GURU TEGH BAHADUR 4TH CENTENARY ENGINEERING COLLEGE

G-8 AREA, RAJOURI GARDEN, NEW DELHI-110064

Artificial Intelligence Applications
PRACTICAL FILE

Course Code: AI-403P

Semester: 7th

Submitted to :

Sarita Yadav

Submitted by :

SAIMA

03423802722

CSE 1

GURU TEGH BAHADUR 4TH CENTENARY ENGINEERING COLLEGE

G-8 AREA, RAJOURI GARDEN, NEW DELHI-110064

ARTIFICIAL INTELLIGENCE APPLICATIONS PRACTICAL FILE

Course Code: AI-403P

Semester: 7th

Submitted to :

Sarita Yadav

Submitted by :

Kanishka Bhatt

01623802722

CSE - 01

S.No.	Experiment Title	Date	Teacher's Sign
1	Write a program to implement **Breadth First Search Traversal**	19/08/25	
2	Write a program to implement **Water Jug Problem**	19/08/25	
3	Write a program to remove **stop words** for a given passage from a text file using **NLTK**	02/09/25	
4	Write a program to solve **Monkey Banana problem** using **Prolog**	02/09/25	
5	Write a program for **POS (Part of Speech) tagging** for the given sentence using **NLTK**	09/09/25	
6	Write a program to implement **Lemmatization** using **NLTK.**	09/09/25	
7	Write a program for **Text Classification** for the given sentence using **NLTK**	16/09/25	
8	Program to demonstrate **Simple Linear Regression**	16/09/25	
9	Program to demonstrate **K-Nearest Neighbor (Flower Classification)**	23/09/25	
10	Program to demonstrate **Naïve Bayes Classifier**	23/09/25	
11	**Greg Viot's Fuzzy Cruise Controller** (New MATLAB Syntax)	14/10/25	
12	**Fuzzy Air Conditioner Controller** (Mamdani FIS)	14/10/25	

PRACTICAL- 1

Write a program to implement Breadth First Search Traversal.

THEORY:

Breadth First Search (BFS):

Breadth First Search is a graph traversal algorithm used to explore nodes of a graph systematically. It starts at a chosen node (often called the "root" or "starting node") and explores all of its neighboring nodes at the present depth level before moving on to the nodes at the next depth level.

Algorithm:

- Start from the given node (usually the root node).
- Initialize an empty queue and enqueue the starting node.
- While the queue is not empty:
- Dequeue a node from the queue.
- Process the dequeued node.
- Enqueue all its adjacent nodes that have not been visited yet.
- Repeat step 3 until the queue is empty.

⇒ BFS uses a queue data structure to keep track of nodes to be visited.
⇒ It guarantees the shortest path to the goal in an unweighted graph.
⇒ BFS is implemented using a "first in, first out" (FIFO) approach, ensuring that nodes are visited in the order they are discovered.
⇒ BFS can be used to find connected components in an undirected graph and to check for bipartiteness.

CODE

```
from collections import deque

class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

```
class Tree:
    def buildtree(self):
        x = int(input('Enter value (-1 for None): '))
        if x == -1:
            return None
        n = Node(x)
        print(f'Enter left child of {x}')
        n.left = self.buildtree()
        print(f'Enter right child of {x}')
        n.right = self.buildtree()
        return n

    def bfs(self):
        if not self.root:
            return
        q = deque()
        q.append(self.root)
        print("BFS Traversal:")
        while q:
            n = q.popleft()
            print(n.data, end=' ')
            if n.left:
                q.append(n.left)
            if n.right:
                q.append(n.right)

if __name__ == '__main__':
    t = Tree()
    t.root = t.buildtree()
    t.bfs()
```

OUTPUT

```
Enter value (-1 for None):  1
Enter left child of 1
Enter value (-1 for None):  2
Enter left child of 2
Enter value (-1 for None):  4
Enter left child of 4
Enter value (-1 for None):  -1
Enter right child of 4
Enter value (-1 for None):  -1
Enter right child of 2
Enter value (-1 for None):  5
Enter left child of 5
Enter value (-1 for None):  -1
Enter right child of 5
Enter value (-1 for None):  -1
Enter right child of 1
Enter value (-1 for None):  3
Enter left child of 3
Enter value (-1 for None):
1
2
3
4
5
```

PRACTICAL- 2

Write a program to implement Water Jug Problem.

THEORY:

The Water Jug Problem is a classic puzzle in computer science and mathematics that involves finding a sequence of steps to measure a specific volume of water using jugs of known capacities. Here's a brief explanation of the theory behind implementing the Water Jug Problem:

Algorithm:

- The Water Jug Problem can be solved using a variation of the Breadth First Search (BFS) algorithm. The idea is to simulate all possible states of the jugs and explore their transitions until the target volume is reached.
- Define the initial state, representing the current volumes of water in each jug.
- Create a queue to store states to be explored, starting with the initial state.
- While the queue is not empty:
- Dequeue a state from the queue.
- Generate all possible next states by pouring water between jugs or filling/emptying them.
- Check if any of the next states match the target volume. If so, the problem is solved.
- Enqueue the valid next states into the queue.
- Repeat step 3 until the target volume is reached or all possible states are explored.

Using Prolog:

Code:

