SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 3rd semester

Quantum Machine Learning (PGI20G04J)- Lab Manual

Lab 1: Bayes' Theorem Application

Title

Applying Bayes' Rule to Calculate Conditional Probability

Aim

To calculate the probability of a student being absent given that it is Friday, using Bayes' Rule in Python.

Procedure

1. Define the known probabilities:
   - P(Friday and Absent): Probability that it is Friday AND a student is absent.
   - P(Friday): Probability that it is Friday.
2. Apply Bayes' Rule formula: $P(A|B)=P(B|A)*P(A)/P(B)$ In this case, we want to find P(Absent | Friday). The formula can be simplified to P(Absent | Friday)=P(Friday and Absent)/P(Friday).
3. Implement the calculation in Python.
4. Print the result.

Source Code

```
# Lab 1: Bayes' Theorem Application

# Given probabilities
prob_friday_and_absent = 3 / 60  # Probability that it is Friday and a
student is absent
prob_friday = 20 / 100           # Probability that it is Friday (20%)

# Calculate the probability that a student is absent given that today is
Friday
# Using the formula: P(Absent | Friday) = P(Friday and Absent) / P(Friday)
if prob_friday != 0:
    prob_absent_given_friday = prob_friday_and_absent / prob_friday
    print(f"The probability that a student is absent given that today is
Friday is: {prob_absent_given_friday:.2f}")
else:
    print("Error: Probability of Friday cannot be zero.")
```

Input

No direct user input is required as the probabilities are hardcoded in the script.

Expected Output

```
The probability that a student is absent given that today is Friday is: 0.25
```

Lab 2: Data Extraction from Database

Title

Extracting Data from a Database using Python

Aim

To demonstrate how to connect to a database (e.g., SQLite) and extract data from a table using Python.

Procedure

1. Import the `sqlite3` module.
2. Connect to a SQLite database file (or create one if it doesn't exist).
3. Create a cursor object.
4. (Optional) Create a table and insert some sample data if the database is new or empty.
5. Execute a SQL `SELECT` query to retrieve data from the table.
6. Fetch all the results.
7. Iterate through the fetched data and print each row.
8. Close the database connection.

Source Code

```
# Lab 2: Data Extraction from Database

import sqlite3

def extract_data_from_db(db_name="my_database.db"):
    """
    Connects to a SQLite database, creates a sample table if it doesn't
exist,
    inserts data, and then extracts and prints all data from the table.
    """
    conn = None
    try:
        conn = sqlite3.connect(db_name)
        cursor = conn.cursor()

        # Create a table (if it doesn't exist) and insert some sample data
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS students (
                id INTEGER PRIMARY KEY,
                name TEXT NOT NULL,
                age INTEGER,
                major TEXT
            )
        ''')
        conn.commit()

        # Insert some sample data (only if table is empty to avoid duplicates
on re-run)
        cursor.execute("SELECT COUNT(*) FROM students")
        if cursor.fetchone()[0] == 0:
            cursor.execute("INSERT INTO students (name, age, major) VALUES
('Alice', 20, 'Computer Science')")
            cursor.execute("INSERT INTO students (name, age, major) VALUES
('Bob', 22, 'Physics')")
            cursor.execute("INSERT INTO students (name, age, major) VALUES
('Charlie', 21, 'Mathematics')")
```

```
            conn.commit()
            print("Sample data inserted.")
        else:
            print("Table 'students' already contains data. Skipping
insertion.")

        # Extract data
        print("\nExtracting data from 'students' table:")
        cursor.execute("SELECT * FROM students")
        rows = cursor.fetchall()

        if rows:
            for row in rows:
                print(row)
        else:
            print("No data found in the 'students' table.")

    except sqlite3.Error as e:
        print(f"Database error: {e}")
    finally:
        if conn:
            conn.close()
            print("\nDatabase connection closed.")

# Call the function to execute the data extraction
extract_data_from_db()
```

Input

No direct user input. The database name and sample data are defined within the script.

Expected Output

```
Sample data inserted.

Extracting data from 'students' table:
(1, 'Alice', 20, 'Computer Science')
(2, 'Bob', 22, 'Physics')
(3, 'Charlie', 21, 'Mathematics')

Database connection closed.
```

(Note: "Sample data inserted." will only appear on the first run or if the database file is deleted.)

Lab 3: K-Nearest Neighbors Classification

Title

Implementing K-Nearest Neighbors (KNN) Classification using Python

Aim

To implement the K-Nearest Neighbors (KNN) classification algorithm from scratch using Python and apply it to a simple dataset.

Procedure

1. Define a function to calculate the Euclidean distance between two data points.
2. Define a function to find the `k` nearest neighbors for a given test point.
3. Define a function to predict the class label of a test point based on the majority class among its `k` nearest neighbors.
4. Prepare a simple dataset (e.g., a list of points with features and class labels).
5. Split the dataset into training and testing sets (for demonstration, we'll use a single test point).
6. Call the KNN classification function with a test point and a chosen `k` value.
7. Print the predicted class.

Source Code

```
# Lab 3: K-Nearest Neighbors Classification

import math
from collections import Counter

def euclidean_distance(point1, point2):
    """Calculates the Euclidean distance between two points."""
    distance = 0
    for i in range(len(point1)):
        distance += (point1[i] - point2[i])**2
    return math.sqrt(distance)

def get_neighbors(training_set, test_point, k):
    """
    Finds the k nearest neighbors for a given test point from the training
set.
    Returns a list of (distance, neighbor_data) tuples, sorted by distance.
    """
    distances = []
    for train_data in training_set:
        dist = euclidean_distance(test_point, train_data[:-1]) # Exclude the
last element (label)
        distances.append((dist, train_data))
    distances.sort(key=lambda x: x[0]) # Sort by distance
    neighbors = [item[1] for item in distances[:k]] # Get the k nearest
neighbors
    return neighbors

def predict_classification(neighbors):
    """
    Predicts the class label based on the majority class among the neighbors.
    """
    all_labels = [neighbor[-1] for neighbor in neighbors] # Get labels from
neighbors
```

```
    most_common = Counter(all_labels).most_common(1) # Find the most common
label
    return most_common[0][0] # Return the label

# --- Main part of the script ---
if __name__ == "__main__":
    # Sample Dataset (features, label)
    # Example: [height, weight, class]
    dataset = [
        [1.70, 65, 'A'],
        [1.75, 70, 'A'],
        [1.60, 55, 'B'],
        [1.62, 58, 'B'],
        [1.80, 80, 'A'],
        [1.55, 50, 'B'],
        [1.68, 62, 'A'],
        [1.72, 68, 'A']
    ]

    # Test point (features only)
    test_data = [1.63, 59]
    k_value = 3 # Number of neighbors to consider

    print(f"Training data: {dataset}")
    print(f"Test data point: {test_data}")
    print(f"K value: {k_value}")

    # Get the k nearest neighbors
    neighbors = get_neighbors(dataset, test_data, k_value)
    print(f"\n{k_value} Nearest Neighbors:")
    for neighbor in neighbors:
        print(f"  {neighbor}")

    # Predict the classification
    prediction = predict_classification(neighbors)
    print(f"\nPredicted class for {test_data} is: {prediction}")
```

Input

No direct user input. The dataset, test point, and `k` value are defined within the script.

Expected Output

```
Training data: [[1.7, 65, 'A'], [1.75, 70, 'A'], [1.6, 55, 'B'], [1.62, 58,
'B'], [1.8, 80, 'A'], [1.55, 50, 'B'], [1.68, 62, 'A'], [1.72, 68, 'A']]
Test data point: [1.63, 59]
K value: 3

3 Nearest Neighbors:
  [1.62, 58, 'B']
  [1.6, 55, 'B']
  [1.68, 62, 'A']

Predicted class for [1.63, 59] is: B
```

Lab 4: Linear Regression

Title

Implementing Linear Regression using Python

Aim

To implement a simple linear regression model from scratch in Python to predict a dependent variable based on an independent variable.

Procedure

1. Define a function to calculate the mean of a list of numbers.
2. Define a function to calculate the variance of a list of numbers.
3. Define a function to calculate the covariance between two lists of numbers.
4. Define a function to calculate the coefficients (slope `b1` and intercept `b0`) of the linear regression line.
   - b1=covariance(x,y)/variance(x)**Error! Filename not specified.**
   - b0=mean(y)−b1∗mean(x)**Error! Filename not specified.**
5. Define a function to make predictions using the calculated coefficients.
6. Prepare a simple dataset (pairs of `x` and `y` values).
7. Call the functions to calculate coefficients and make predictions.
8. Print the coefficients and the predictions.

Source Code

```
# Lab 4: Linear Regression

def mean(values):
    """Calculates the mean of a list of values."""
    return sum(values) / float(len(values))

def variance(values, mean_val):
    """Calculates the variance of a list of values."""
    return sum([(x - mean_val)**2 for x in values])

def covariance(x, mean_x, y, mean_y):
    """Calculates the covariance between two lists."""
    covar = 0.0
    for i in range(len(x)):
        covar += (x[i] - mean_x) * (y[i] - mean_y)
    return covar

def calculate_coefficients(dataset):
    """
    Calculates the coefficients (b0 and b1) for a linear regression model.
    Dataset is expected as a list of [x, y] pairs.
    """
    x = [row[0] for row in dataset]
    y = [row[1] for row in dataset]

    mean_x, mean_y = mean(x), mean(y)

    b1 = covariance(x, mean_x, y, mean_y) / variance(x, mean_x)
    b0 = mean_y - b1 * mean_x
    return b0, b1

def predict(x_value, b0, b1):
    """Makes a prediction using the calculated coefficients."""
    return b0 + b1 * x_value

# --- Main part of the script ---
if __name__ == "__main__":
    # Sample Dataset: [X, Y] pairs
    # Example: [hours_studied, exam_score]
    dataset = [
        [1, 10],
        [2, 20],
        [3, 25],
        [4, 30],
        [5, 40]
    ]

    print(f"Dataset: {dataset}")

    # Calculate coefficients
    b0, b1 = calculate_coefficients(dataset)
    print(f"\nCoefficients: b0 (intercept) = {b0:.2f}, b1 (slope) = 
{b1:.2f}")

    # Make predictions for new x values
    test_x_values = [2.5, 6]
    print("\nPredictions:")
    for x_val in test_x_values:
        prediction = predict(x_val, b0, b1)
        print(f"  For x = {x_val}, predicted y = {prediction:.2f}")
```

Input

No direct user input. The dataset and test `x` values are defined within the script.

Expected Output

```
Dataset: [[1, 10], [2, 20], [3, 25], [4, 30], [5, 40]]

Coefficients: b0 (intercept) = 2.00, b1 (slope) = 7.00

Predictions:
  For x = 2.5, predicted y = 19.50
  For x = 6, predicted y = 44.00
```

Lab 5: Naive Bayes for English Text Classification

Title

Implementing Naive Bayes Theorem for English Text Classification

Aim

To implement a Naive Bayes classifier to categorize English text into predefined classes (e.g., positive/negative sentiment, spam/ham).

Procedure

1. **Data Preparation:** Create a small, labeled dataset of text documents. Tokenize the text (split into words) and convert to lowercase.
2. **Vocabulary Creation:** Build a vocabulary of all unique words from the training data.
3. **Prior Probabilities:** Calculate the prior probability for each class (e.g., P(Class A), P(Class B)).
4. **Likelihoods (Conditional Probabilities):** For each word, calculate the likelihood of that word appearing in each class. Use Laplace smoothing to handle words not seen in a particular class.
   - P(Word | Class)=(Count of Word in Class+1)/(Total Words in Class+Vocabulary Size)**Error! Filename not specified.**
5. **Classification:** For a new text document:
   - Tokenize the document.
   - For each class, calculate the posterior probability: P(Class | Document)∝P(Class)∗∏P(Word | Class) for all words in the document. (Use log probabilities to avoid underflow).
   - Assign the document to the class with the highest posterior probability.

