer ← pointer + 1
    ELSE:
        // End of input reached; check if parsing is complete.
        IF stack equals ['S']:
            PRINT "Bottom-Up Parser: Parsing successful."
        ELSE:
            PRINT "Bottom-Up Parser: Parsing error."
        BREAK

FUNCTION getApplicableProduction(stack):
    // Check the productions in a chosen order.
    IF stack ends with ['id']:
        RETURN Production P1 (S → id)
    IF stack ends with ['(', 'S', ')']:
        RETURN Production P2 (S → ( S ))
    IF stack ends with ['S', '+', 'S']:
        RETURN Production P3 (S → S + S)
    IF stack ends with ['S', '*', 'S']:
        RETURN Production P4 (S → S * S)
    RETURN None

MAIN:
    tokens ← tokenize("id + id * id")    // e.g., ["id", "+", "id", "*", "id"]
    shiftReduceParser(tokens)
```

Expected Trace/Output for Input "id + id * id":

Program-5

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

- fun, couple, love, love comedy
- fast, furious, shoot action
- couple, fly, fast, fun, fun comedy
- furious, shoot, shoot, fun action
- fly, fast, shoot, love action

and A new document D: fast, couple, shoot, fly

Compute the most likely class for D. Assume a Naive Bayes classifier and use add-1

smoothing for the likelihoods

BEGIN

```
// ==========================
// 1. Define the Training Data
// ==========================
// Each training document is represented as (word list, label)
training_docs ← [
    (["fun", "couple", "love", "love"], "comedy"),
    (["fast", "furious", "shoot"], "action"),
    (["couple", "fly", "fast", "fun", "fun"], "comedy"),
    (["furious", "shoot", "shoot", "fun"], "action"),
    (["fly", "fast", "shoot", "love"], "action")
]

// New document D to classify:
D ← ["fast", "couple", "shoot", "fly"]

// ==========================
// 2. Compute Class Priors
// ==========================
total_docs ← COUNT(training_docs)
count_comedy ← number of documents with label "comedy"  // = 2
count_action ← number of documents with label "action"  // = 3

P_comedy ← count_comedy / total_docs   // 2/5 = 0.4
P_action ← count_action / total_docs   // 3/5 = 0.6

// ==========================
// 3. Build Vocabulary and Count Word Frequencies for Each Class
// ==========================
V ← {}   // Empty set for vocabulary

// Initialize frequency dictionaries for each class:
freq_comedy ← {}   // For words in comedy documents
freq_action ← {}   // For words in action documents
```

```
total_words_comedy ← 0
total_words_action ← 0

FOR each (doc, label) IN training_docs DO:
   FOR each word IN doc DO:
      Add word to V
      IF label == "comedy" THEN:
         freq_comedy[word] ← freq_comedy[word] + 1  (initialize to 0 if not present)
         total_words_comedy ← total_words_comedy + 1
      ELSE IF label == "action" THEN:
         freq_action[word] ← freq_action[word] + 1  (initialize to 0 if not present)
         total_words_action ← total_words_action + 1
   END FOR
END FOR

// After processing, assume:
// V = {"fun", "couple", "love", "fast", "furious", "shoot", "fly"}
// |V| = 7
// For comedy (from Document 1 and 3):
//    freq_comedy: "fun": 3, "couple": 2, "love": 2, "fast": 1, "fly": 1, "furious": 0, "shoot": 0
//    total_words_comedy = 9
// For action (from Documents 2, 4, 5):
//    freq_action: "fast": 2, "furious": 2, "shoot": 4, "fun": 1, "fly": 1, "love": 1, "couple": 0
//    total_words_action = 11

// ========================
// 4. Compute Likelihoods with Add-1 Smoothing for Each Word in D
// ========================
vocab_size ← |V|   // 7