```
jug(0, 2, 4, 3, 2) :- 2 =:= 2,write('Goal state reached').
jug(X, Y, Vx, Vy, Z) :-
   Y =:= 0, Y1 is Vy, write('Step : X is '),write(X),write(' and Y is '), write(Y1),nl,jug(X, Y1, Vx,
Vy, Z).
jug(X, Y, Vx, Vy, Z) :-
   X =:= Vx, X1 is 0,write('Step : X is '),write(X1),write(' and Y is '), write(Y),nl, jug(X1, Y, Vx,
Vy, Z).
jug(X, Y, Vx, Vy, Z) :-
   Y =\= 0, X < Vx, K is min(Y, Vx - X),
   X1 is X + K, Y1 is Y - K,
   write('Step : X is '),write(X1),write(' and Y is '), write(Y1),nl,
```

```
    jug(X1, Y1, Vx, Vy, Z).
jug(X, Y, _, _, Z) :- X =:= Z; Y =:= Z,write('Goal state reached').
```

Output:

```
% c:/Users/slaye/OneDrive/Desktop/ai lab/jugproblem.pl
?-
|    jug(0,0,4, 3, 2).
Step : X is 0 and Y is 3
Step : X is 3 and Y is 0
Step : X is 3 and Y is 3
Step : X is 4 and Y is 2
Step : X is 0 and Y is 2
Goal state reached
true .

?-
```

Using Python:

Code:

```
from collections import deque

def BFS(a, b, target):
    m = {}
    isSolvable = False
    path = []

    q = deque()

    q.append((0, 0))

    while (len(q) > 0):
        u = q.popleft()
        if ((u[0], u[1]) in m):
            continue
        if ((u[0] > a or u[1] > b or
             u[0] < 0 or u[1] < 0)):
            continue

        path.append([u[0], u[1]])

        m[(u[0], u[1])] = 1

        if (u[0] == target or u[1] == target):
            isSolvable = True

            if (u[0] == target):
                if (u[1] != 0):
                    path.append([u[0], 0])
            else:
                if (u[0] != 0):
                    path.append([0, u[1]])

            sz = len(path)
            for i in range(sz):
                print("(", path[i][0], ",",
                      path[i][1], ")")
```

```
            break

        q.append([u[0], b])
        q.append([a, u[1]])

        for ap in range(max(a, b) + 1):

            c = u[0] + ap
            d = u[1] - ap

            if (c == a or (d == 0 and d >= 0)):
                q.append([c, d])

            c = u[0] - ap
            d = u[1] + ap

            if ((c == 0 and c >= 0) or d == b):
                q.append([c, d])

        q.append([a, 0])

        q.append([0, b])

    if (not isSolvable):
        print("No solution")


# Driver code
if __name__ == '__main__':
    Jug1, Jug2, target = 3, 5, 2
    print("Path from initial state to solution state ::")

    BFS(Jug1, Jug2, target)
```

Output:

```
============== RESTART: C:\Users\Shivam\
Path from initial state to solution state ::
( 0 , 0 )
( 0 , 5 )
( 3 , 0 )
( 3 , 5 )
( 3 , 2 )
( 0 , 2 )
>>>
```

PRACTICAL 3

Write a program to remove stop words for a given passage from a text file using NLTK.

THEORY:

To remove stop words from a given passage using NLTK (Natural Language Toolkit), you can follow these steps:

1. Tokenization: Tokenize the given passage into words or tokens. NLTK provides various tokenizers to achieve this task.

2. Stop Words Removal: Filter out the stop words from the tokenized passage. NLTK provides a list of commonly used stop words in different languages.

3. Reconstruct the Passage: Join the remaining words back together to reconstruct the passage without the stop words.

Here's the theory behind each step:

1. Tokenization:

Tokenization is the process of splitting a text into smaller units such as words or sentences.

NLTK provides various tokenizers such as word_tokenize() for word-level tokenization and sent_tokenize() for sentence-level tokenization.

2. Stop Words Removal:

Stop words are common words that do not carry significant meaning, such as "the", "is", "and", etc.

NLTK provides pre-defined lists of stop words for different languages, which can be used to filter out stop words from text.

3. Reconstruct the Passage:

After removing stop words, reconstruct the passage by joining the remaining words back together.

Optionally, you may perform additional text processing tasks such as stemming or lemmatization.

CODE :

```
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')


def read_text_file(file_path):
    with open(file_path, 'r') as file:
        return file.read()


def remove_stop_words(text):
    stop_words = set(stopwords.words('english'))
    words = nltk.word_tokenize(text)
    filtered_words = [word for word in words if word.lower() not in stop_words]
    return ' '.join(filtered_words)


file_path = "AI.txt"
passage = read_text_file(file_path)
passage_without_stopwords = remove_stop_words(passage)


print("Passage without stop words:") print(passage_without_stopwords)
```

Output