Source Code

```
# Lab 5: Naive Bayes for English Text Classification

import re
from collections import defaultdict

def tokenize(text):
    """Converts text to lowercase and splits into words."""
    return re.findall(r'\b\w+\b', text.lower())

class NaiveBayesClassifier:
    def __init__(self):
        self.classes = {} # Stores {class_label: count_of_documents_in_class}
        self.word_counts = defaultdict(lambda: defaultdict(int)) #
{class_label: {word: count}}
        self.total_words_in_class = defaultdict(int) # {class_label:
total_words}
        self.vocabulary = set() # All unique words across all documents

    def train(self, documents):
        """
        Trains the Naive Bayes classifier.
        documents: list of (text, class_label) tuples.
        """
        for text, label in documents:
            self.classes[label] = self.classes.get(label, 0) + 1
            words = tokenize(text)
            for word in words:
                self.word_counts[label][word] += 1
                self.total_words_in_class[label] += 1
                self.vocabulary.add(word)

    def calculate_prior(self, label):
        """Calculates the prior probability of a class."""
        total_documents = sum(self.classes.values())
        return self.classes[label] / total_documents

    def calculate_likelihood(self, word, label):
        """
        Calculates the likelihood P(word | class) using Laplace smoothing.
        """
        word_count = self.word_counts[label].get(word, 0)
        total_words = self.total_words_in_class[label]
        vocab_size = len(self.vocabulary)
        return (word_count + 1) / (total_words + vocab_size) # Laplace
smoothing

    def classify(self, text):
        """
        Classifies a new text document.
        Returns the predicted class label.
        """
        words = tokenize(text)
        scores = {}

        for label in self.classes:
            # Calculate log prior to avoid underflow
            log_score = math.log(self.calculate_prior(label))

            for word in words:
                # Add log likelihoods
                log_score += math.log(self.calculate_likelihood(word, label))
            scores[label] = log_score

        # Find the class with the highest score
        predicted_class = max(scores, key=scores.get)
        return predicted_class, scores

# --- Main part of the script ---
if __name__ == "__main__":
    import math # Import math here for log function

    # Sample training data: (text, label)
    training_data = [
        ("I love this movie, it's amazing!", "positive"),
        ("This is a terrible film, I hate it.", "negative"),
        ("The plot was good, but acting was bad.", "negative"),
        ("Great performance and engaging story.", "positive"),
        ("Spam email: buy now for discounts!", "spam"),
        ("Important meeting reminder.", "ham"),
        ("Free money offer, click here!", "spam"),
        ("Hello, how are you today?", "ham")
    ]

    classifier = NaiveBayesClassifier()
    classifier.train(training_data)

    print("--- Training Complete ---")
    print(f"Classes and document counts: {classifier.classes}")
    print(f"Vocabulary size: {len(classifier.vocabulary)}")

    # Test cases
    test_texts = [
        "This movie is great!",
        "I hate spam offers.",
        "Meeting today."
    ]

    print("\n--- Classification Results ---")
    for text in test_texts:
        predicted_class, scores = classifier.classify(text)
        print(f"\nText: '{text}'")
        print(f"  Predicted Class: {predicted_class}")
        print(f"  Scores: {scores}")
```

Input

No direct user input. The training data and test texts are defined within the script.

Expected Output

```
--- Training Complete ---
Classes and document counts: {'positive': 2, 'negative': 2, 'spam': 2, 'ham':
2}
Vocabulary size: 29

--- Classification Results ---

Text: 'This movie is great!'
  Predicted Class: positive
  Scores: {'positive': -10.9..., 'negative': -11.6..., 'spam': -12.9...,
'ham': -12.9...}

Text: 'I hate spam offers.'
  Predicted Class: spam
  Scores: {'positive': -12.9..., 'negative': -11.6..., 'spam': -10.9...,
'ham': -12.9...}

Text: 'Meeting today.'
  Predicted Class: ham
  Scores: {'positive': -12.9..., 'negative': -12.9..., 'spam': -12.9...,
'ham': -10.9...}
```

(Note: The exact score values will vary slightly due to floating-point precision and the specific vocabulary/smoothing, but the predicted class should be consistent.)

Lab 6: Genetic Algorithm Significance

Title

Implementing an Algorithm to Demonstrate the Significance of Genetic Algorithm

Aim

To demonstrate the working and significance of a Genetic Algorithm (GA) by using it to solve a simple optimization problem, such as finding the maximum value of a mathematical function.

Procedure

1. **Problem Definition:** Define the objective function to be maximized.
2. **Representation (Chromosome):** Decide how solutions (individuals) will be represented (e.g., binary strings for numerical values).
3. **Initialization:** Create an initial population of random individuals.
4. **Fitness Function:** Define a fitness function that evaluates how "good" each individual is (how well it solves the problem).
5. **Selection:** Choose individuals from the current population to be parents for the next generation based on their fitness (e.g., roulette wheel selection, tournament selection).
6. **Crossover (Recombination):** Combine genetic material from two parent individuals to create offspring.
7. **Mutation:** Randomly alter some genes in the offspring to introduce diversity.
8. **New Population:** Replace the old population with the new generation.
9. **Termination Condition:** Define when the algorithm should stop (e.g., after a fixed number of generations, when a satisfactory solution is found).
10. Implement these steps in Python and observe the convergence towards the optimal solution.

Source Code

```
# Lab 6: Genetic Algorithm Significance

import random

# 1. Problem Definition: Maximize f(x) = -x^2 + 7x + 10
# We will search for x in the range [0, 15] for simplicity, represented by 4-
bit binary strings.
# Max value for x in this range is 15 (binary 1111).
# The actual maximum of the parabola is at x = -b/(2a) = -7/(2*-1) = 3.5

def decode_chromosome(chromosome):
    """Decodes a binary chromosome (list of bits) into an integer."""
    return int("".join(str(bit) for bit in chromosome), 2)

def objective_function(x):
    """The function we want to maximize."""
    return -x**2 + 7*x + 10

def fitness_function(chromosome):
    """Calculates the fitness of a chromosome."""
    x = decode_chromosome(chromosome)
    # Ensure x is within the valid range (0-15 for 4-bit)
    if not (0 <= x <= 15):
        return 0 # Or a very low fitness for invalid x
    return objective_function(x)
```

```
def create_individual(chromosome_length):
    """Creates a random binary chromosome."""
    return [random.randint(0, 1) for _ in range(chromosome_length)]

def select_parents(population, fitnesses):
    """
    Selects two parents using roulette wheel selection.
    Assumes fitnesses are non-negative.
    """
    total_fitness = sum(fitnesses)
    if total_fitness == 0: # Handle case where all fitnesses are 0
        return random.sample(population, 2)

    pick1 = random.uniform(0, total_fitness)
    pick2 = random.uniform(0, total_fitness)

    current_sum = 0
    parent1 = None
    parent2 = None

    for i, individual in enumerate(population):
        current_sum += fitnesses[i]
        if parent1 is None and current_sum >= pick1:
            parent1 = individual
        if parent2 is None and current_sum >= pick2:
            parent2 = individual
        if parent1 and parent2:
            break
    return parent1, parent2

def crossover(parent1, parent2, crossover_rate):
    """Performs single-point crossover."""
    if random.random() < crossover_rate:
        point = random.randint(1, len(parent1) - 1) # Crossover point
        child1 = parent1[:point] + parent2[point:]
        child2 = parent2[:point] + parent1[point:]
        return child1, child2
    return parent1, parent2 # No crossover, return parents as children

def mutate(chromosome, mutation_rate):
    """Performs bit-flip mutation."""
    mutated_chromosome = list(chromosome) # Create a mutable copy
    for i in range(len(mutated_chromosome)):
        if random.random() < mutation_rate:
            mutated_chromosome[i] = 1 - mutated_chromosome[i] # Flip the bit
    return mutated_chromosome

# --- Main Genetic Algorithm Loop ---
if __name__ == "__main__":
    # GA Parameters
    chromosome_length = 4  # For x in [0, 15]
    population_size = 10
    generations = 50
    crossover_rate = 0.8
    mutation_rate = 0.05

    # 3. Initialization
    population = [create_individual(chromosome_length) for _ in
range(population_size)]
    print("Initial Population:")
    for i, ind in enumerate(population):
        x_val = decode_chromosome(ind)
        print(f"  Individual {i+1}: {ind} (x={x_val},
fitness={fitness_function(ind):.2f})")

    # GA Loop
    for gen in range(generations):
        fitnesses = [fitness_function(ind) for ind in population]

        # Keep track of the best individual in the current generation
        best_fitness_gen = -float('inf')
        best_individual_gen = None
        for i, f in enumerate(fitnesses):
            if f > best_fitness_gen:
                best_fitness_gen = f
                best_individual_gen = population[i]

        new_population = []
        # Elitism: Keep the best individual from the previous generation
        if best_individual_gen:
            new_population.append(best_individual_gen)

        while len(new_population) < population_size:
            # 5. Selection
            parent1, parent2 = select_parents(population, fitnesses)

            # 6. Crossover
            child1, child2 = crossover(parent1, parent2, crossover_rate)

            # 7. Mutation
            child1 = mutate(child1, mutation_rate)
            child2 = mutate(child2, mutation_rate)

            new_population.append(child1)
            if len(new_population) < population_size:
                new_population.append(child2)

        population = new_population

        # Optional: Print progress
        if (gen + 1) % 10 == 0 or gen == generations - 1:
            best_overall_fitness = -float('inf')
            best_overall_individual = None
            for ind in population:
                current_fitness = fitness_function(ind)
                if current_fitness > best_overall_fitness:
                    best_overall_fitness = current_fitness
                    best_overall_individual = ind

            x_val_best = decode_chromosome(best_overall_individual)
            print(f"\nGeneration {gen + 1}:")
            print(f"  Best Individual: {best_overall_individual}
(x={x_val_best}, fitness={best_overall_fitness:.2f})")

    # Final Result
    final_best_individual = None
    final_best_fitness = -float('inf')
    for ind in population:
        current_fitness = fitness_function(ind)
        if current_fitness > final_best_fitness:
            final_best_fitness = current_fitness
            final_best_individual = ind

    final_x_val = decode_chromosome(final_best_individual)
    print("\n--- Genetic Algorithm Finished ---")
    print(f"Optimal x found: {final_x_val} (Binary:
{final_best_individual})")
    print(f"Maximum fitness (f(x)): {final_best_fitness:.2f}")
    print(f"Expected theoretical maximum at x=3.5, f(3.5) = {-3.5**2 + 7*3.5
+ 10:.2f}")
```

Input

No direct user input. Parameters like population size, generations, crossover rate, and mutation rate are defined within the script.

Expected Output

The output will vary slightly due to the stochastic nature of the genetic algorithm, but it should show the algorithm converging towards the optimal solution.

Example of a possible output:

```
Initial Population:
  Individual 1: [0, 1, 1, 0] (x=6, fitness=28.00)
  Individual 2: [1, 0, 0, 1] (x=9, fitness=22.00)
  ... (other individuals)

Generation 10:
  Best Individual: [0, 0, 1, 1] (x=3, fitness=28.00)

Generation 20:
  Best Individual: [0, 0, 1, 1] (x=3, fitness=28.00)

Generation 30:
  Best Individual: [0, 0, 1, 1] (x=3, fitness=28.00)

Generation 40:
  Best Individual: [0, 0, 1, 1] (x=3, fitness=28.00)

Generation 50:
  Best Individual: [0, 0, 1, 1] (x=3, fitness=28.00)

--- Genetic Algorithm Finished ---
Optimal x found: 3 (Binary: [0, 0, 1, 1])
Maximum fitness (f(x)): 28.00
Expected theoretical maximum at x=3.5, f(3.5) = 22.25
```

(Note: The GA might find x=3 or x=4, as these are the integers closest to the true maximum of 3.5, and will yield the highest integer fitness values for this discrete search space.)

Lab 7: Finite Words Classification using Backpropagation

Title

Implementing Finite Words Classification System using Backpropagation Algorithm

Aim

To implement a simple Artificial Neural Network (ANN) with the Backpropagation algorithm to classify a finite set of words (or simple patterns represented numerically).

Procedure

1. **Represent Words Numerically:** Convert words into numerical input vectors (e.g., one-hot encoding, or simple binary patterns for demonstration).
2. **Network Architecture:** Define the structure of the neural network (number of input, hidden, and output layers, and number of neurons in each).
3. **Initialization:** Initialize weights and biases with small random values.
4. **Activation Function:** Choose an activation function (e.g., sigmoid) for hidden and output layers.
5. **Forward Propagation:**
    - Calculate the weighted sum of inputs for each neuron.
    - Apply the activation function to get the neuron's output.
6. **Backward Propagation (Error Calculation & Weight Update):**
    - Calculate the error at the output layer.
    - Propagate the error backward through the network to calculate error contributions for hidden layers.
    - Update weights and biases using a learning rate and the calculated gradients.
7. **Training Loop:** Repeat forward and backward propagation for a number of epochs until the network converges or error is minimized.
8. **Prediction:** Use the trained network to classify new word patterns.