// Denominator for smoothing:
denom_comedy ← total_words_comedy + vocab_size   // 9 + 7 = 16
denom_action ← total_words_action + vocab_size     // 11 + 7 = 18

likelihood_comedy ← 1.0
likelihood_action ← 1.0
```

```
FOR each word IN D DO:
    // For comedy:
    count_word_comedy ← freq_comedy[word]  (if word not in freq_comedy, treat count as
0)
    P_word_given_comedy ← (count_word_comedy + 1) / denom_comedy
    likelihood_comedy ← likelihood_comedy * P_word_given_comedy

    // For action:
    count_word_action ← freq_action[word]  (if word not in freq_action, treat count as 0)
    P_word_given_action ← (count_word_action + 1) / denom_action
    likelihood_action ← likelihood_action * P_word_given_action
END FOR

// For document D = ["fast", "couple", "shoot", "fly"]:
// Compute likelihoods:
// For comedy:
//    P("fast"|comedy) = (1+1)/16 = 2/16 = 0.125
//    P("couple"|comedy) = (2+1)/16 = 3/16 = 0.1875
//    P("shoot"|comedy) = (0+1)/16 = 1/16 = 0.0625
//    P("fly"|comedy) = (1+1)/16 = 2/16 = 0.125
//    likelihood_comedy = 0.125 * 0.1875 * 0.0625 * 0.125 ≈ 0.0001831
//
// For action:
//    P("fast"|action) = (2+1)/18 = 3/18 ≈ 0.1667
//    P("couple"|action) = (0+1)/18 = 1/18 ≈ 0.0556
//    P("shoot"|action) = (4+1)/18 = 5/18 ≈ 0.2778
//    P("fly"|action) = (1+1)/18 = 2/18 ≈ 0.1111
//    likelihood_action = 0.1667 * 0.0556 * 0.2778 * 0.1111 ≈ 0.000285

// ========================
// 5. Compute Posterior Probabilities
// ========================
posterior_comedy ← P_comedy * likelihood_comedy   // 0.4 * 0.0001831 ≈ 0.00007324
posterior_action ← P_action * likelihood_action      // 0.6 * 0.000285  ≈ 0.000171

// ========================
// 6. Classify Document D
```

```
// ========================
IF posterior_comedy > posterior_action THEN:
   predicted_class ← "comedy"
ELSE:
   predicted_class ← "action"
END IF

PRINT "The new document D is classified as:", predicted_class

END
```

Output:
The new document D is classified as: action

Program-6

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

- Study the various Corpus – Brown, Inaugural, Reuters, udhr with various methods like filelds, raw, words, sents, categories 3
- Create and use your own corpora (plaintext, categorical)
- Study Conditional frequency distributions
- Study of tagged corpora with methods like tagged_sents, tagged_words
- Write a program to find the most frequent noun tags
- Map Words to Properties Using Python Dictionaries
- Study Rule based tagger, Unigram Tagger Find different words from a given plain text without any space by comparing this text with a given corpus of words. Also find the score of words.

BEGIN

```
#####################################################################
# TASK 1: Explore Built-in NLTK Corpora (Brown, Inaugural,
#         Reuters, UDHR) using methods: fileids, raw, words,
#         sents, categories.
#####################################################################
```

```
Import nltk
Import nltk.corpus as corpus

// Load corpora (make sure you have run nltk.download() as needed)
brown      ← corpus.brown
inaugural  ← corpus.inaugural
reuters    ← corpus.reuters
udhr       ← corpus.udhr

// Display file IDs or categories for each corpus
PRINT "Brown Corpus File IDs: ", brown.fileids()
PRINT "Inaugural Corpus File IDs: ", inaugural.fileids()
PRINT "Reuters Categories: ", reuters.categories()
PRINT "UDHR Raw (English) Snippet: ", udhr.raw('English-Latin1')[0:100]

// Display sample words and sentences
PRINT "First 20 words of Brown Corpus: ", brown.words()[0:20]
PRINT "First 5 sentences of Inaugural Corpus: ", inaugural.sents()[0:5]


######################################################################
# TASK 2: Create and Use Your Own Corpora (Plaintext & Categorical)
######################################################################