```
========== RESTART: C:\Users\shiva\Desktop\Practical\AI\11.py ==========
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\shiva\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
Passage without stop words:
Artificial Intelligence ( AI ) branch computer science aims create intelligent machines perform tasks typically require human intelligence . tasks include understanding natural
ognizing patterns , learning experience , making decisions .
```

PRACTICAL 4

Write a program to solve Monkey Banana problem using prolog

THEORY

The Monkey and Banana Problem is a classic problem in Artificial Intelligence that involves a monkey in a room trying to get a banana hanging from the ceiling, which is out of reach. The room also contains a movable box. The monkey can perform a series of actions such as walking, pushing the box, climbing the box, and grasping the banana.

The problem is to find a sequence of actions that will enable the monkey to get the banana, starting from an initial state and reaching a goal state. It is used to demonstrate goal-based planning, state representation, and reasoning about actions in AI, often implemented using logic programming languages like Prolog.

Steps:

Define the Initial State

Monkey at door, on floor

Box at window

Monkey does not have the banana

Define the Goal State

Monkey has the banana

List Possible Actions

walk(X, Y): Move from X to Y

push(X, Y): Push box from X to Y (if monkey is on floor and at same place as box)

climb: Climb onto box (if at same location)

grasp: Take banana (if on box under banana)

Represent States in Prolog

Use a structure like: state(MonkeyLocation, MonkeyStatus, BoxLocation, HasBanana)

Use Logical Rules to Model Action Effects

Define how each action changes the state

Search for a Sequence of Actions

Use Prolog's backtracking to find a valid path from the initial state to the goal state

Output the Action Sequence

Once the goal state is reached, print the list of actions taken

CODE:

```
% Represent the state as state(MonkeyPos,
BoxPos, MonkeyStatus, HasBanana)

% MonkeyStatus: onfloor | onbox

% Goal state: monkey has banana

goal(state(_, _, _, has)).

% Initial state: monkey and box at different
positions, monkey on floor, doesn't have banana

start(state(atdoor, atwindow, onfloor, hasnot)).

% Action: monkey walks to a place

move(state(MonkeyPos, BoxPos, onfloor,
HasBanana),

 walk(MonkeyPos, NewPos),

    state(NewPos, BoxPos, onfloor,
HasBanana)).
```

```
% Action: monkey pushes box to a new
location
move(state(Pos, Pos, onfloor, HasBanana),
    push(Pos, NewPos),
    state(NewPos, NewPos, onfloor,
HasBanana)).

% Action: monkey climbs the box
move(state(Pos, Pos, onfloor, HasBanana),
    climb,
    state(Pos, Pos, onbox, HasBanana)).

% Action: monkey grasps banana
move(state(middle, middle, onbox, hasnot),
    grasp,
    state(middle, middle, onbox, has)).

% Plan from initial state to goal
plan(State, [], _) :- goal(State).
```

```
plan(State, [Action | Rest], Visited) :-
    move(State, Action, NewState),
    \+ member(NewState, Visited),
    plan(NewState, Rest, [NewState | Visited]).

% Entry point to run the solution
solve :-
    start(StartState),
    plan(StartState, Plan, [StartState]),
    write('Plan to get the banana: '), nl,
    print_plan(Plan).

print_plan([]).
print_plan([Step | Rest]) :-
    write(' -> '), write(Step), nl,
    print_plan(Rest).
```

Output

Plan to get the banana:

-> walk(atdoor, atwindow)

-> push(atwindow, middle)

-> climb

-> grasp

PRACTICAL 5

Write a program to POS (part of speech) tagging for the give sentence using NLTK.

THEORY:

Part-of-speech (POS) tagging is the process of marking each word in a sentence with its corresponding part of speech, such as noun, verb, adjective, etc. NLTK (Natural Language Toolkit) provides tools and resources to perform POS tagging in Python.

Here's the theory behind implementing POS tagging for a given sentence using NLTK:

1. Tokenization:

Tokenize the given sentence into words or tokens. NLTK provides various tokenizers for this purpose.

2. POS Tagging:

Apply POS tagging to each token in the sentence to determine its part of speech. NLTK provides pre-trained models and taggers for POS tagging.

3. Result Interpretation:

Review the POS tags assigned to each word in the sentence to understand their grammatical roles.

CODE:

```
import nltk
from nltk.tokenize import word_tokenize
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
def pos_tagging(sentence):
    words = word_tokenize(sentence)
    pos_tags = nltk.pos_tag(words)
    return pos_tags
sentence = "I love running in the park and playing with my dog"
tagged_words = pos_tagging(sentence)
print("POS tagging for the sentence:")
print(tagged_words)
```

Output:

```
\AI\13.py ==========================================
POS tagging for the sentence:
[('I', 'PRP'), ('love', 'VBP'), ('running', 'VBG'), ('in', 'IN'), ('the', 'DT'), ('park', 'NN'), ('and', 'CC'), ('playing', 'NN'), ('with', 'IN'), ('my', 'PRP$'), ('dog', 'NN')]
```

PRACTICAL 6

Write a program to implement lemmatization using NLTK.

THEORY:

Lemmatization is the process of reducing words to their base or root form, typically by considering the word's context and part of speech. Unlike stemming, lemmatization ensures that the resulting word is a valid word in the language. NLTK (Natural Language Toolkit) provides tools and resources to perform lemmatization in Python.

Here's the theory behind implementing lemmatization using NLTK:

1. Tokenization:

Tokenize the given text into words or tokens. NLTK provides various tokenizers for this purpose.

2. Lemmatization:

Apply lemmatization to each token in the text to reduce it to its base form. NLTK provides WordNetLemmatizer for lemmatization.

3. POS Tagging (Optional):

Lemmatization often requires POS tagging to determine the correct part of speech of each word. NLTK's POS tagger can be used for this purpose.

4. Reconstruct the Text:

Join the lemmatized tokens back together to reconstruct the text with lemmatized words.

CODE:

```
import nltk

from nltk.tokenize import word_tokenize

from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

def get_wordnet_pos(word):

    tag = nltk.pos_tag([word])[0][1][0].upper()

    tag_dict = {"J": nltk.corpus.wordnet.ADJ, nltk.corpus.wordnet.NOUN,
nltk.corpus.wordnet.VERB, nltk.corpus.wordnet.ADV}

    return tag_dict.get(tag, nltk.corpus.wordnet.NOUN)

def lemmatize_sentence(sentence):

    words = word_tokenize(sentence)

    lemmatized_words = [lemmatizer.lemmatize(word, get_wordnet_pos(word)) for word in
words]

    lemmatized_sentence = ' '.join(lemmatized_words)

    return lemmatized_sentence

original_sentence = "She was running to catch the bus."

lemmatized_sentence = lemmatize_sentence(original_sentence)

print("Original:", original_sentence)

print("Lemmatized:", lemmatized_sentence)
```

Output:

```
========== REST
Original: She was running to catch the bus.
Lemmatized: She be run to catch the bus .
```

PRACTICAL 7

Write a program for Text Classification for the given sentence using NLTK.

THEORY:

Text classification is the process of categorizing text documents into predefined classes or categories based on their content. It is a fundamental task in natural language processing (NLP) and is used in various applications such as sentiment analysis, spam detection, topic classification, etc. NLTK (Natural Language Toolkit) provides tools and resources to perform text classification in Python.

Here's the theory behind implementing text classification for a given sentence using NLTK:

1. Preprocessing:

Tokenization: Tokenize the given text into words or tokens.

Stop Words Removal: Remove common stop words that do not carry significant meaning.

Lemmatization or Stemming: Reduce words to their base or root form to normalize the text.

2. Feature Extraction:

Convert the preprocessed text into numerical feature vectors that can be used as input to machine learning algorithms. This is typically done using techniques such as Bag-of-Words, TF-IDF (Term Frequency-Inverse Document Frequency), Word Embeddings, etc.

3. Model Training:

Select a suitable machine learning algorithm (e.g., Naive Bayes, Support Vector Machines, Logistic Regression, etc.).

Train the model using labeled data (i.e., text documents with known categories).

4. Model Evaluation:

Evaluate the trained model using metrics such as accuracy, precision, recall, F1-score, etc., on a separate test dataset.

5. Prediction

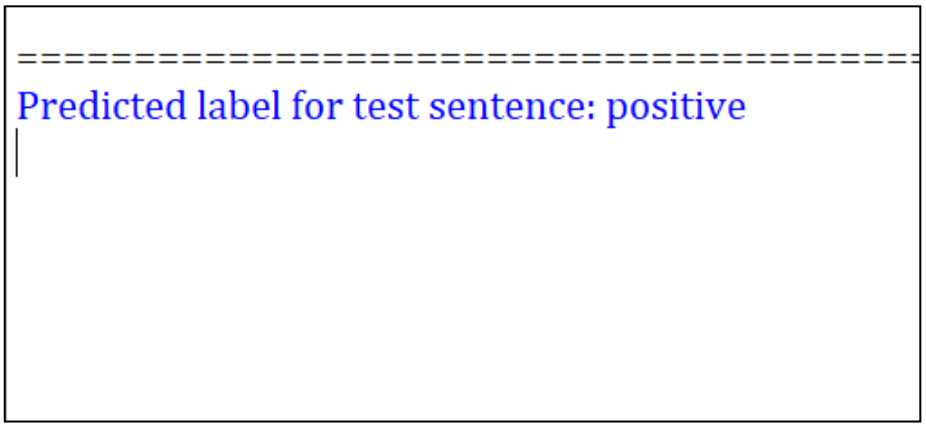
Use the trained model to predict the category of new or unseen text documents

CODE:

```
import nltk

from nltk.tokenize import word_tokenize

from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer

from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.naive_bayes import MultinomialNB


training_data = [

("The movie is good","positive"),("The movie is bad","Negative"),("I find this movie amazing","positive"),("I found this movie terrible","negative"), ("I love this movie", "positive"), ("This movie is great", "positive"), ("I dislike this movie", "negative"), ("This movie is terrible", "negative")]


def preprocess(sentence):

    lemmatizer = WordNetLemmatizer()

    stop_words = set(stopwords.words('english'))

    words = [lemmatizer.lemmatize(word.lower()) for word in word_tokenize(sentence) if word.isalpha() and word.lower() not in stop_words]

    return ' '.join(words)