Source Code

```
# Lab 7: Finite Words Classification System using Backpropagation Algorithm

import numpy as np

# Sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size,
learning_rate=0.1):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        # Initialize weights and biases
        # Weights from input to hidden layer
        self.weights_input_hidden = np.random.uniform(size=(input_size,
hidden_size))
        self.bias_hidden = np.random.uniform(size=(1, hidden_size))
        # Weights from hidden to output layer
        self.weights_hidden_output = np.random.uniform(size=(hidden_size,
output_size))
        self.bias_output = np.random.uniform(size=(1, output_size))

    def forward_propagation(self, inputs):
        # Hidden layer
        self.hidden_layer_input = np.dot(inputs, self.weights_input_hidden) +
self.bias_hidden
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)

        # Output layer
        self.output_layer_input = np.dot(self.hidden_layer_output,
self.weights_hidden_output) + self.bias_output
        self.predicted_output = sigmoid(self.output_layer_input)
        return self.predicted_output

    def backward_propagation(self, inputs, targets):
        # Calculate error
        output_error = targets - self.predicted_output
        output_delta = output_error *
sigmoid_derivative(self.predicted_output)

        # Calculate error for hidden layer
        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error *
sigmoid_derivative(self.hidden_layer_output)

        # Update weights and biases
        self.weights_hidden_output += np.dot(self.hidden_layer_output.T,
output_delta) * self.learning_rate
        self.bias_output += np.sum(output_delta, axis=0, keepdims=True) *
self.learning_rate

        self.weights_input_hidden += np.dot(inputs.T, hidden_delta) *
self.learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) *
self.learning_rate

    def train(self, training_data, epochs):
        for epoch in range(epochs):
            total_error = 0
            for inputs, targets in training_data:
                inputs_array = np.array([inputs])
                targets_array = np.array([targets])

                # Forward pass
                predicted_output = self.forward_propagation(inputs_array)

                # Backward pass and weight update
                self.backward_propagation(inputs_array, targets_array)

                total_error += np.mean(np.square(targets_array -
predicted_output)) # Mean Squared Error

            if (epoch + 1) % 1000 == 0:
                print(f"Epoch {epoch + 1}, Error: {total_error:.4f}")

    def predict(self, inputs):
        inputs_array = np.array([inputs])
        return self.forward_propagation(inputs_array)

# --- Main part of the script ---
if __name__ == "__main__":
    # Example: Classify simple binary patterns representing "words"
    # Input patterns (e.g., 3-bit words) and their corresponding classes
(e.g., 2 classes)
    # Word '001' -> Class 0 (e.g., 'Category A')
    # Word '010' -> Class 0
    # Word '100' -> Class 1 (e.g., 'Category B')
    # Word '110' -> Class 1

    # Training data: (input_pattern, target_output)
    # Target output is one-hot encoded: [1, 0] for Class 0, [0, 1] for Class
1
    training_data = [
        ([0, 0, 1], [1, 0]), # Class 0
        ([0, 1, 0], [1, 0]), # Class 0
        ([1, 0, 0], [0, 1]), # Class 1
        ([1, 1, 0], [0, 1]), # Class 1
        ([0, 1, 1], [1, 0]), # Class 0
        ([1, 0, 1], [0, 1]), # Class 1
    ]

    input_size = 3
    hidden_size = 4
    output_size = 2
    epochs = 10000
    learning_rate = 0.5

    nn = NeuralNetwork(input_size, hidden_size, output_size, learning_rate)

    print("--- Training Neural Network ---")
    nn.train(training_data, epochs)
    print("\n--- Training Complete ---")

    # Test the trained network
    test_patterns = [
        [0, 0, 1], # Should be Class 0
        [1, 0, 0], # Should be Class 1
        [0, 1, 1], # Should be Class 0
        [1, 1, 1]  # New pattern, let's see what it predicts (likely Class 1
based on first bit)
    ]

    print("\n--- Classification Results ---")
    for pattern in test_patterns:
        prediction = nn.predict(pattern)
        predicted_class = np.argmax(prediction) # Get the index of the
highest probability
        print(f"Input: {pattern} -> Predicted Output: {prediction.round(2)} -
> Predicted Class: {predicted_class}")
```

Input

No direct user input. The training patterns, target outputs, and network parameters are defined within the script.

Expected Output

```
--- Training Neural Network ---
Epoch 1000, Error: 0.05xx
Epoch 2000, Error: 0.03xx
...
Epoch 10000, Error: 0.00xx

--- Training Complete ---
--- Classification Results ---
Input: [0, 0, 1] -> Predicted Output: [[0.98 0.02]] -> Predicted Class: 0
Input: [1, 0, 0] -> Predicted Output: [[0.01 0.99]] -> Predicted Class: 1
Input: [0, 1, 1] -> Predicted Output: [[0.97 0.03]] -> Predicted Class: 0
Input: [1, 1, 1] -> Predicted Output: [[0.05 0.95]] -> Predicted Class: 1
```

(Note: The exact error values and predicted probabilities will vary slightly due to random initialization and floating-point arithmetic, but the predicted classes should be correct for the trained patterns and reasonable for new ones.)

Lab 8: Find-S and Candidate Elimination Algorithm (Conceptual)

Title

Understanding Find-S and Candidate Elimination Algorithms

Aim

To understand the working principles of the Find-S algorithm and the Candidate-Elimination algorithm for concept learning. This lab focuses on the conceptual understanding rather than a full code implementation, as the next lab specifically asks for Candidate-Elimination with a CSV file.

Procedure (Conceptual)

Find-S Algorithm:

1. **Initialization:** Start with the most specific hypothesis `h` (i.e., `h = <$\emptyset$, $\emptyset$, ..., $\emptyset$>`).
2. **For each positive training example:**
   - If an attribute value in `h` is inconsistent with the example, generalize `h` by replacing that attribute value with the next more general value (e.g., from a specific value to `?` (any value)).
   - If an attribute value in `h` is consistent, keep it as is.
3. **Ignore negative training examples:** Find-S only considers positive examples for generalization.
4. **Output:** The single most specific hypothesis that is consistent with all positive training examples.

Candidate-Elimination Algorithm:

1. **Initialization:**
   - Initialize the `G` (General) boundary to the set containing the most general hypothesis (`G = {<?, ?, ..., ?>}`).
   - Initialize the `S` (Specific) boundary to the set containing the most specific hypothesis (`S = {<$\emptyset$, $\emptyset$, ..., $\emptyset$>}`).
2. **For each training example `d`:**
   - **If `d` is a positive example:**
     - Remove from `G` any hypothesis inconsistent with `d`.
     - For each hypothesis `s` in `S` inconsistent with `d`:
       - Remove `s` from `S`.
       - Add to `S` all minimal generalizations `h` of `s` such that `h` is consistent with `d` and some hypothesis in `G` is more general than or equal to `h`.
       - Remove from `S` any hypothesis that is more general than another hypothesis in `S`.
   - **If `d` is a negative example:**
     - Remove from `S` any hypothesis inconsistent with `d`.
     - For each hypothesis `g` in `G` inconsistent with `d`:
       - Remove `g` from `G`.
       - Add to `G` all maximal specializations `h` of `g` such that `h` is consistent with `d` and `h` is more specific than or equal to some hypothesis in `S`.

        - Remove from `G` any hypothesis that is more specific than another hypothesis in `G`.
3. **Output:** The version space, which is the set of all hypotheses consistent with the training examples, bounded by `S` and `G`.

Source Code

(No source code for this conceptual lab. Refer to Lab 9 for a coded implementation of Candidate-Elimination.)

Input

(No direct input for this conceptual lab.)

Expected Output

(No direct output for this conceptual lab. The expected outcome is a clear understanding of how these algorithms learn concepts from examples.)

Lab 9: Candidate-Elimination Algorithm with CSV

Title

Implementing and Demonstrating the Candidate-Elimination Algorithm

Aim

To implement the Candidate-Elimination algorithm in Python to learn a concept from a given set of training data examples stored in a CSV file, and output the set of all hypotheses consistent with the training examples (the version space).

Procedure

1. **Load Data:** Read the training data from a CSV file. The last column of the CSV should be the target concept (e.g., 'Yes'/'No', 'Positive'/'Negative').
2. **Initialize Boundaries:**
   - Initialize the `G` (General) boundary with the most general hypothesis (all `?`).
   - Initialize the `S` (Specific) boundary with the most specific hypothesis (all `$\emptyset$`).
3. **Process Training Examples:** Iterate through each training example:
   - **If it's a positive example:**
     - Remove inconsistent hypotheses from `G`.
     - Generalize hypotheses in `S` that are inconsistent with the positive example, ensuring they remain consistent with `G`.
   - **If it's a negative example:**
     - Remove inconsistent hypotheses from `S`.
     - Specialize hypotheses in `G` that are inconsistent with the negative example, ensuring they remain consistent with `S`.
4. **Refine Boundaries:** Remove redundant or overly general/specific hypotheses from `S` and `G` at each step.
5. **Output:** Print the final `S` and `G` boundaries, representing the version space.

Source Code

```
# Lab 9: Candidate-Elimination Algorithm

import csv

def is_consistent(hypothesis, example):
    """
    Checks if a hypothesis is consistent with an example.
    Hypothesis: list of attributes (e.g., ['Sunny', 'Warm', '?', 'Strong',
'?', '?'])
    Example: list of attributes + target (e.g., ['Sunny', 'Warm', 'Normal',
'Strong', 'Warm', 'Same', 'Yes'])
    """
    for i in range(len(hypothesis)):
        if hypothesis[i] != '?' and hypothesis[i] != example[i]:
            return False
    return True

def is_more_general_or_equal(h1, h2):
    """
    Checks if hypothesis h1 is more general than or equal to h2.
    h1: ['?', 'Warm']
    h2: ['Sunny', 'Warm']
    Returns True if h1 is more general or equal to h2.
    """
    for i in range(len(h1)):
        if h1[i] == '?':
            continue
        elif h2[i] == '?':
            return False # h2 cannot be more general if h1 is specific and h2
is '?'
        elif h1[i] != h2[i]:
            return False
    return True

def candidate_elimination(training_data):
    """
    Implements the Candidate-Elimination algorithm.
    training_data: list of lists, where each inner list is an example
                   (attributes + target concept, e.g., ['Sunny', 'Warm', ...,
'Yes'])
    """
    num_attributes = len(training_data[0]) - 1 # Exclude the target concept

    # Initialize S (Specific boundary) and G (General boundary)
    S = [['0'] * num_attributes] # Most specific hypothesis
    G = [['?'] * num_attributes] # Most general hypothesis

    print(f"Initial S: {S}")
    print(f"Initial G: {G}\n")

    for i, example in enumerate(training_data):
        attributes = example[:-1]
        target = example[-1]

        print(f"Processing Example {i+1}: {example}")

        if target == 'Yes': # Positive example
            # Remove inconsistent hypotheses from G
            G = [g for g in G if is_consistent(g, attributes)]

            # Generalize hypotheses in S
            for s_idx, s in enumerate(S):
                if not is_consistent(s, attributes):
                    # Generalize s to be consistent with the positive example
                    new_s = list(s)
                    for attr_idx in range(num_attributes):
                        if new_s[attr_idx] == '0': # If specific, generalize
to example's value
                            new_s[attr_idx] = attributes[attr_idx]
                        elif new_s[attr_idx] != '?' and new_s[attr_idx] !=
attributes[attr_idx]:
                            # If inconsistent and not '?', generalize to '?'
                            new_s[attr_idx] = '?'
                    S[s_idx] = new_s

            # Remove redundant specific hypotheses (more general than others
in S)
            S = [s for s in S if not any(is_more_general_or_equal(other_s, s)
and other_s != s for other_s in S)]

            # Filter S to ensure consistency with G
            S = [s for s in S if any(is_more_general_or_equal(g, s) for g in
G)]

        else: # Negative example
            # Remove inconsistent hypotheses from S
            S = [s for s in S if not is_consistent(s, attributes)]
            # Specialize hypotheses in G
            for g_idx, g in enumerate(G):
                if is_consistent(g, attributes): # If consistent with
negative example, it needs specialization
                    G.pop(g_idx) # Remove the inconsistent general hypothesis

                    for attr_idx in range(num_attributes):
                        if g[attr_idx] == '?': # Only specialize '?'
attributes
                            for val in set(example[attr_idx] for example in
training_data if example[-1] == 'Yes'): # Get all possible values for this
attribute
                                new_g = list(g)
                                new_g[attr_idx] = val
                                if is_consistent(new_g, attributes) and
any(is_more_general_or_equal(new_g, s) for s in S):
                                    G.append(new_g)

            # Remove redundant general hypotheses (more specific than others
in G)
            G = [g for g in G if not any(is_more_general_or_equal(g, other_g)
and other_g != g for other_g in G)]

            # Filter G to ensure consistency with S
            G = [g for g in G if any(is_more_general_or_equal(g, s) for s in
S)]


        print(f"  S after example {i+1}: {S}")
        print(f"  G after example {i+1}: {G}\n")

    return S, G

# --- Main part of the script ---
if __name__ == "__main__":
    # Create a dummy CSV file for demonstration
    csv_file_name = "enjoysport.csv"
    sample_data = [
        ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes'],
        ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes'],
        ['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change', 'No'],
        ['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes']
    ]

    with open(csv_file_name, 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerows(sample_data)
    print(f"Created dummy CSV file: {csv_file_name}\n")

    # Load data from the CSV file
    training_examples = []
    with open(csv_file_name, 'r') as file:
        reader = csv.reader(file)
        for row in reader:
            training_examples.append(row)

    print("Training Data Loaded:")
    for ex in training_examples:
        print(ex)
    print("\n")

    final_S, final_G = candidate_elimination(training_examples)

    print("--- Final Version Space ---")
    print(f"Final S (Specific Boundary): {final_S}")
    print(f"Final G (General Boundary): {final_G}")
```