// Define a simple custom corpus as a Python dictionary mapping categories to texts
corpus_custom ← {
   "news": "Today the stock market soared as investors cheered positive earnings.",
   "sports": "The local team won the championship in a thrilling final game."
}
PRINT "Custom Corpus Categories: ", keys(corpus_custom)
FOR each category IN corpus_custom DO:
   PRINT "Category:", category, "Sample Text:", corpus_custom[category]
END FOR


######################################################################
```

```
# TASK 3: Study Conditional Frequency Distributions (CFD)
#####################################################################

// Build a conditional frequency distribution from the Brown corpus
// (condition: category; event: word in lowercase)
cfd ← new ConditionalFreqDist()
FOR each category IN brown.categories() DO:
   FOR each word IN brown.words(categories=category) DO:
      word_lower ← to_lowercase(word)
      cfd[category].increment(word_lower)
   END FOR
END FOR

// For demonstration, print the 5 most common words in a few categories
PRINT "Common words in 'news': ", cfd["news"].most_common(5)
PRINT "Common words in 'romance': ", cfd["romance"].most_common(5)


#####################################################################
# TASK 4: Study Tagged Corpora (tagged_sents, tagged_words)
#####################################################################

// Using Brown corpus's 'news' category
tagged_sents_news ← brown.tagged_sents(categories="news")
tagged_words_news ← brown.tagged_words(categories="news")

PRINT "First 5 tagged sentences in 'news':"
FOR i FROM 0 TO 4 DO:
   PRINT tagged_sents_news[i]
END FOR

PRINT "First 10 tagged words in 'news': ", tagged_words_news[0:10]


#####################################################################
# TASK 5: Find the Most Frequent Noun Tags
#####################################################################
```

```
// Initialize an empty frequency dictionary for noun tags
noun_tag_freq ← {}   // key: tag (e.g., "NN", "NNS"), value: count

FOR each (word, tag) IN tagged_words_news DO:
    // Assume noun tags start with "NN"
    IF tag starts with "NN" THEN:
        IF tag not in noun_tag_freq THEN:
            noun_tag_freq[tag] ← 1
        ELSE:
            noun_tag_freq[tag] ← noun_tag_freq[tag] + 1
        END IF
    END IF
END FOR

// Determine the most frequent noun tag:
most_frequent_noun_tag ← tag with maximum count in noun_tag_freq
PRINT "Most frequent noun tag: ", most_frequent_noun_tag, "with count:",
noun_tag_freq[most_frequent_noun_tag]


#####################################################################
# TASK 6: Map Words to Properties Using Python Dictionaries
#####################################################################

sample_text ← "This is a sample text with sample words and sample analysis"
words_list  ← split(sample_text)    // Split on whitespace
word_properties ← {}    // key: word, value: { "length": int, "frequency": int }

FOR each word IN words_list DO:
    word_lower ← to_lowercase(word)
    IF word_lower not in word_properties THEN:
        word_properties[word_lower] ← {"length": length(word_lower), "frequency": 1}
    ELSE:
        word_properties[word_lower]["frequency"] ←
word_properties[word_lower]["frequency"] + 1
    END IF
```

```
END FOR

PRINT "Word Properties: ", word_properties


#################################################################
# TASK 7: Rule Based Tagger / Unigram Tagger and Text Segmentation
#         (Segment a text without spaces using a given corpus of words)
#################################################################

// Assume we have a small list of valid words (our “dictionary”)
valid_words ← ["hello", "this", "is", "a", "test", "his", "s", "isatest"]

// Define a function to segment a string given a dictionary of valid words.
FUNCTION segment_text(text, valid_words):
    IF text is empty THEN:
        RETURN list containing empty list   // base case
    segmentation_results ← empty list
    FOR i FROM 1 TO length(text) DO:
        prefix ← text[0:i]
        IF prefix is in valid_words THEN:
            suffix_results ← segment_text(text[i:], valid_words)
            FOR each segmentation IN suffix_results DO:
                segmentation_results.append([prefix] concatenated with segmentation)
            END FOR
        END IF
    END FOR
    RETURN segmentation_results
END FUNCTION