X, y = zip(*[(preprocess(sentence), label) for sentence, label in training_data])


vectorizer = TfidfVectorizer()
```

```
X_vectorized = vectorizer.fit_transform(X)

classifier = MultinomialNB()
classifier.fit(X_vectorized, y)


test_sentence = "This movie is amazing!"
preprocessed_test_sentence = preprocess(test_sentence)
test_vectorized = vectorizer.transform([preprocessed_test_sentence])
predicted_label = classifier.predict(test_vectorized)[0]
print("Predicted label for test sentence:", predicted_label)
```

Output:



PRACTICAL 8

Program to demonstrate Simple Linear Regression

THEORY:

Simple Linear Regression is a supervised machine learning algorithm that models the relationship between two variables by fitting a straight line to the data.
It is used to predict the value of one variable (dependent) based on the value of another (independent).

Steps to Implement Simple Linear Regression

Import necessary libraries
Use libraries like NumPy, matplotlib, and scikit-learn.

Prepare the dataset
Organize the data into:

Independent variable (X) — input

Dependent variable (Y) — output

Reshape or convert the data (if needed)
Ensure input data (X) is in the correct format (usually 2D array for scikit-learn).

Create the regression model
Instantiate the LinearRegression() class from scikit-learn.

Train the model (fit the data)
Use .fit(X, y) to train the model and find the best-fit line.

Extract the model parameters

Slope (coefficient): model.coef_

Make predictions

Use .predict(X) to estimate Y values for given X inputs.

Visualize the results

Plot the actual data points and the regression line using a library like matplotlib.

Evaluate the model (optional but recommended)

Use metrics like:

R-squared (R^2) for goodness-of-fit

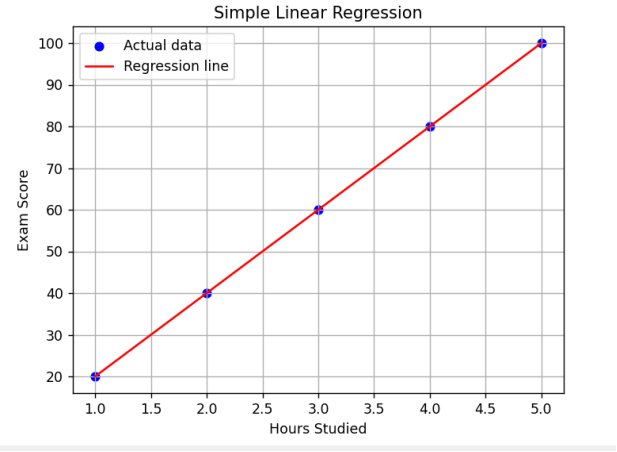
Mean Squared Error (MSE)

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression


# Sample data: Hours studied vs. Exam score
# X = independent variable (e.g., hours studied)
# y = dependent variable (e.g., exam score)
X = np.array([[1], [2], [3], [4], [5]])  # 2D array for sklearn
y = np.array([20, 40, 60, 80, 100])


# Create linear regression model
model = LinearRegression()


# Train the model
```

```
model.fit(X, y)


# Predict using the model
predicted = model.predict(X)


# Print model parameters
print("Slope (Coefficient):", model.coef_[0])
print("Intercept:", model.intercept_)


# Plot the results
plt.scatter(X, y, color='blue', label='Actual data')
plt.plot(X, predicted, color='red', label='Regression line')
plt.xlabel('Hours Studied')
plt.ylabel('Exam Score')
plt.title('Simple Linear Regression')
plt.legend()
plt.grid(True)
plt.show()
```

Output:


Simple Linear Regression
Actual data
Regression line
Exam Score
Hours Studied
100
90
80
70
60
50
40
30
20
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

PRACTICAL 9

Program to demonstrate K-Nearest Neighbor flowers classification

THEORY

K-Nearest Neighbor is a **supervised machine learning algorithm** used for classification and regression. It predicts the label of a new data point by looking at the **K closest training examples** (neighbors) in the feature space and assigning the most common label (for classification) or the average value (for regression) among those neighbors.

Steps of K-Nearest Neighbor Algorithm:

1. Choose the number K of nearest neighbors to consider.
2. Calculate the distance between the new data point and all training points (using Euclidean or other distance metrics).
3. Sort the distances and select the K nearest neighbors.
4. For classification: Count the class labels of the K neighbors and assign the most frequent label to the new data point.
5. For regression: Compute the average value of the K neighbors and assign it to the new data point.
6. Return the predicted label or value.