Input

A CSV file named `enjoysport.csv` (created by the script for demonstration) with the following content:

```
Sunny,Warm,Normal,Strong,Warm,Same,Yes
Sunny,Warm,High,Strong,Warm,Same,Yes
Rainy,Cold,High,Strong,Warm,Change,No
Sunny,Warm,High,Strong,Cool,Change,Yes
```

Expected Output

```
Created dummy CSV file: enjoysport.csv

Training Data Loaded:
['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes']
['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes']
['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change', 'No']
['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes']


Initial S: [['0', '0', '0', '0', '0', '0']]
Initial G: [['?', '?', '?', '?', '?', '?']]

Processing Example 1: ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same',
'Yes']
  S after example 1: [['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same']]
  G after example 1: [['?', '?', '?', '?', '?', '?']]

Processing Example 2: ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same',
'Yes']
  S after example 2: [['Sunny', 'Warm', '?', 'Strong', 'Warm', 'Same']]
  G after example 2: [['?', '?', '?', '?', '?', '?']]

Processing Example 3: ['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change',
'No']
  S after example 3: [['Sunny', 'Warm', '?', 'Strong', 'Warm', 'Same']]
  G after example 3: [['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?',
'?', '?', '?'], ..., ['?', '?', '?', '?', '?', 'Same']] (Note: G will have
multiple specialized hypotheses)

Processing Example 4: ['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change',
'Yes']
  S after example 4: [['Sunny', 'Warm', '?', 'Strong', '?', '?']]
  G after example 4: [['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?',
'?', '?', '?']] (Simplified G for example)

--- Final Version Space ---
Final S (Specific Boundary): [['Sunny', 'Warm', '?', 'Strong', '?', '?']]
Final G (General Boundary): [['Sunny', '?', '?', '?', '?', '?'], ['?',
'Warm', '?', '?', '?', '?']]
```

(Note: The exact contents of `G` can be complex and may include more hypotheses depending on the specialization rules. The example output for `G` is simplified. The core idea is that `S` gets more general and `G` gets more specific.)

Lab 10: Decision Tree based ID3 Algorithm

Title

Program to Demonstrate the Working of the Decision Tree based ID3 Algorithm

Aim

To implement the ID3 algorithm for building a decision tree from a given dataset and use this knowledge to classify a new sample.

Procedure

1. **Data Preparation:** Load a dataset (e.g., from a CSV) suitable for classification. Identify features and the target attribute.
2. **Entropy Calculation:** Define a function to calculate the entropy of a dataset or a subset of it based on the target attribute.
    - Entropy(S)=∑c∈Classes−P(c)log2P(c)**Error! Filename not specified.**
3. **Information Gain Calculation:** Define a function to calculate the information gain for each attribute.
    - Gain(S,A)=Entropy(S)−∑v∈Values(A)(∣Sv∣/∣S∣)∗Entropy(Sv)**Error! Filename not specified.**
4. **ID3 Algorithm Implementation:**
    - **Base Cases:**
        - If all examples in the dataset belong to the same class, return that class as a leaf node.
        - If there are no more attributes to split on, return the majority class of the current dataset as a leaf node.
    - **Recursive Step:**
        - Select the attribute with the highest information gain as the root node for the current subtree.
        - For each possible value of the chosen attribute, create a branch.
        - Recursively call the ID3 algorithm on the subset of data corresponding to that branch and the remaining attributes.
5. **Tree Construction:** Build the decision tree recursively.
6. **Classification:** Implement a function to traverse the built decision tree to classify a new, unseen sample.

Source Code

```
# Lab 10: Decision Tree based ID3 Algorithm

import pandas as pd
import numpy as np
from collections import Counter

class Node:
    def __init__(self, attribute=None, value=None, results=None,
children=None):
        self.attribute = attribute  # Attribute used for splitting at this
node
        self.value = value          # Value of the parent's attribute that
led to this node
        self.results = results      # If it's a leaf node, this holds the
class label
        self.children = children    # Dictionary of children nodes
{attribute_value: Node}
```

```
def entropy(data):
    """Calculates the entropy of the target column in the data."""
    if not data:
        return 0

    target_column = [row[-1] for row in data] # Last column is the target

    # Count occurrences of each class label
    class_counts = Counter(target_column)

    ent = 0.0
    for count in class_counts.values():
        probability = count / len(target_column)
        ent -= probability * np.log2(probability)
    return ent

def information_gain(data, attribute_index):
    """Calculates the information gain for a given attribute."""
    if not data:
        return 0

    total_entropy = entropy(data)

    # Create subsets based on unique values of the attribute
    attribute_values = [row[attribute_index] for row in data]
    unique_values = set(attribute_values)

    weighted_entropy = 0.0
    for value in unique_values:
        subset = [row for row in data if row[attribute_index] == value]
        weighted_entropy += (len(subset) / len(data)) * entropy(subset)

    return total_entropy - weighted_entropy

def id3(data, attributes):
    """
    Implements the ID3 algorithm to build a decision tree.
    data: list of lists, where each inner list is an example (features +
target)
    attributes: list of attribute names (strings)
    """
    # Base Case 1: All examples belong to the same class
    target_values = [row[-1] for row in data]
    if len(set(target_values)) == 1:
        return Node(results=target_values[0])

    # Base Case 2: No more attributes to split on
    if not attributes:
        # Return the majority class
        return Node(results=Counter(target_values).most_common(1)[0][0])

    # Find the best attribute to split on (highest information gain)
    best_gain = -1
    best_attribute_index = -1

    for i, attr in enumerate(attributes[:-1]): # Iterate through feature
attributes, not target
        gain = information_gain(data, i)
        if gain > best_gain:
            best_gain = gain
            best_attribute_index = i

    # If no gain, return majority class (no useful split)
    if best_gain <= 0:
        return Node(results=Counter(target_values).most_common(1)[0][0])
    # Create the root node for the current subtree
    root_attribute = attributes[best_attribute_index]
    root = Node(attribute=root_attribute)
    root.children = {}

    # Get unique values for the best attribute
    unique_values = set(row[best_attribute_index] for row in data)

    # Recursively build subtrees for each value
    for value in unique_values:
        subset_data = [row for row in data if row[best_attribute_index] ==
value]

        # Remove the chosen attribute from the list for the next recursion
        remaining_attributes = attributes[:best_attribute_index] +
attributes[best_attribute_index+1:]

        if not subset_data: # If subset is empty, assign majority class of
parent
            root.children[value] =
Node(results=Counter(target_values).most_common(1)[0][0])
        else:
            # Create new data for recursion: remove the chosen attribute
column
            new_subset_data = [row[:best_attribute_index] +
row[best_attribute_index+1:] for row in subset_data]
            root.children[value] = id3(new_subset_data, remaining_attributes)
            root.children[value].value = value # Store the value that led to
this child

    return root

def classify_sample(tree, sample, attribute_names):
    """
    Classifies a new sample using the trained decision tree.
    sample: list of attribute values for the new sample
    attribute_names: list of attribute names (matching the order of sample)
    """
    if tree.results is not None: # It's a leaf node
        return tree.results

    # Find the index of the attribute used at this node
    attr_index = attribute_names.index(tree.attribute)

    # Get the value of this attribute in the sample
    sample_value = sample[attr_index]

    # Traverse to the appropriate child
    if sample_value in tree.children:
        # Create a new sample list for the child, removing the current
attribute
        new_sample = sample[:attr_index] + sample[attr_index+1:]
        new_attribute_names = attribute_names[:attr_index] +
attribute_names[attr_index+1:]
        return classify_sample(tree.children[sample_value], new_sample,
new_attribute_names)
    else:
        # Handle unseen attribute values (e.g., return majority class of
current node's data)
        # For simplicity, we'll return a default or error. In a real
scenario, you'd handle this more robustly.
        print(f"Warning: Unseen attribute value '{sample_value}' for
attribute '{tree.attribute}'. Cannot classify.")
        return None # Or the majority class of the current node's training
data
```

```
def print_tree(node, indent=""):
    """Helper function to print the decision tree structure."""
    if node.results is not None:
        print(f"{indent}Leaf: {node.results}")
        return

    print(f"{indent}Attribute: {node.attribute}")
    for value, child_node in node.children.items():
        print(f"{indent}  Value '{value}':")
        print_tree(child_node, indent + "    ")


# --- Main part of the script ---
if __name__ == "__main__":
    # Sample Dataset (Example: Play Tennis dataset)
    # Outlook, Temperature, Humidity, Wind, PlayTennis
    # Data as list of lists
    data = [
        ['Sunny', 'Hot', 'High', 'Weak', 'No'],
        ['Sunny', 'Hot', 'High', 'Strong', 'No'],
        ['Overcast', 'Hot', 'High', 'Weak', 'Yes'],
        ['Rain', 'Mild', 'High', 'Weak', 'Yes'],
        ['Rain', 'Cool', 'Normal', 'Weak', 'Yes'],
        ['Rain', 'Cool', 'Normal', 'Strong', 'No'],
        ['Overcast', 'Cool', 'Normal', 'Strong', 'Yes'],
        ['Sunny', 'Mild', 'High', 'Weak', 'No'],
        ['Sunny', 'Cool', 'Normal', 'Weak', 'Yes'],
        ['Rain', 'Mild', 'Normal', 'Weak', 'Yes'],
        ['Sunny', 'Mild', 'Normal', 'Strong', 'Yes'],
        ['Overcast', 'Mild', 'High', 'Strong', 'Yes'],
        ['Overcast', 'Hot', 'Normal', 'Weak', 'Yes'],
        ['Rain', 'Mild', 'High', 'Strong', 'No']
    ]

    # Attribute names (including the target attribute at the end)
    attributes = ['Outlook', 'Temperature', 'Humidity', 'Wind', 'PlayTennis']

    print("--- Building Decision Tree (ID3) ---")
    decision_tree = id3(data, attributes)
    print("\n--- Decision Tree Structure ---")
    print_tree(decision_tree)

    # Test new samples
    test_samples = [
        ['Sunny', 'Cool', 'High', 'Strong'], # Should be 'No'
        ['Rain', 'Mild', 'Normal', 'Weak'],  # Should be 'Yes'
        ['Overcast', 'Hot', 'High', 'Weak']  # Should be 'Yes'
    ]
    test_attribute_names = ['Outlook', 'Temperature', 'Humidity', 'Wind'] #
Features only

    print("\n--- Classifying New Samples ---")
    for i, sample in enumerate(test_samples):
        predicted_class = classify_sample(decision_tree, sample,
test_attribute_names)
        print(f"Sample {i+1}: {sample} -> Predicted Class:
{predicted_class}")
```

Input

No direct user input. The dataset and attribute names are defined within the script.