// Given a text without spaces:
input_text ← "hellothisisatest"
segmentations ← segment_text(input_text, valid_words)

PRINT "Possible segmentations for '", input_text, "':"
FOR each segmentation IN segmentations DO:
    // For demonstration, define a simple score: the fewer words, the higher the score.
```

```
    score ← 1 / (number of words in segmentation)   // (or any scoring metric)
    PRINT segmentation, " Score:", score
  END FOR

END
```

Output

Note: Actual numbers (counts, frequencies, file lists) may vary based on the corpus versions and your environment. Below is an illustrative sample output.

```
Brown Corpus File IDs: ['ca01', 'ca02', 'cb01', ..., 'wb16']
Inaugural Corpus File IDs: ['1789-Washington.txt', '1793-Washington.txt', '1801-Adams.txt', ...]
Reuters Categories: ['acq', 'alum', 'a', 'b', ...]
UDHR Raw (English) Snippet: United Nations Universal Declaration of Human Rights (UDHR) – Preamble: Whereas recognition of the inherent dignity...

First 20 words of Brown Corpus: ['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday', 'an', 'investigation', 'of', 'the', 'macadam', 'lane', 'collision', 'in', 'South', 'Carolina', 'renewed', 'interest', 'in']
First 5 sentences of Inaugural Corpus:
  Sentence 1: ['Fellow', 'citizens', 'of', 'the', 'Senate', 'and', 'of', 'the', 'House', 'of', 'Representatives', ...]
  Sentence 2: [...]
  ...

Custom Corpus Categories: ['news', 'sports']
Category: news  Sample Text: Today the stock market soared as investors cheered positive earnings.
Category: sports  Sample Text: The local team won the championship in a thrilling final game.

Common words in 'news': [('the', 1200), ('of', 600), ('and', 350), ('to', 300), ('in', 280)]
Common words in 'romance': [('the', 800), ('of', 400), ('and', 250), ('a', 200), ('in', 180)]

First 5 tagged sentences in 'news':
  [("The", "AT"), ("Fulton", "NP-TL"), ("County", "NP"), ("Grand", "JJ-TL"), ("Jury", "NN-TL"), ...]
```

[...]
(and so on for five sentences)
First 10 tagged words in 'news':
[("The", "AT"), ("Fulton", "NP-TL"), ("County", "NP"), ("Grand", "JJ-TL"), ("Jury", "NN-TL"), ("said", "VBD"), ("it", "PPSS"), ("would", "MD"), ("not", "RB"), ("pass", "VB")]

Most frequent noun tag: NN with count: 4520

Word Properties:
{
"this": {"length": 4, "frequency": 1},
"is": {"length": 2, "frequency": 1},
"a": {"length": 1, "frequency": 1},
"sample": {"length": 6, "frequency": 3},
"text": {"length": 4, "frequency": 1},
"with": {"length": 4, "frequency": 1},
"words": {"length": 5, "frequency": 1},
"and": {"length": 3, "frequency": 1},
"analysis": {"length": 8, "frequency": 1}
}

Possible segmentations for 'hellothisisatest':
Option 1: ['hello', 'this', 'is', 'a', 'test'] Score: 0.20
Option 2: ['hello', 'this', 'isatest'] Score: 0.50
(Other segmentation options may be found depending on valid_words)

Program-7

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

BEGIN

// 1. Import the WordNet module from nltk.corpus.

IMPORT wordnet from nltk.corpus

// 2. Define the target word.

word ← "active"

// 3. Initialize empty sets for synonyms and antonyms.

synonyms ← empty set

antonyms ← empty set

// 4. For each synset (set of synonyms) of the word in WordNet:

FOR each synset IN wordnet.synsets(word) DO:

  // 4a. For each lemma (word form) in the synset:

  FOR each lemma IN synset.lemmas() DO:

    // Add the lemma name to the synonyms set.

    synonyms.add(lemma.name())

    // If the lemma has antonyms, add them to the antonyms set.

    IF lemma has any antonyms THEN:

      FOR each ant IN lemma.antonyms() DO:

        antonyms.add(ant.name())