CODE

```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix

# Load the iris dataset
iris = load_iris()
X = iris.data  # features
y = iris.target  # labels

# Split the dataset into training and test sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Create and train the KNN model
k = 3  # number of neighbors
model = KNeighborsClassifier(n_neighbors=k)
model.fit(X_train, y_train)

# Predict the test set results
y_pred = model.predict(X_test)

# Evaluate the model
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=iris.target_names))

# Optional: Visualize decision boundaries using only first two features
def plot_decision_boundary():
```

```
    import numpy as np
    from matplotlib.colors import ListedColormap

    X_vis = X[:, :2]  # use only the first two features for 2D plotting
    X_train_vis, X_test_vis, y_train_vis, y_test_vis = train_test_split(X_vis, y, test_size=0.2,
random_state=42)

    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_vis, y_train_vis)

    h = .02  # step size in the mesh
    x_min, x_max = X_vis[:, 0].min() - 1, X_vis[:, 0].max() + 1
    y_min, y_max = X_vis[:, 1].min() - 1, X_vis[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
    cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, cmap=cmap_light)
    plt.scatter(X_vis[:, 0], X_vis[:, 1], c=y, cmap=cmap_bold, edgecolor='k', s=50)
    plt.xlabel(iris.feature_names[0])
    plt.ylabel(iris.feature_names[1])
```

```
# Uncomment to visualize
plot_decision_boundary()
```

Output

Confusion Matrix:

```
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
```

Classification Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	10
versicolor	1.00	1.00	1.00	9
virginica	1.00	1.00	1.00	11
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30

PRACTICAL 10

Program to demonstrate Naïve Bayes Classifier

THEORY

Naïve Bayes is a probabilistic supervised machine learning algorithm based on applying Bayes' Theorem with the assumption of feature independence. It predicts the class label for a given input by calculating the probability of each class given the features, then choosing the class with the highest probability.

Despite the "naïve" assumption that all features are independent, it often performs very well in practice, especially in text classification, spam filtering, and medical diagnosis.

Steps of Naïve Bayes Classifier

1. Prepare training data with labeled features and classes.
2. Calculate prior probabilities P(C)P(C)P(C) for each class from training data.
3. Calculate likelihoods P(xi∣ C)P(x_i | C)P(xi∣ C) for each feature xix_ixi given the class.
4. Apply Naïve Bayes assumption: Treat features as independent to compute joint likelihood:
5. P(X∣ C)=∏iP(xi∣ C)P(X | C) = \prod_{i} P(x_i | C)P(X∣ C)=i∏P(xi∣ C)
6. Compute posterior probabilities for each class using Bayes' Theorem.
7. Assign the class with the highest posterior probability to the input sample.

CODE

```
# Import necessary libraries
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
```

```
from sklearn.metrics import confusion_matrix, classification_report

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
target_names = iris.target_names

# Split into train and test sets (80/20)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and train the Naive Bayes classifier
model = GaussianNB()
model.fit(X_train, y_train)

# Predict the test set results
y_pred = model.predict(X_test)

# Evaluate the model
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=target_names))
```

OUTPUT

Confusion Matrix:

[[10 0 0]

[0 9 0]

[0 0 11]]

Classification Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	10
versicolor	1.00	1.00	1.00	9
virginica	1.00	1.00	1.00	11
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30

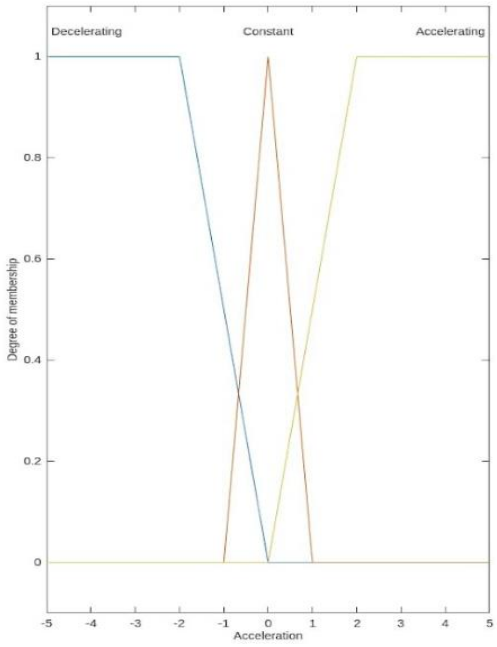
Practical 11

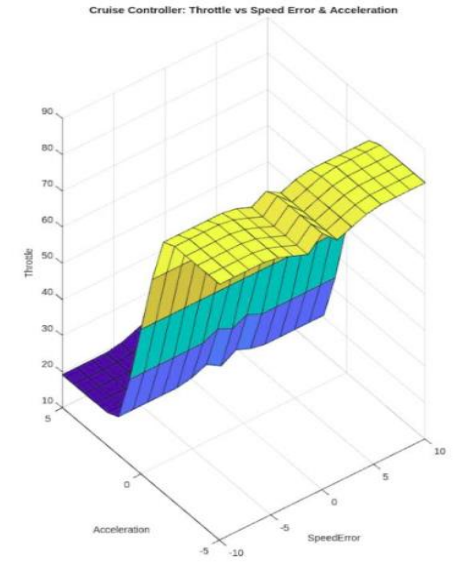
Greg Viot's Fuzzy Cruise Controller (New MATLAB Syntax)