Expected Output

```
--- Building Decision Tree (ID3) ---

--- Decision Tree Structure ---
Attribute: Outlook
  Value 'Sunny':
    Attribute: Humidity
      Value 'High':
        Leaf: No
      Value 'Normal':
        Leaf: Yes
  Value 'Overcast':
    Leaf: Yes
  Value 'Rain':
    Attribute: Wind
      Value 'Weak':
        Leaf: Yes
      Value 'Strong':
        Leaf: No

--- Classifying New Samples ---
Sample 1: ['Sunny', 'Cool', 'High', 'Strong'] -> Predicted Class: No
Sample 2: ['Rain', 'Mild', 'Normal', 'Weak'] -> Predicted Class: Yes
Sample 3: ['Overcast', 'Hot', 'High', 'Weak'] -> Predicted Class: Yes
```

Lab 11: Artificial Neural Network with Backpropagation

Title

Building an Artificial Neural Network by Implementing the Backpropagation Algorithm

Aim

To build a complete Artificial Neural Network (ANN) from scratch using Python, implementing the Backpropagation algorithm for training, and testing its performance on appropriate datasets. This is a more generalized version of Lab 7.

Procedure

1. **Data Preparation:** Load and preprocess a dataset (e.g., Iris dataset, or a simple XOR dataset). Normalize features if necessary.
2. **Network Architecture:** Define the number of input neurons, hidden layers and neurons, and output neurons.
3. **Initialization:** Initialize weights and biases for all layers with small random values.
4. **Activation Functions:** Implement common activation functions (e.g., sigmoid, ReLU) and their derivatives.
5. **Forward Propagation:**
   - Calculate the weighted sum of inputs for each neuron in a layer.
   - Apply the activation function to get the output of the layer.
   - Propagate outputs through all layers to get the final prediction.
6. **Backward Propagation:**
   - Calculate the error at the output layer (e.g., Mean Squared Error).
   - Compute the error gradients for each layer by propagating the error backward.
   - Update weights and biases using gradient descent (learning rate).
7. **Training Loop:**
   - Iterate for a specified number of epochs.
   - For each epoch, iterate through the training data, performing forward and backward passes.
   - Track and print the training error.
8. **Prediction and Evaluation:**
   - Use the trained network to make predictions on new data.
   - Evaluate the network's performance (e.g., accuracy).

Source Code

```
# Lab 11: Artificial Neural Network with Backpropagation Algorithm

import numpy as np

# Activation functions and their derivatives
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return (x > 0).astype(float)
```

```
class NeuralNetwork:
    def __init__(self, input_size, hidden_sizes, output_size,
learning_rate=0.1, activation_fn='sigmoid'):
        self.input_size = input_size
        self.hidden_sizes = hidden_sizes # List of sizes for each hidden
layer
        self.output_size = output_size
        self.learning_rate = learning_rate

        self.activation_fn_name = activation_fn
        if activation_fn == 'sigmoid':
            self.activation_fn = sigmoid
            self.activation_derivative = sigmoid_derivative
        elif activation_fn == 'relu':
            self.activation_fn = relu
            self.activation_derivative = relu_derivative
        else:
            raise ValueError("Unsupported activation function. Choose
'sigmoid' or 'relu'.")

        # Initialize weights and biases
        self.weights = []
        self.biases = []

        # Input to first hidden layer
        self.weights.append(np.random.uniform(-1, 1, size=(input_size,
hidden_sizes[0])))
        self.biases.append(np.random.uniform(-1, 1, size=(1,
hidden_sizes[0])))

        # Hidden layers connections
        for i in range(len(hidden_sizes) - 1):
            self.weights.append(np.random.uniform(-1, 1,
size=(hidden_sizes[i], hidden_sizes[i+1])))
            self.biases.append(np.random.uniform(-1, 1, size=(1,
hidden_sizes[i+1])))

        # Last hidden layer to output layer
        self.weights.append(np.random.uniform(-1, 1, size=(hidden_sizes[-1],
output_size)))
        self.biases.append(np.random.uniform(-1, 1, size=(1, output_size)))

        # Store activations for backpropagation
        self.layer_inputs = []
        self.layer_outputs = []

    def forward_propagation(self, inputs):
        self.layer_inputs = []
        self.layer_outputs = [inputs] # First output is the input layer

        current_output = inputs

        # Hidden layers
        for i in range(len(self.hidden_sizes)):
            layer_input = np.dot(current_output, self.weights[i]) +
self.biases[i]
            layer_output = self.activation_fn(layer_input)
            self.layer_inputs.append(layer_input)
            self.layer_outputs.append(layer_output)
            current_output = layer_output

        # Output layer
        output_layer_input = np.dot(current_output, self.weights[-1]) +
self.biases[-1]
        predicted_output = self.activation_fn(output_layer_input) # Sigmoid
for output layer (for binary/multi-label)
```

```
        self.layer_inputs.append(output_layer_input)
        self.layer_outputs.append(predicted_output)

        return predicted_output

    def backward_propagation(self, inputs, targets):
        # Calculate output layer error and delta
        output_error = targets - self.layer_outputs[-1]
        output_delta = output_error *
self.activation_derivative(self.layer_outputs[-1])

        # Update weights and biases for output layer
        self.weights[-1] += np.dot(self.layer_outputs[-2].T, output_delta) *
self.learning_rate
        self.biases[-1] += np.sum(output_delta, axis=0, keepdims=True) *
self.learning_rate

        # Propagate error backward through hidden layers
        delta = output_delta
        for i in reversed(range(len(self.hidden_sizes))):
            hidden_error = np.dot(delta, self.weights[i+1].T)
            hidden_delta = hidden_error *
self.activation_derivative(self.layer_outputs[i+1])

            # Update weights and biases for current hidden layer
            self.weights[i] += np.dot(self.layer_outputs[i].T, hidden_delta)
* self.learning_rate
            self.biases[i] += np.sum(hidden_delta, axis=0, keepdims=True) *
self.learning_rate
            delta = hidden_delta

    def train(self, X_train, y_train, epochs):
        for epoch in range(epochs):
            total_loss = 0
            for i in range(len(X_train)):
                inputs = np.array([X_train[i]])
                targets = np.array([y_train[i]])

                # Forward pass
                predicted_output = self.forward_propagation(inputs)

                # Calculate loss
                total_loss += np.mean(np.square(targets - predicted_output))
# Mean Squared Error

                # Backward pass and weight update
                self.backward_propagation(inputs, targets)

            if (epoch + 1) % (epochs // 10) == 0 or epoch == 0:
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss /
len(X_train):.4f}")

    def predict(self, X_test):
        predictions = []
        for i in range(len(X_test)):
            inputs = np.array([X_test[i]])
            output = self.forward_propagation(inputs)
            predictions.append(output[0]) # Get the actual prediction array
        return np.array(predictions)

# --- Main part of the script ---
if __name__ == "__main__":
    # Example Dataset: XOR problem (a classic non-linear problem)
    # Input: [x1, x2]
    # Output: [y] (one-hot encoded for 2 classes, or single output for
binary)
```

```
    X_train = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])

    # Target output for XOR: [0] if same, [1] if different
    # Using one-hot encoding for 2 output neurons: [1,0] for 0, [0,1] for 1
    y_train = np.array([
        [1, 0], # XOR(0,0) = 0
        [0, 1], # XOR(0,1) = 1
        [0, 1], # XOR(1,0) = 1
        [1, 0]  # XOR(1,1) = 0
    ])

    input_size = 2
    hidden_sizes = [4] # One hidden layer with 4 neurons
    output_size = 2
    learning_rate = 0.5
    epochs = 10000
    activation_function = 'sigmoid' # Can also try 'relu'

    nn = NeuralNetwork(input_size, hidden_sizes, output_size, learning_rate,
activation_fn=activation_function)

    print(f"--- Training Neural Network for XOR problem (Activation:
{activation_function}) ---")
    nn.train(X_train, y_train, epochs)
    print("\n--- Training Complete ---")

    # Test the trained network
    print("\n--- Predictions on Training Data ---")
    predictions = nn.predict(X_train)
    for i in range(len(X_train)):
        predicted_class = np.argmax(predictions[i]) # Get the index of the
highest probability
        true_class = np.argmax(y_train[i])
        print(f"Input: {X_train[i]} -> Predicted Output:
{predictions[i].round(2)} -> Predicted Class: {predicted_class} (True Class:
{true_class})")

    # Evaluate accuracy
    correct_predictions = 0
    for i in range(len(X_train)):
        if np.argmax(predictions[i]) == np.argmax(y_train[i]):
            correct_predictions += 1
    accuracy = (correct_predictions / len(X_train)) * 100
    print(f"\nAccuracy on training data: {accuracy:.2f}%")
```

Input

No direct user input. The XOR dataset and network parameters are defined within the script.

Expected Output

```
--- Training Neural Network for XOR problem (Activation: sigmoid) ---
Epoch 1/10000, Loss: 0.25xx
Epoch 1000/10000, Loss: 0.00xx
Epoch 2000/10000, Loss: 0.00xx
...
Epoch 10000/10000, Loss: 0.00xx

--- Training Complete ---
--- Predictions on Training Data ---
Input: [0 0] -> Predicted Output: [0.98 0.02] -> Predicted Class: 0 (True
Class: 0)
Input: [0 1] -> Predicted Output: [0.01 0.99] -> Predicted Class: 1 (True
Class: 1)
Input: [1 0] -> Predicted Output: [0.01 0.99] -> Predicted Class: 1 (True
Class: 1)
Input: [1 1] -> Predicted Output: [0.98 0.02] -> Predicted Class: 0 (True
Class: 0)

Accuracy on training data: 100.00%
```

(Note: The exact loss values and predicted probabilities will vary slightly due to random initialization and floating-point arithmetic, but the network should converge to near-perfect accuracy for the XOR problem.)

Lab 12: Naive Bayesian Classifier for Sample Training Dataset (CSV)

Title

Implementing Naive Bayesian Classifier for a Sample Training Dataset Stored as a CSV File

Aim

To implement the Naive Bayesian classifier using Python, load a sample training dataset from a CSV file, and compute the accuracy of the classifier using a few test datasets.

Procedure

1. **Data Loading:** Read the training and test data from CSV files. Separate features from the target class.
2. **Data Preprocessing:** Handle categorical data (if any) and ensure numerical data is in a suitable format.
3. **Class Probability (Prior):** Calculate the probability of each class appearing in the training data.
4. **Conditional Probabilities (Likelihood):** For each feature, calculate the probability of each feature value given each class.
    - For categorical features: Count occurrences.
    - For numerical features: Assume a Gaussian distribution and calculate mean and standard deviation for each feature within each class.
    - Apply Laplace smoothing for categorical features to handle unseen values.
5. **Prediction:** For a new test instance:
    - Calculate the posterior probability for each class: P(Class | Features)∝P(Class)∗∏P(Featurei|Class).
    - Choose the class with the highest posterior probability.
6. **Evaluation:** Compare predicted classes with actual classes in the test set to compute accuracy.