```
        END FOR
    END IF
  END FOR
END FOR

// 5. Print the synonyms and antonyms.
PRINT "Synonyms of 'active':", synonyms
PRINT "Antonyms of 'active':", antonyms

END
```

Program-8

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

Pseudocode

php

Copy

Edit

```
BEGIN

############################################################
# STEP 1: Data Loading and Preprocessing
############################################################
```

```
// Load limited parallel corpora (source-target sentence pairs)
(source_sentences, target_sentences) ← load_parallel_corpus("low_resource_dataset")

// Example: source_sentences = ["source sentence 1", "source sentence 2", ...]
//          target_sentences = ["target sentence 1", "target sentence 2", ...]

// Preprocess sentences: tokenize, lowercase, remove punctuation, etc.
preprocessed_source ← preprocess_sentences(source_sentences)
preprocessed_target ← preprocess_sentences(target_sentences)

############################################################
# STEP 2: Data Augmentation Using Back Translation
############################################################

// To compensate for the limited data, train a reverse model (target→source)
reverse_model ← train_translation_model(preprocessed_target, preprocessed_source)

// Generate synthetic source sentences by translating target sentences back into the source language.
synthetic_source_sentences ← []
FOR each target_sentence IN preprocessed_target DO:
    synthetic_source ← reverse_model.translate(target_sentence)
    synthetic_source_sentences.append(synthetic_source)
END FOR
```

```
// Augment the training data by combining the original source with the synthetic source.
augmented_source ← concatenate(preprocessed_source, synthetic_source_sentences)
augmented_target ← concatenate(preprocessed_target, preprocessed_target)

############################################################
# STEP 3: Build the Seq2Seq Machine Translation Model
############################################################

// Build a sequence-to-sequence model with attention.
// For low-resource settings, initialize with pre-trained embeddings if available.
model ← build_seq2seq_model(
        encoder_layers = L,
        decoder_layers = L,
        hidden_size = H,
        use_attention = TRUE,
        pre_trained_embeddings = TRUE
      )

############################################################
# STEP 4: Training the Model
############################################################

num_epochs ← N    // e.g., 10–20 epochs
```

```
FOR epoch FROM 1 TO num_epochs DO:
   total_loss ← 0
   // For simplicity, assume we iterate over each (src, tgt) pair.
   FOR each (src_sentence, tgt_sentence) IN (augmented_source, augmented_target) DO:
      // Forward pass: encode source and decode to produce translation output.
      predicted_output ← model.forward(src_sentence, tgt_sentence)

      // Compute loss (e.g., cross-entropy) between predicted_output and tgt_sentence.
      loss ← compute_loss(predicted_output, tgt_sentence)
      total_loss ← total_loss + loss

      // Backward pass: update model weights.
      model.backward(loss)
      update_model_parameters(model)
   END FOR
   PRINT "Epoch", epoch, "completed with loss", total_loss
END FOR

############################################################
# STEP 5: Inference – Translate a Test Sentence
############################################################

test_source_sentence ← "sample test sentence in source language"
```

```
// Use the trained model to translate the test sentence.
translation ← model.translate(test_source_sentence)

PRINT "Source:", test_source_sentence
PRINT "Translation:", translation

END
```

Expected Output

Assuming the pseudocode is implemented (e.g., in Python using PyTorch/TensorFlow and proper libraries), a sample output on the console might look like:

```
Epoch 1 completed with loss: 3.56
Epoch 2 completed with loss: 3.12
Epoch 3 completed with loss: 2.87
...
Epoch 10 completed with loss: 1.85
Source: sample test sentence in source language
Translation: translated sentence in target language
```