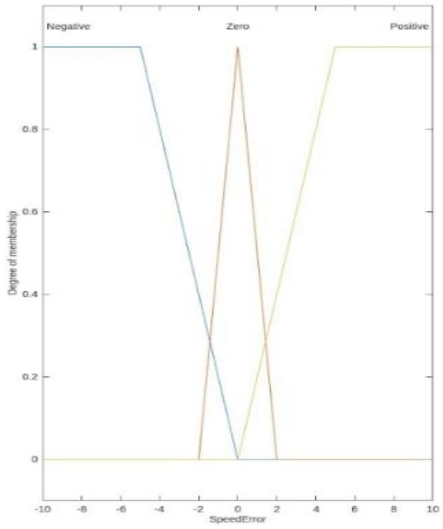
```
% Create Mamdani-type fuzzy inference system
fis = mamfis('Name','CruiseControl');
% Input 1: Speed Error
fis = addInput(fis,[-10 10],'Name','SpeedError');
fis = addMF(fis,'SpeedError','trapmf',[-10 -10 -5 0],'Name','Negative');
fis = addMF(fis,'SpeedError','trimf',[-2 0 2],'Name','Zero');
fis = addMF(fis,'SpeedError','trapmf',[0 5 10 10],'Name','Positive');
% Input 2: Acceleration
fis = addInput(fis,[-5 5],'Name','Acceleration');
fis = addMF(fis,'Acceleration','trapmf',[-5 -5 -2 0],'Name','Decelerating');
fis = addMF(fis,'Acceleration','trimf',[-1 0 1],'Name','Constant');
fis = addMF(fis,'Acceleration','trapmf',[0 2 5 5],'Name','Accelerating');
% Output: Throttle
fis = addOutput(fis,[0 100],'Name','Throttle');
fis = addMF(fis,'Throttle','trapmf',[0 0 25 50],'Name','Low');
fis = addMF(fis,'Throttle','trimf',[25 50 75],'Name','Medium');
fis = addMF(fis,'Throttle','trapmf',[50 75 100 100],'Name','High');
% Add fuzzy rules

ruleList = [ ...
    "SpeedError==Negative & Acceleration==Decelerating => Throttle=High", ...
    "SpeedError==Negative & Acceleration==Constant => Throttle=Medium", ...


    "SpeedError==Negative & Acceleration==Accelerating => Throttle=Low", ...
```

```
    "SpeedError==Zero & Acceleration==Decelerating => Throttle=High", ...
    "SpeedError==Zero & Acceleration==Constant => Throttle=Medium", ...
    "SpeedError==Zero & Acceleration==Accelerating => Throttle=Low", ...
    "SpeedError==Positive & Acceleration==Decelerating => Throttle=High", ...
    "SpeedError==Positive & Acceleration==Constant => Throttle=Medium", ...
    "SpeedError==Positive & Acceleration==Accelerating => Throttle=Low"];
fis = addRule(fis,ruleList);
% Show membership functions
figure; plotmf(fis,'input',1);
figure; plotmf(fis,'input',2);
% Test with sample input
out = evalfis(fis,[3 -2]); % Speed Error=3, Acceleration=-2
disp(['Throttle Output: ', num2str(out)]);
% Generate surface plot
figure;
gensurf(fis);
title('Cruise Controller: Throttle vs Speed Error & Acceleration');
```

OUTPUT


Decelerating
Constant
Accelerating
Degree of membership
Acceleration


Cruise Controller: Throttle vs Speed Error & Acceleration
Throttle
Acceleration
SpeedError