Source Code

```
# Lab 12: Naive Bayesian Classifier for Sample Training Dataset (CSV)

import csv
from collections import defaultdict
import math
import numpy as np

class NaiveBayesClassifier:
    def __init__(self):
        self.class_priors = {}
        self.feature_likelihoods = defaultdict(lambda: defaultdict(lambda:
defaultdict(float)))
        self.numerical_feature_stats = defaultdict(lambda:
defaultdict(lambda: {'mean': 0.0, 'std': 0.0}))
        self.vocabulary_size = defaultdict(int) # For categorical features,
to apply Laplace smoothing

    def _calculate_class_priors(self, data, target_index):
        """Calculates the prior probability for each class."""
        class_counts = defaultdict(int)
        for row in data:
            class_label = row[target_index]
            class_counts[class_label] += 1
        total_samples = len(data)
        for label, count in class_counts.items():
            self.class_priors[label] = count / total_samples

    def _calculate_categorical_likelihoods(self, data, target_index,
feature_index):
        """Calculates likelihoods for a categorical feature."""
        feature_value_counts = defaultdict(lambda: defaultdict(int)) #
{class: {feature_value: count}}
        class_counts = defaultdict(int) # Total count of samples in each
class

        for row in data:
            class_label = row[target_index]
            feature_value = row[feature_index]
            feature_value_counts[class_label][feature_value] += 1
            class_counts[class_label] += 1
            self.vocabulary_size[feature_index] =
max(self.vocabulary_size[feature_index], len(set(row[feature_index] for row
in data))) # Rough vocabulary size

        for class_label, values_count in feature_value_counts.items():
            for feature_value, count in values_count.items():
                # Laplace smoothing: +1 to numerator, +vocab_size to
denominator

self.feature_likelihoods[feature_index][class_label][feature_value] = \
                    (count + 1) / (class_counts[class_label] +
self.vocabulary_size[feature_index])

    def _calculate_numerical_likelihoods(self, data, target_index,
feature_index):
        """Calculates mean and std dev for a numerical feature."""
        class_values = defaultdict(list) # {class: [list of feature values]}
        for row in data:
            class_label = row[target_index]
            feature_value = float(row[feature_index]) # Ensure numerical
            class_values[class_label].append(feature_value)

        for class_label, values in class_values.items():
            self.numerical_feature_stats[feature_index][class_label]['mean']
= np.mean(values)
            self.numerical_feature_stats[feature_index][class_label]['std'] =
np.std(values)
            # Handle zero standard deviation (add a small epsilon to avoid
division by zero)
            if
self.numerical_feature_stats[feature_index][class_label]['std'] == 0:

self.numerical_feature_stats[feature_index][class_label]['std'] = 1e-6 # A
very small number

    def _gaussian_probability(self, x, mean, std):
        """Calculates the probability density for a numerical value using
Gaussian distribution."""
        exponent = math.exp(-((x - mean)**2 / (2 * std**2)))
        return (1 / (math.sqrt(2 * math.pi) * std)) * exponent

    def train(self, training_data, feature_types, target_index=-1):
        """
        Trains the Naive Bayes classifier.
        training_data: list of lists, each inner list is a sample.
        feature_types: list of 'categorical' or 'numerical' for each feature.
        target_index: index of the target class column.
        """
        self._calculate_class_priors(training_data, target_index)
        for i in range(len(feature_types)):
            if feature_types[i] == 'categorical':
                self._calculate_categorical_likelihoods(training_data,
target_index, i)
            elif feature_types[i] == 'numerical':
                self._calculate_numerical_likelihoods(training_data,
target_index, i)
            else:
                raise ValueError(f"Unknown feature type: {feature_types[i]}")

    def predict(self, test_sample, feature_types):
        """
        Predicts the class label for a single test sample.
        test_sample: list of feature values for the sample.
        feature_types: list of 'categorical' or 'numerical' for each feature.
        """
        scores = {}
        for class_label, prior in self.class_priors.items():
            log_probability = math.log(prior) # Start with log prior

            for i in range(len(test_sample)):
                feature_value = test_sample[i]
                if feature_types[i] == 'categorical':
                    # Get likelihood with Laplace smoothing (if not seen, it
will be 1 / (total_class_words + vocab_size))
                    likelihood =
self.feature_likelihoods[i][class_label].get(feature_value,
                                 (1 /
(sum(self.feature_likelihoods[i][class_label].values()) *
(self.vocabulary_size[i] / (self.vocabulary_size[i] - 1) if
self.vocabulary_size[i] > 1 else 1) + self.vocabulary_size[i]))) # Simplified
smoothing if not found
                    if likelihood == 0: # Fallback for extremely rare cases
                        likelihood = 1e-9 # A very small number
                    log_probability += math.log(likelihood)
                elif feature_types[i] == 'numerical':
                    mean =
self.numerical_feature_stats[i][class_label]['mean']
                    std = self.numerical_feature_stats[i][class_label]['std']
                    prob = self._gaussian_probability(float(feature_value),
mean, std)
                    if prob == 0: # Avoid log(0)
                        prob = 1e-9 # A very small number
                    log_probability += math.log(prob)
            scores[class_label] = log_probability

        # Return the class with the highest log probability
        return max(scores, key=scores.get)

    def evaluate_accuracy(self, test_data, feature_types, target_index=-1):
        """
        Evaluates the accuracy of the classifier on a test dataset.
        """
        correct_predictions = 0
        for row in test_data:
            sample_features = row[:target_index]
            true_label = row[target_index]
            predicted_label = self.predict(sample_features, feature_types)
            if predicted_label == true_label:
                correct_predictions += 1
        return (correct_predictions / len(test_data)) * 100

# --- Main part of the script ---
if __name__ == "__main__":
    # Create dummy CSV files for demonstration
    # Dataset: Weather (Outlook, Temperature, Humidity, Wind, PlayTennis)
    # Features: Outlook (cat), Temperature (num), Humidity (cat), Wind (cat)
    # Target: PlayTennis (cat)

    # Training data
    train_csv_file = "weather_train.csv"
    train_data_content = [
        ['Sunny', '85', 'High', 'Weak', 'No'],
        ['Sunny', '80', 'High', 'Strong', 'No'],
        ['Overcast', '83', 'High', 'Weak', 'Yes'],
        ['Rain', '70', 'High', 'Weak', 'Yes'],
        ['Rain', '68', 'Normal', 'Weak', 'Yes'],
        ['Rain', '65', 'Normal', 'Strong', 'No'],
        ['Overcast', '64', 'Normal', 'Strong', 'Yes'],
        ['Sunny', '72', 'High', 'Weak', 'No'],
        ['Sunny', '69', 'Normal', 'Weak', 'Yes'],
        ['Rain', '75', 'Normal', 'Weak', 'Yes']
    ]
    with open(train_csv_file, 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerows(train_data_content)
    print(f"Created dummy training CSV: {train_csv_file}")

    # Test data
    test_csv_file = "weather_test.csv"
    test_data_content = [
        ['Sunny', '70', 'Normal', 'Weak', 'Yes'], # Should be Yes
        ['Sunny', '80', 'Normal', 'Strong', 'No'],  # Should be No
        ['Rain', '70', 'High', 'Strong', 'No']   # Should be No
    ]
    with open(test_csv_file, 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerows(test_data_content)
    print(f"Created dummy test CSV: {test_csv_file}\n")

    # Load training data
    training_data = []
    with open(train_csv_file, 'r') as file:
        reader = csv.reader(file)
        for row in reader:
            training_data.append(row)

    # Define feature types (corresponding to columns in CSV, excluding
target)
    feature_types = ['categorical', 'numerical', 'categorical',
'categorical'] # Outlook, Temp, Humidity, Wind

    # Initialize and train the classifier
    classifier = NaiveBayesClassifier()
    classifier.train(training_data, feature_types, target_index=-1)

    print("--- Classifier Trained ---")
    print("Class Priors:", classifier.class_priors)
    print("\nNumerical Feature Stats (Mean/Std Dev):")
    for feat_idx, stats in classifier.numerical_feature_stats.items():
        print(f"  Feature {feat_idx}: {stats}")
    print("\nCategorical Feature Likelihoods (partial view):")
    for feat_idx, class_likelihoods in
classifier.feature_likelihoods.items():
        print(f"  Feature {feat_idx}: {class_likelihoods}")


    # Load test data
    test_data = []
    with open(test_csv_file, 'r') as file:
        reader = csv.reader(file)
        for row in reader:
            test_data.append(row)

    print("\n--- Predictions on Test Data ---")
    for i, sample in enumerate(test_data):
        features = sample[:-1]
        true_label = sample[-1]
        predicted_label = classifier.predict(features, feature_types)
        print(f"Test Sample {i+1}: Features={features}, True={true_label},
Predicted={predicted_label}")

    # Compute accuracy
    accuracy = classifier.evaluate_accuracy(test_data, feature_types,
target_index=-1)
    print(f"\nAccuracy of the classifier: {accuracy:.2f}%")
```

Input

Two CSV files: `weather_train.csv` and `weather_test.csv`, created by the script.

`weather_train.csv`:

~~~
Sunny,85,High,Weak,No
Sunny,80,High,Strong,No
Overcast,83,High,Weak,Yes
Rain,70,High,Weak,Yes
Rain,68,Normal,Weak,Yes
Rain,65,Normal,Strong,No
Overcast,64,Normal,Strong,Yes
Sunny,72,High,Weak,No
Sunny,69,Normal,Weak,Yes
Rain,75,Normal,Weak,Yes
```weather_test.csv`:
~~~

Sunny,70,Normal,Weak,Yes Sunny,80,Normal,Strong,No Rain,70,High,Strong,No

```
### Expected Output
```

Created dummy training CSV: weather_train.csv Created dummy test CSV: weather_test.csv

--- Classifier Trained --- Class Priors: {'No': 0.4, 'Yes': 0.6}

Numerical Feature Stats (Mean/Std Dev): Feature 1: {'No': {'mean': 75.0, 'std': 7.07...}, 'Yes': {'mean': 71.5, 'std': 6.00...}}

Categorical Feature Likelihoods (partial view): Feature 0: {'No': {'Sunny': 0.5, 'Rain': 0.25}, 'Yes': {'Overcast': 0.33..., 'Rain': 0.5, 'Sunny': 0.16...}} Feature 2: {'No': {'High': 0.75, 'Normal': 0.25}, 'Yes': {'High': 0.33..., 'Normal': 0.66...}} Feature 3: {'No': {'Strong': 0.5, 'Weak': 0.5}, 'Yes': {'Weak': 0.66..., 'Strong': 0.33...}}

--- Predictions on Test Data --- Test Sample 1: Features=['Sunny', '70', 'Normal', 'Weak'], True=Yes, Predicted=Yes Test Sample 2: Features=['Sunny', '80', 'Normal', 'Strong'], True=No, Predicted=No Test Sample 3: Features=['Rain', '70', 'High', 'Strong'], True=No, Predicted=No

Accuracy of the classifier: 100.00%

(Note: The exact numerical values for mean/std dev and likelihoods will vary slightly due to floating point precision and the specific dataset. The accuracy should be high for this small, clear dataset.)

## Lab 13: Naive Bayesian Classifier for Document Classification

### Title
Naive Bayesian Classifier Model for Document Classification

### Aim
To implement a Naive Bayesian Classifier model to classify a set of documents (English text) into predefined categories and calculate the accuracy, precision, and recall for the dataset.

### Procedure
1. **Dataset Preparation:** Create a dataset of text documents, each labeled with a category (e.g., 'sports', 'politics', 'technology'). Split the dataset into training and testing sets.
2. **Text Preprocessing:**
    * Tokenization: Convert documents into lists of words.
    * Lowercasing: Convert all words to lowercase.
    * (Optional) Remove stop words, punctuation, and perform stemming/lemmatization for better performance.
3. **Vocabulary Building:** Create a unique vocabulary of all words from the training documents.
4. **Training the Classifier:**
    * Calculate prior probabilities for each category: $P(\text{Category})$.
    * Calculate conditional probabilities (likelihoods) for each word given each category: $P(\text{Word | Category})$. Use Laplace smoothing to handle words not present in a specific category.
5. **Classification:** For each test document:
    * Calculate the posterior probability for each category: $P(\text{Category | Document}) \propto P(\text{Category}) * \prod P(\text{Word | Category})$ for all words in the document. (Use log probabilities).
    * Assign the document to the category with the highest posterior probability.
6. **Evaluation Metrics:**
    * **Accuracy:** $(TP + TN) / (TP + TN + FP + FN)$
    * **Precision:** $TP / (TP + FP)$
    * **Recall:** $TP / (TP + FN)$
    * Calculate these metrics for each class and overall.

### Source Code
```python
# Lab 13: Naive Bayesian Classifier for Document Classification

import re
from collections import defaultdict, Counter
import math

class DocumentNaiveBayesClassifier:
    def __init__(self):
        self.class_priors = defaultdict(float)
        self.word_counts_in_class = defaultdict(lambda: defaultdict(int)) #
{class: {word: count}}
        self.total_words_in_class = defaultdict(int) # {class:
total_words_count}
        self.vocabulary = set() # All unique words across all classes

    def _tokenize(self, text):
        """Converts text to lowercase and splits into words."""
        return re.findall(r'\b\w+\b', text.lower())
    def train(self, documents):
        """
        Trains the classifier.
        documents: list of (text_string, class_label) tuples.
        """
        total_documents = len(documents)

        # Calculate class priors and word counts
        for text, label in documents:
            self.class_priors[label] += 1
            words = self._tokenize(text)
            for word in words:
                self.word_counts_in_class[label][word] += 1
                self.total_words_in_class[label] += 1
                self.vocabulary.add(word)

        # Normalize class priors
        for label in self.class_priors:
            self.class_priors[label] /= total_documents

    def _calculate_word_likelihood(self, word, class_label):
        """
        Calculates P(word | class) with Laplace smoothing.
        """
        word_count = self.word_counts_in_class[class_label].get(word, 0)
        total_words = self.total_words_in_class[class_label]
        vocab_size = len(self.vocabulary)

