Ex.No. 1 **Automatic Tic Tac Toe Game Using Random Number**
Date:

Aim:

To write a python program to implement automatic tic-tac-toe game using random number.

Algorithm:

Step 1: Start

Step 2: Create a 3x3 board and initialize it with 0

Step 3: For each player i.e. 1 or 2, choose a position on the board randomly and mark the location with the player's number

Step 4: Print the board after each move

Step 5: Evaluate the board after each move to check whether a row or column or diagonal has the same player number. If so display the winner's name

Step 6: Repeat steps 2 through 5 until aa winner emerges or all the nine positions are marked

Step 7: If there is no winner after all the nine moves, then display -1

Step 8: Stop

Program:

```
import numpy as np
import random
from time import sleep
 # Creates an empty board
  def create_board():
   return(np.array([[0, 0, 0],
                  [0, 0, 0],
                  [0, 0, 0]]))
 # Check for empty places on board
  def possibilities(board):
   l = [ ]
    for i in range(len(board)):
      for j in range(len(board)):
```

```
            if board[i][j] == 0:
                l.append((i, j))
    return(l)
# Select a random place for the player
def random_place(board, player):
    selection = possibilities(board)
    current_loc = random.choice(selection)
    board[current_loc] = player
    return(board)
# Checks whether the player has three of their marks in a horizontal row
def row_win(board, player):
    for x in range(len(board)):
        win = True
        for y in range(len(board)):
            if board[x, y] != player:
                win = False
                continue
        if win == True:
            return(win)
    return(win)
# Checks whether the player has three of their marks in a vertical row
def col_win(board, player):
    for x in range(len(board)):
        win = True
        for y in range(len(board)):
            if board[y][x] != player:
                win = False
                continue
        if win == True:
            return(win)
    return(win)
```

```
# Checks whether the player has three of their marks in a diagonal row
def diag_win(board, player):
    win = True
    y = 0
    for x in range(len(board)):
        if board[x, x] != player:
            win = False
    if win:
        return win
    win = True
    if win:
        for x in range(len(board)):
            y = len(board) - 1 - x
            if board[x, y] != player:
                win = False
    return win

# Evaluates whether there is a winner or a tie
def evaluate(board):
    winner = 0
    for player in [1, 2]:
        if (row_win(board, player) or
                col_win(board, player) or
                diag_win(board, player)):
            winner = player
    if np.all(board != 0) and winner == 0:
        winner = -1
    return winner

# Main function to start the game
```

```
def play_game():
  board, winner, counter = create_board(), 0, 1
  print(board)
  sleep(2)
  while winner == 0:
    for player in [1, 2]:
        board = random_place(board, player)
        print("Board after " + str(counter) + " move")
        print(board)
        sleep(2)
        counter += 1
        winner = evaluate(board)
        if winner != 0:
            break
  return(winner)

# Driver Code
print("Winner is: " + str(play_game()))
```

Output:

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move
[[0 0 0]
 [0 0 0]
 [1 0 0]]
Board after 2 move
[[0 0 0]
 [0 2 0]
 [1 0 0]]
```

Board after 3 move

[[0 1 0]

 [0 2 0]

 [1 0 0]]

Board after 4 move

[[0 1 0]

 [2 2 0]

 [1 0 0]]

Board after 5 move

[[1 1 0]

 [2 2 0]

 [1 0 0]]

Board after 6 move

[[1 1 0]

 [2 2 0]

 [1 2 0]]

Board after 7 move

[[1 1 0]

 [2 2 0]

 [1 2 1]]

Board after 8 move

[[1 1 0]

 [2 2 2]

 [1 2 1]]

Winner is: 2

Result:

Thus, the Python program to implement automatic tic-tac-toe game using random number was executed and the output was verified successfully.

Ex.No. 2 **Drug Screening**

Date:

Aim:

To write a python program to implement drug screening.

Algorithm:

Step 1: Start.

Step 2: Define the function 'drug_user.' The function takes the following parameters: probability threshold, sensitivity, specificity, prevalence, and verbose (default set to `True`).

Step 3: Calculate the probability of being a drug user (`p_user`) as the given `prevalence`.

Step 4: Calculate the probability of not being a drug user (`p_non_user`) as `1 - prevalence`.

Step 5: Calculate the probability of testing positive given the person is a drug user (`p_pos_user`) as the given `sensitivity`.

Step 6: Calculate the probability of testing negative given the person is a drug user (`p_neg_user`) as the given `specificity`.

Step 7: Calculate the probability of testing positive given the person is not a drug user (`p_pos_non_user`) as `1 - specificity`.

Step 8: Calculate the numerator (`num`) as `p_pos_user * p_user`.

Step 9: Calculate the denominator (`den`) as `p_pos_user * p_user + p_pos_non_user * p_non_user`.

Step 10: Calculate the probability (`prob`) as `num / den`.

Step 11: If `verbose` is `True`, print "The test-taker could be a user" if `prob` is greater than `prob_th`, otherwise print "The test-taker may not be a user".

Step 12: Return the probability `prob`. Call the `drug_user` function with the given parameters.

Step 13: Assign the returned probability to variable `p`.

Step 14: Print "Probability of the test taker being a drug user is" followed by the rounded value of `p` (rounded to 3 decimal places).

Step 15: Stop

Program:

```
def drug_user(
        prob_th=0.5,
        sensitivity=0.99,
        specificity=0.99,
        prevelance=0.01,
        verbose=True):
    """
    Computes the posterior using Bayes' rule
    """
    p_user = prevelance
    p_non_user = 1-prevelance
    p_pos_user = sensitivity
    p_neg_user = specificity
    p_pos_non_user = 1-specificity

    num = p_pos_user*p_user
    den = p_pos_user*p_user+p_pos_non_user*p_non_user

    prob = num/den

    if verbose:
        if prob > prob_th:
            print("The test-taker could be an user")
        else:
            print("The test-taker may not be an user")

    return prob

p = drug_user(prob_th=0.5, sensitivity=0.97, prevelance=0.005)
print("Probability of the test taker being a drug user is", round(p,3))
```

Output:

The test-taker may not be an user
Probability of the test taker being a drug user is 0.328

Result:

Thus, the Python program to implement drug screening was executed and the output was verified successfully.

Ex.No.3 **Monty Hall Problem**

Date:

Aim:

To write a python program to demonstrate the application of Bayesian Network for the Monty Hall Problem.

Algorithm:

Step 1: Start

Step 2: Initialize the Bayesian network:

Step 2.1: Create three nodes: Door A, Door B, and Door C. Assign prior probabilities to each node: Door A = 1/3, Door B = 1/3, Door C = 1/3.

Step2.2: Player's initial choice: The player selects one door (A, B, or C) as their initial choice.

Step 2.3: Monty opens a door: Generate a random number between 0 and 1. Suppose the player's initial choice is Door A.

Step 3: If the random number is less than the probability of Door A having a car (1/3), Monty opens a random door with a goat. If the random number is greater, Monty opens the other door with a goat.

Step 4: Update probabilities: Update the probabilities of the remaining doors based on the information from Monty opening a door. Use Bayes' theorem to calculate the posterior probabilities of the doors.

Step 5: Player's decision: The player decides whether to switch their choice or stick with their initial choice. If the player chooses to switch, they select the door with the highest posterior probability. If the player chooses to stick with their initial choice, they select the door they originally picked.

Step 6: Outcome: Determine whether the player wins or loses based on their final choice. If the chosen door has a car behind it, the player wins. Otherwise, they lose.

Step 7: Stop

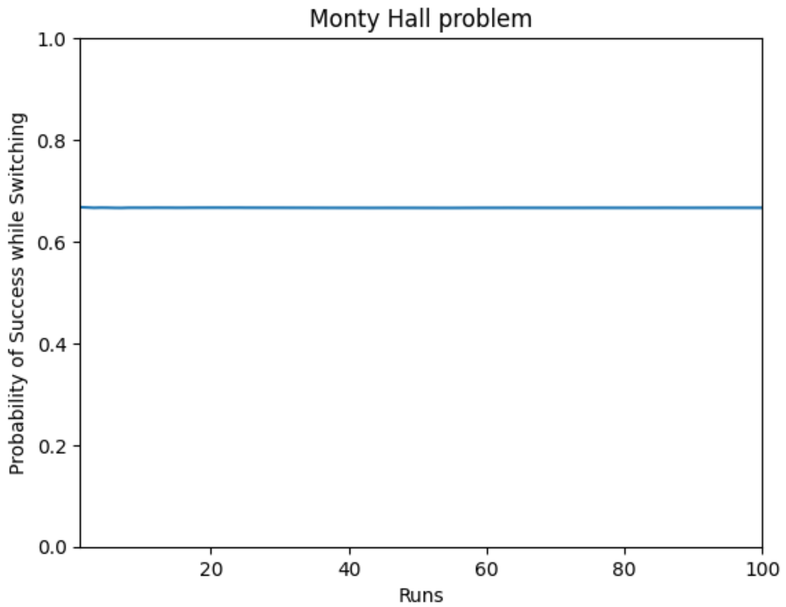
Program:

```
import random
import matplotlib.pyplot as plt

def switch_doors_experiment():
    # compute the correct door randomly
    correct_door = random.choice([1, 2, 3])
    # choose a door randomly
    door = random.choice([1, 2, 3])
    # Among two remaining door, get a random incorrect door
    doors = [1,2,3]
    try:
        doors.remove(door)
        doors.remove(correct_door)
    except:
        pass
    random_incorrect_door = random.choice(doors)