Negative
Zero
Positive
Degree of membership
SpeedError

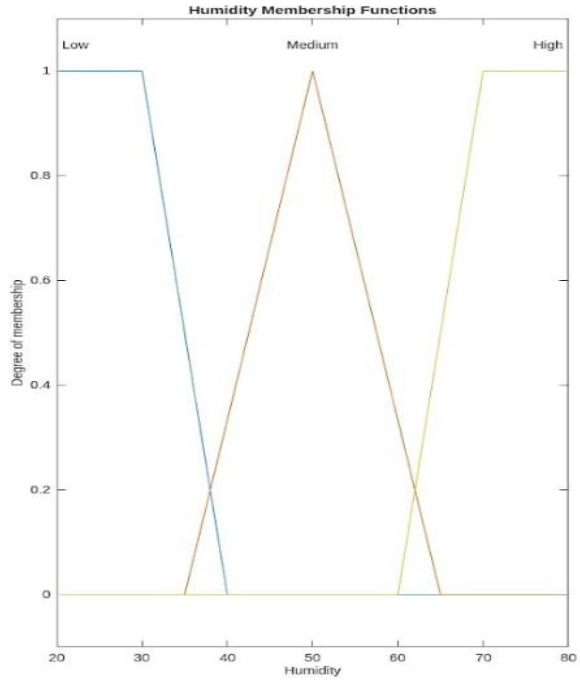
Practical 12

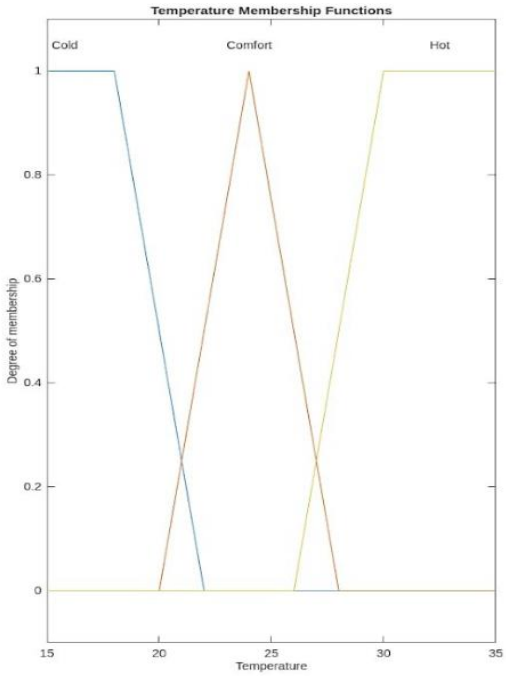
Fuzzy Air Conditioner Controller (Mamdani FIS)

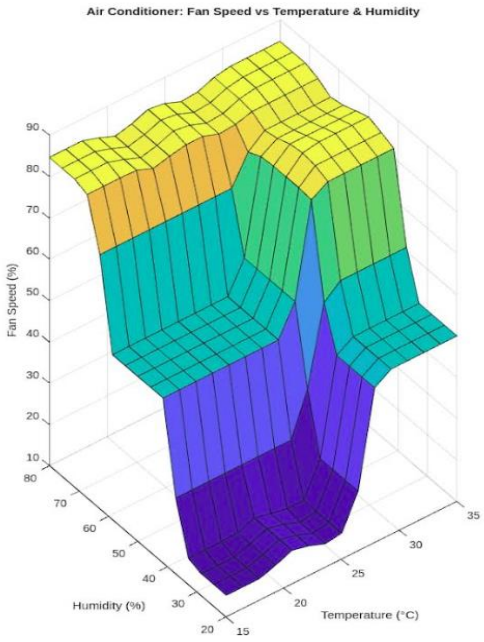
```
% Create Mamdani-type FIS
fis = mamfis('Name','AirConditioner');
% Input 1: Temperature (°C)
fis = addInput(fis,[15 35],'Name','Temperature');
fis = addMF(fis,'Temperature','trapmf',[15 15 18 22],'Name','Cold');
fis = addMF(fis,'Temperature','trimf',[20 24 28],'Name','Comfort');
fis = addMF(fis,'Temperature','trapmf',[26 30 35 35],'Name','Hot');
% Input 2: Humidity (%)
fis = addInput(fis,[20 80],'Name','Humidity');
fis = addMF(fis,'Humidity','trapmf',[20 20 30 40],'Name','Low');
fis = addMF(fis,'Humidity','trimf',[35 50 65],'Name','Medium');
fis = addMF(fis,'Humidity','trapmf',[60 70 80 80],'Name','High');
% Output: Fan Speed (%)
fis = addOutput(fis,[0 100],'Name','FanSpeed');
fis = addMF(fis,'FanSpeed','trapmf',[0 0 20 40],'Name','Low');
fis = addMF(fis,'FanSpeed','trimf',[30 50 70],'Name','Medium');
fis = addMF(fis,'FanSpeed','trapmf',[60 80 100 100],'Name','High');
% Add fuzzy rules
ruleList = [ ...
    "Temperature==Cold & Humidity==Low => FanSpeed=Low", ...


    "Temperature==Cold & Humidity==Medium => FanSpeed=Medium", ...
```

```
    "Temperature==Cold & Humidity==High => FanSpeed=High", ...
    "Temperature==Comfort & Humidity==Low => FanSpeed=Low", ...
    "Temperature==Comfort & Humidity==Medium => FanSpeed=Medium", ...
    "Temperature==Comfort & Humidity==High => FanSpeed=High", ...
    "Temperature==Hot & Humidity==Low => FanSpeed=Medium", ...
    "Temperature==Hot & Humidity==Medium => FanSpeed=High", ...
    "Temperature==Hot & Humidity==High => FanSpeed=High"];
fis = addRule(fis,ruleList);
% Show membership functions
figure; plotmf(fis,'input',1);
title('Temperature Membership Functions');
figure; plotmf(fis,'input',2);
title('Humidity Membership Functions');
% Test with sample input
out = evalfis(fis,[30 70]); % Example: Temp = 30°C, Humidity = 70%
disp(['Fan Speed: ', num2str(out)]);
% Generate surface plot
figure;
gensurf(fis);
title('Air Conditioner: Fan Speed vs Temperature & Humidity');
xlabel('Temperature (°C)');
ylabel('Humidity (%)');
zlabel('Fan Speed (%)');
```

OUTPUT


Humidity Membership Functions
Low
Medium
High
Degree of membership
Humidity


Temperature Membership Functions
Cold
Comfort
Hot
Degree of membership
Temperature


Air Conditioner: Fan Speed vs Temperature & Humidity
Fan Speed (%)
Humidity (%)
Temperature (°C)