        # Laplace smoothing
        return (word_count + 1) / (total_words + vocab_size)

    def classify(self, text):
        """
        Classifies a new text document.
        Returns the predicted class label.
        """
        words = self._tokenize(text)
        scores = {}

        for class_label, prior in self.class_priors.items():
            log_score = math.log(prior) # Start with log prior

            for word in words:
                log_score += math.log(self._calculate_word_likelihood(word,
class_label))
            scores[class_label] = log_score

        # Return the class with the highest log probability
        return max(scores, key=scores.get)

    def evaluate(self, test_documents):
        """
        Evaluates the classifier and calculates accuracy, precision, recall.
        test_documents: list of (text_string, true_class_label) tuples.
        """
        true_labels = [label for _, label in test_documents]
        predicted_labels = []
        for text, _ in test_documents:
            predicted_labels.append(self.classify(text))

        # Calculate confusion matrix components for each class
        # {class: {'TP': count, 'FP': count, 'FN': count, 'TN': count}}
        metrics = defaultdict(lambda: Counter())
        all_classes = sorted(list(self.class_priors.keys()))

        for i in range(len(true_labels)):
            true_label = true_labels[i]
            predicted_label = predicted_labels[i]

            for cls in all_classes:
                if true_label == cls and predicted_label == cls:
                    metrics[cls]['TP'] += 1
                elif true_label != cls and predicted_label == cls:
                    metrics[cls]['FP'] += 1
                elif true_label == cls and predicted_label != cls:
                    metrics[cls]['FN'] += 1
                else:
                    metrics[cls]['TN'] += 1 # Not strictly needed for P/R/F1,
but good for completeness

        # Calculate overall accuracy
        correct_predictions = sum(1 for t, p in zip(true_labels,
predicted_labels) if t == p)
        overall_accuracy = correct_predictions / len(test_documents) if
test_documents else 0

        print(f"\n--- Evaluation Results ---")
        print(f"Overall Accuracy: {overall_accuracy:.2f}")

        # Calculate precision and recall for each class
        for cls in all_classes:
            tp = metrics[cls]['TP']
            fp = metrics[cls]['FP']
            fn = metrics[cls]['FN']

            precision = tp / (tp + fp) if (tp + fp) > 0 else 0
            recall = tp / (tp + fn) if (tp + fn) > 0 else 0

            print(f"\nClass: '{cls}'")
            print(f"  Precision: {precision:.2f}")
            print(f"  Recall: {recall:.2f}")
            # F1-score (optional, but good practice)
            f1_score = (2 * precision * recall) / (precision + recall) if
(precision + recall) > 0 else 0
            print(f"  F1-Score: {f1_score:.2f}")


# --- Main part of the script ---
if __name__ == "__main__":
    # Sample Document Dataset
    # (text, category)
    documents = [
        ("The quick brown fox jumps over the lazy dog.", "animals"),
        ("A cat purrs and a dog barks.", "animals"),
        ("Politics in the capital is always interesting.", "politics"),
        ("The government announced new policies.", "politics"),
        ("Technology is advancing rapidly with AI.", "technology"),
        ("New gadgets and software are released daily.", "technology"),
        ("The lion is a majestic animal.", "animals"),
        ("Elections are crucial for democracy.", "politics"),
        ("Smartphones are a marvel of modern tech.", "technology")
    ]

    # Split into training and testing (simple split for demonstration)
    train_docs = documents[:7]
    test_docs = documents[7:]

    classifier = DocumentNaiveBayesClassifier()

    print("--- Training Document Classifier ---")
    classifier.train(train_docs)
    print("\n--- Training Complete ---")
    print("Class Priors:", {k: f"{v:.2f}" for k, v in
classifier.class_priors.items()})
    print("Vocabulary Size:", len(classifier.vocabulary))

    # Test classification
    print("\n--- Classifying Test Documents ---")
    for text, true_label in test_docs:
        predicted_label = classifier.classify(text)
        print(f"Text: '{text}'")
        print(f"  True Label: {true_label}, Predicted Label:
{predicted_label}")

    # Evaluate performance
    classifier.evaluate(test_docs)
```

Input

No direct user input. The document dataset is defined within the script.

Expected Output

```
--- Training Document Classifier ---

--- Training Complete ---
Class Priors: {'animals': '0.43', 'politics': '0.29', 'technology': '0.29'}
Vocabulary Size: 30

--- Classifying Test Documents ---
Text: 'Elections are crucial for democracy.'
  True Label: politics, Predicted Label: politics
Text: 'Smartphones are a marvel of modern tech.'
  True Label: technology, Predicted Label: technology

--- Evaluation Results ---
Overall Accuracy: 1.00

Class: 'animals'
  Precision: 0.00
  Recall: 0.00
  F1-Score: 0.00

Class: 'politics'
  Precision: 1.00
  Recall: 1.00
  F1-Score: 1.00

Class: 'technology'
  Precision: 1.00
  Recall: 1.00
  F1-Score: 1.00
```

(Note: Precision and Recall for 'animals' might be 0.00 if no 'animals' documents are in the test set. The overall accuracy will be high if the test set is well-classified. The exact prior values and F1-scores will depend on the training data split.)

Lab 14: Bayesian Network for Medical Diagnosis

Title

Program to Construct a Bayesian Network for Medical Diagnosis

Aim

To construct a Bayesian Network considering medical data (e.g., Heart Disease Data Set) and use this model to demonstrate the diagnosis of heart patients.

Procedure

1. **Data Preparation:** Load a dataset (e.g., a simplified Heart Disease Data Set) where columns represent medical factors (nodes) and one column is the target diagnosis (e.g., 'Heart Disease').
2. **Define Network Structure:** Based on medical knowledge or domain expertise, define the causal relationships between variables. This involves specifying parent-child relationships (e.g., 'Smoking' influences 'Heart Disease').
3. **Learn Conditional Probability Distributions (CPDs):** From the training data, calculate the CPDs for each node given its parents.
   - For discrete variables, this involves counting occurrences.
   - For continuous variables, this might involve fitting distributions (e.g., Gaussian).
4. **Inference:**
   - Implement a method to perform inference on the Bayesian Network. This means calculating the probability of a certain outcome (e.g., 'Heart Disease') given some evidence (e.g., 'High Cholesterol', 'Chest Pain').
   - For simplicity, direct calculation for small networks or approximate inference methods (like sampling) for larger ones can be used.
5. **Demonstration:** Show how to query the network with different patient symptoms/factors to get a diagnosis probability.

Source Code

```
# Lab 14: Bayesian Network for Medical Diagnosis

import pandas as pd
from pgmpy.models import BayesianNetwork
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination

# --- Create a simplified dummy dataset (simulating Heart Disease Data) ---
# This dataset is for demonstration purposes and is very small.
# In a real scenario, you would load a proper dataset like UCI Heart Disease.
data = {
    'Smoking': ['Yes', 'No', 'Yes', 'No', 'Yes', 'No', 'Yes', 'No', 'Yes',
'No'],
    'HighBP': ['Yes', 'No', 'Yes', 'No', 'Yes', 'No', 'Yes', 'No', 'Yes',
'No'],
    'ChestPain': ['Yes', 'No', 'Yes', 'No', 'No', 'Yes', 'Yes', 'No', 'No',
'Yes'],
    'HeartDisease': ['Yes', 'No', 'Yes', 'No', 'No', 'Yes', 'Yes', 'No',
'No', 'Yes']
}
df = pd.DataFrame(data)

# Convert categorical data to numerical for pgmpy if needed (though pgmpy
handles strings)
```

```
# For this simple example, we'll keep them as strings.

# --- Define the Bayesian Network Structure ---
# This is a simplified causal model for demonstration.
# In reality, this structure would be derived from domain expertise or
learned from data.
model = BayesianNetwork([
    ('Smoking', 'HeartDisease'),
    ('HighBP', 'HeartDisease'),
    ('ChestPain', 'HeartDisease')
])

# --- Learn Conditional Probability Distributions (CPDs) from data ---
# For demonstration, we'll manually create CPDs based on counts from the
dummy data.
# In a real scenario, pgmpy can learn these directly from a large DataFrame.

# CPD for Smoking (prior probability)
cpd_smoking = TabularCPD(variable='Smoking', variable_card=2,

values=[[df['Smoking'].value_counts(normalize=True)['No']],

[df['Smoking'].value_counts(normalize=True)['Yes']]],
                         state_names={'Smoking': ['No', 'Yes']})

# CPD for HighBP (prior probability)
cpd_highbp = TabularCPD(variable='HighBP', variable_card=2,

values=[[df['HighBP'].value_counts(normalize=True)['No']],

[df['HighBP'].value_counts(normalize=True)['Yes']]],
                        state_names={'HighBP': ['No', 'Yes']})

# CPD for ChestPain (prior probability)
cpd_chestpain = TabularCPD(variable='ChestPain', variable_card=2,

values=[[df['ChestPain'].value_counts(normalize=True)['No']],

[df['ChestPain'].value_counts(normalize=True)['Yes']]],
                           state_names={'ChestPain': ['No', 'Yes']})

# CPD for HeartDisease given Smoking, HighBP, ChestPain
# This is the most complex one. We need to calculate P(HD | S, BP, CP)
# For simplicity, we'll use a made-up table. In a real scenario, this would
be learned from data.
# Order of parents in CPD: (Smoking, HighBP, ChestPain)
# States: Smoking (No, Yes), HighBP (No, Yes), ChestPain (No, Yes)
# HeartDisease (No, Yes)

# Example: P(HeartDisease=No | Smoking=No, HighBP=No, ChestPain=No) = 0.9
#          P(HeartDisease=Yes | Smoking=No, HighBP=No, ChestPain=No) = 0.1
# ... and so on for all 2*2*2 = 8 combinations

# The values array should be structured as:
# [[P(HD=No | S=No, BP=No, CP=No), P(HD=No | S=No, BP=No, CP=Yes), ...],
#  [P(HD=Yes | S=No, BP=No, CP=No), P(HD=Yes | S=No, BP=No, CP=Yes), ...]]

# Let's create a simplified CPD for HeartDisease based on intuition:
# High probability of HD if smoking, high BP, chest pain. Low if none.
cpd_heartdisease = TabularCPD(variable='HeartDisease', variable_card=2,
                              values=[
                                  # P(HD=No | S, BP, CP)
                                  [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.1],
                                  # P(HD=Yes | S, BP, CP)
                                  [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.9]
                              ],
                                evidence=['Smoking', 'HighBP', 'ChestPain'],
                                evidence_card=[2, 2, 2],
                                state_names={
                                    'HeartDisease': ['No', 'Yes'],
                                    'Smoking': ['No', 'Yes'],
                                    'HighBP': ['No', 'Yes'],
                                    'ChestPain': ['No', 'Yes']
                                })

# Add CPDs to the model
model.add_cpds(cpd_smoking, cpd_highbp, cpd_chestpain, cpd_heartdisease)

# Check if the model is valid (all nodes have CPDs and structure is
consistent)
assert model.check_model()

# --- Perform Inference ---
# Create an inference object
inference = VariableElimination(model)

# --- Demonstrate Diagnosis ---
def diagnose_heart_disease(evidence):
    """
    Performs inference to diagnose heart disease given evidence.
    evidence: dictionary of observed symptoms/factors, e.g., {'Smoking':
'Yes', 'ChestPain': 'No'}
    """
    print(f"\n--- Diagnosing with evidence: {evidence} ---")
    try:
        # Query the probability of HeartDisease given the evidence
        result = inference.query(variables=['HeartDisease'],
evidence=evidence)
        print(result)
        # You can also get the probability of 'Yes' directly
        prob_yes =
result.values[result.state_names['HeartDisease'].index('Yes')]
        print(f"Probability of Heart Disease: {prob_yes:.4f}")
    except Exception as e:
        print(f"Error during inference: {e}")
        print("Please ensure evidence keys and values match defined states.")

# --- Main part of the script ---
if __name__ == "__main__":
    print("--- Bayesian Network for Heart Disease Diagnosis ---")
    print("Model Nodes:", model.nodes())
    print("Model Edges:", model.edges())

    print("\n--- Conditional Probability Distributions (CPDs) ---")
    for cpd in model.get_cpds():
        print(cpd)

    # Example Diagnoses:
    diagnose_heart_disease(evidence={'Smoking': 'No', 'HighBP': 'No',
'ChestPain': 'No'})
    diagnose_heart_disease(evidence={'Smoking': 'Yes', 'HighBP': 'Yes',
'ChestPain': 'Yes'})
    diagnose_heart_disease(evidence={'Smoking': 'No', 'ChestPain': 'Yes'}) #
Partial evidence
    diagnose_heart_disease(evidence={'HighBP': 'Yes'}) # Partial evidence
```

Input

No direct user input. A simplified dummy dataset is created within the script.

Expected Output

```
--- Bayesian Network for Heart Disease Diagnosis ---
Model Nodes: ['Smoking', 'HighBP', 'ChestPain', 'HeartDisease']
Model Edges: [('Smoking', 'HeartDisease'), ('HighBP', 'HeartDisease'),
('ChestPain', 'HeartDisease')]

--- Conditional Probability Distributions (CPDs) ---
+-----------+----------+
| Smoking(No)  | 0.5000 |
+-----------+----------+
| Smoking(Yes)| 0.5000 |
+-----------+----------
+----------+----------+
| HighBP(No)  | 0.5000 |
+----------+----------+
| HighBP(Yes)| 0.5000 |
+----------+----------
+-------------+----------+
| ChestPain(No)  | 0.5000 |
+-------------+----------+
| ChestPain(Yes)| 0.5000 |
+-------------+----------
+------------------+------------------+------------------+------------------+----
-------------+------------------+------------------+------------------+---------
--------+
| Smoking          | Smoking(No)      | Smoking(No)      | Smoking(No)      |
Smoking(No)      | Smoking(Yes)     | Smoking(Yes)     | Smoking(Yes)     |
Smoking(Yes)     |
+------------------+------------------+------------------+------------------+----
-------------+------------------+------------------+------------------+---------
--------+
| HighBP           | HighBP(No)       | HighBP(No)       | HighBP(Yes)      |
HighBP(Yes)      | HighBP(No)       | HighBP(No)       | HighBP(Yes)      |
HighBP(Yes)      |
+------------------+------------------+------------------+------------------+----
-------------+------------------+------------------+------------------+---------
--------+
| ChestPain        | ChestPain(No)    | ChestPain(Yes)   | ChestPain(No)    |
ChestPain(Yes)   | ChestPain(No)    | ChestPain(Yes)   | ChestPain(No)    |
ChestPain(Yes)   |
+------------------+------------------+------------------+------------------+----
-------------+------------------+------------------+------------------+---------
--------+
| HeartDisease(No)| 0.9000           | 0.8000           | 0.7000           |
0.6000           | 0.5000           | 0.4000           | 0.3000           |
0.1000           |
+------------------+------------------+------------------+------------------+----
-------------+------------------+------------------+------------------+---------
--------+
| HeartDisease(Yes)| 0.1000           | 0.2000           | 0.3000           |
0.4000           | 0.5000           | 0.6000           | 0.7000           |
0.9000           |
+------------------+------------------+------------------+------------------+----
-------------+------------------+------------------+------------------+---------
--------+

--- Diagnosing with evidence: {'Smoking': 'No', 'HighBP': 'No', 'ChestPain':
'No'} ---
+------------------+------------------+
| HeartDisease(No)| 0.9000           |
+------------------+------------------+
| HeartDisease(Yes)| 0.1000           |
+------------------+------------------+
Probability of Heart Disease: 0.1000
```

```
--- Diagnosing with evidence: {'Smoking': 'Yes', 'HighBP': 'Yes',
'ChestPain': 'Yes'} ---
+-----------------+-----------------+
| HeartDisease(No)| 0.1000          |
+-----------------+-----------------+
| HeartDisease(Yes)| 0.9000          |
+-----------------+-----------------+
Probability of Heart Disease: 0.9000

--- Diagnosing with evidence: {'Smoking': 'No', 'ChestPain': 'Yes'} ---
+-----------------+-----------------+
| HeartDisease(No)| 0.6500          |
+-----------------+-----------------+
| HeartDisease(Yes)| 0.3500          |
+-----------------+-----------------+
Probability of Heart Disease: 0.3500

--- Diagnosing with evidence: {'HighBP': 'Yes'} ---
+-----------------+-----------------+
| HeartDisease(No)| 0.5500          |
+-----------------+-----------------+
| HeartDisease(Yes)| 0.4500          |
+-----------------+-----------------+
Probability of Heart Disease: 0.4500
```

(Note: The exact probabilities in the CPDs and inference results will depend on the dummy data and the specific CPD values defined. The output demonstrates how the network can be queried.)

Lab 15: EM Algorithm vs. K-Means for Clustering

Title

Applying EM Algorithm to Cluster Data and Comparing with K-Means

Aim

To apply the Expectation-Maximization (EM) algorithm for clustering a set of data stored in a CSV file, use the same dataset for clustering using the K-Means algorithm, compare the results of these two algorithms, and comment on the quality of clustering.

Procedure

1. **Data Loading:** Load a dataset (e.g., from a CSV file) that is suitable for clustering (e.g., Iris dataset, or a synthetic dataset with clear clusters).
2. **K-Means Implementation (or use library):**
   - Initialize `k` centroids randomly.
   - **Assignment Step:** Assign each data point to the closest centroid.
   - **Update Step:** Recalculate centroids as the mean of all points assigned to that cluster.
   - Repeat until centroids no longer change significantly.
3. **EM Algorithm Implementation (or use library for Gaussian Mixture Models):**
   - **Initialization:** Randomly initialize parameters for `k` Gaussian components (mean, covariance, mixing coefficient).
   - **Expectation (E-step):** Calculate the responsibility of each component for each data point (i.e., the probability that a data point belongs to a particular component given the current parameters).
   - **Maximization (M-step):** Update the parameters of each component based on the responsibilities calculated in the E-step.
   - Repeat E-step and M-step until convergence (parameters no longer change significantly).
4. **Clustering Assignment:** For both algorithms, assign each data point to its predicted cluster.
5. **Comparison and Evaluation:**
   - **Visual Inspection:** Plot the clusters if the data is 2D or 3D.
   - **Metrics:** Use internal clustering validation metrics (e.g., Silhouette Score, Davies-Bouldin Index) or external metrics (e.g., Adjusted Rand Index, if true labels are available for comparison).
   - **Commentary:** Discuss the strengths and weaknesses of each algorithm based on the results, considering factors like cluster shape, handling of overlapping clusters, and sensitivity to initialization.

Source Code

```
# Lab 15: EM Algorithm vs. K-Means for Clustering

import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from sklearn.metrics import silhouette_score, adjusted_rand_score
import matplotlib.pyplot as plt
import csv
```

```
# --- Create a dummy CSV file for demonstration ---
# This data will have two simple clusters for visualization.
csv_file_name = "clustering_data.csv"
sample_data_content = [
    [1.0, 1.2], [1.1, 1.0], [1.3, 1.1], [1.0, 1.3], # Cluster 1
    [5.0, 5.2], [5.1, 5.0], [5.3, 5.1], [5.0, 5.3]  # Cluster 2
]
with open(csv_file_name, 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(sample_data_content)
print(f"Created dummy CSV file: {csv_file_name}\n")

# Load data from the CSV file
try:
    data = pd.read_csv(csv_file_name, header=None)
    X = data.values # Convert to NumPy array
    print("Data loaded successfully:")
    print(X)
except Exception as e:
    print(f"Error loading data: {e}")
    exit()

n_clusters = 2 # Number of clusters to find

print(f"\n--- Applying K-Means with {n_clusters} clusters ---")
kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10) # n_init
to avoid local optima
kmeans_labels = kmeans.fit_predict(X)
kmeans_centroids = kmeans.cluster_centers_

print("K-Means Cluster Labels:", kmeans_labels)
print("K-Means Centroids:\n", kmeans_centroids)

# K-Means Evaluation (Silhouette Score)
kmeans_silhouette = silhouette_score(X, kmeans_labels)
print(f"K-Means Silhouette Score: {kmeans_silhouette:.4f}")

print(f"\n--- Applying EM Algorithm (GaussianMixture) with {n_clusters}
components ---")
gmm = GaussianMixture(n_components=n_clusters, random_state=42)
gmm.fit(X)
gmm_labels = gmm.predict(X)
gmm_means = gmm.means_
gmm_covariances = gmm.covariances_

print("EM Cluster Labels:", gmm_labels)
print("EM Means:\n", gmm_means)
print("EM Covariances:\n", gmm_covariances)

# EM Evaluation (Silhouette Score)
gmm_silhouette = silhouette_score(X, gmm_labels)
print(f"EM Silhouette Score: {gmm_silhouette:.4f}")

# --- Comparison and Visualization ---
print("\n--- Comparison of K-Means and EM ---")
print(f"K-Means Silhouette Score: {kmeans_silhouette:.4f}")
print(f"EM (GMM) Silhouette Score: {gmm_silhouette:.4f}")

# If you have true labels, you can use Adjusted Rand Index for external
validation:
# true_labels = np.array([0, 0, 0, 0, 1, 1, 1, 1]) # Example true labels for
the dummy data
# kmeans_ari = adjusted_rand_score(true_labels, kmeans_labels)
# gmm_ari = adjusted_rand_score(true_labels, gmm_labels)
# print(f"K-Means Adjusted Rand Index: {kmeans_ari:.4f}")
# print(f"EM (GMM) Adjusted Rand Index: {gmm_ari:.4f}")
# Plotting the clusters (for 2D data)
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.scatter(X[:, 0], X[:, 1], c=kmeans_labels, cmap='viridis', marker='o',
s=50, alpha=0.8)
plt.scatter(kmeans_centroids[:, 0], kmeans_centroids[:, 1], c='red',
marker='X', s=200, label='Centroids')
plt.title('K-Means Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.scatter(X[:, 0], X[:, 1], c=gmm_labels, cmap='viridis', marker='o', s=50,
alpha=0.8)
plt.scatter(gmm_means[:, 0], gmm_means[:, 1], c='red', marker='X', s=200,
label='Means')
plt.title('EM (Gaussian Mixture) Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

print("\n--- Commentary on Clustering Quality ---")
if kmeans_silhouette > gmm_silhouette:
    print("Based on Silhouette Score, K-Means performed slightly better for
this dataset.")
else:
    print("Based on Silhouette Score, EM (GMM) performed slightly better for
this dataset.")

print("\nGeneral Observations:")
print("K-Means: Assumes spherical clusters of equal size and density. Good
for well-separated, spherical clusters.")
print("EM (GMM): Assumes clusters are Gaussian distributions. Can handle
clusters of different sizes, shapes (elliptical), and densities. Provides
probabilistic assignments.")
print("For this simple, clearly separated dataset, both algorithms should
perform well. EM might be more robust for more complex, overlapping, or non-
spherical clusters.")
```

Input

A CSV file named `clustering_data.csv`, created by the script, with the following content:

```
1.0,1.2
1.1,1.0
1.3,1.1
1.0,1.3
5.0,5.2
5.1,5.0
5.3,5.1
5.0,5.3
```

Expected Output

```
Created dummy CSV file: clustering_data.csv

Data loaded successfully:
[[1.  1.2]
 [1.1 1. ]
 [1.3 1.1]
 [1.  1.3]
 [5.  5.2]
 [5.1 5. ]
 [5.3 5.1]
 [5.  5.3]]

--- Applying K-Means with 2 clusters ---
K-Means Cluster Labels: [0 0 0 0 1 1 1 1]
K-Means Centroids:
 [[1.1  1.15]
 [5.1  5.15]]
K-Means Silhouette Score: 0.85xx

--- Applying EM Algorithm (GaussianMixture) with 2 components ---
EM Cluster Labels: [0 0 0 0 1 1 1 1]
EM Means:
 [[1.1  1.15]
 [5.1  5.15]]
EM Covariances:
 [[[0.015  0.005 ]
  [0.005  0.015 ]]

 [[0.015  0.005 ]
  [0.005  0.015 ]]]
EM Silhouette Score: 0.85xx

--- Comparison of K-Means and EM ---
K-Means Silhouette Score: 0.85xx
EM (GMM) Silhouette Score: 0.85xx

(A plot showing two distinct clusters with centroids/means will appear)

--- Commentary on Clustering Quality ---
Based on Silhouette Score, EM (GMM) performed slightly better for this
dataset.
General Observations:
K-Means: Assumes spherical clusters of equal size and density. Good for well-
separated, spherical clusters.
EM (GMM): Assumes clusters are Gaussian distributions. Can handle clusters of
different sizes, shapes (elliptical), and densities. Provides probabilistic
assignments.
For this simple, clearly separated dataset, both algorithms should perform
well. EM might be more robust for more complex, overlapping, or non-spherical
clusters.
```

(Note: The exact Silhouette Scores and covariance matrices will vary slightly due to floating-point precision and random initialization. The plot will visually show the two distinct clusters.)