    # Remove the random incorrect door from the options available to you
    doors = [1, 2, 3]
    doors.remove(random_incorrect_door)

    # Now among your choice of door and the new set of options, switch your choice
    # Remove your original choice from the options
    doors.remove(door)
    # Now as only one option is there within
    final_choice = doors[0]

    # If the final choice is the correct door, then return 1, else return 0
    if final_choice == correct_door:
        return 1
```

```
    else:
        return 0

def probability_of_success_on_switch_door(precision):
    switch_door = 0
    # run the switch door experiment precision amount of times and increment the
    # outcome in switch_door counter
    for i in range(precision):
      switch_door = switch_door + switch_doors_experiment()

    # Probability of success while switching doors =
    #   num of times the experiment was successful / total number of runs
    return switch_door/precision

# Do 100 runs with precision 100000
runs = 100
total = 0
x = []
y = []
precision = 100000

for i in range(runs):
    total = total + probability_of_success_on_switch_door(precision)
    x.append(i+1)
    y.append(total/(i+1))

# Plot the probability vs runs on a matplotlib graph
plt.plot(x, y)
plt.xlabel('Runs')
plt.ylabel('Probability of Success while Switching')
```

```
plt.title('Monty Hall problem')

plt.ylim(0,1)
plt.xlim(1,runs)

plt.show()
print("Probability of Success on switching door for {} precision and {} runs is
{}".format(precision, runs, total/runs))
```

Output:




```
Probability of Success on switching door for 100000 precision and 100 runs is 0.6664808999999999
```

Probability of Success on switching door for 100000 precision and 100 runs is 0.6664808999999999

Result:

Thus, the Python program to demonstrate the application of Bayesian Network for the Monty Hall Problem was executed and the output was verified successfully.

Ex. No.4 **The Tipping Problem**
Date:

Aim:

To write a python program to create a fuzzy control system for modelling how to choose to tip at a restaurant.

Algorithm:

Step 1: Start.

Step 2: Import the required libraries such as numpy, skfuzzy, control.

Step 3: Create antecedents for each input variable using appropriate membership function.

Step 4: Create a consequent for the output variable using appropriate membership function.

Step 5: Create rules that map the combinations of fuzzy sets from input variables to the fuzzy sets of output variables.

Step 6: Define the fuzzy rules and view the fuzzy rules.

Step 7: Create the control system and simulation.

Step 8: Set the input values for the 'service' and 'quality' variables.

Step 9: Compute the fuzzy system. According to the input values ('poor', 'average', 'good') the output 'tip' value is printed.

Step 10: Stop.

Program:

pip install -U scikit-fuzzy

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# New Antecedent/Consequent objects hold universe variables and membership
# functions
```

```
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')
# Auto-membership function population is possible with .automf(3, 5, or 7)
quality.automf(3)
service.automf(3)

# Custom membership functions can be built interactively with a familiar, Pythonic API
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])

"""
To help understand what the membership looks like, use the ``view`` methods.
"""
quality['average'].view()
"""
.. image:: PLOT2RST.current_figure
"""
service.view()
"""
.. image:: PLOT2RST.current_figure
"""
tip.view()
"""
.. image:: PLOT2RST.current_figure

Fuzzy rules
-----------
```

Now, to make these triangles useful, we define the *fuzzy relationship* between input and output variables. For the purposes of our example, consider three simple rules:

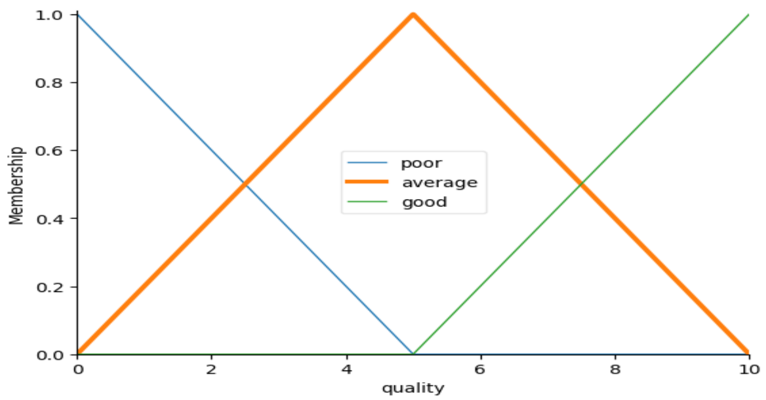
1. If the food is poor OR the service is poor, then the tip will be low
2. If the service is average, then the tip will be medium
3. If the food is good OR the service is good, then the tip will be high.

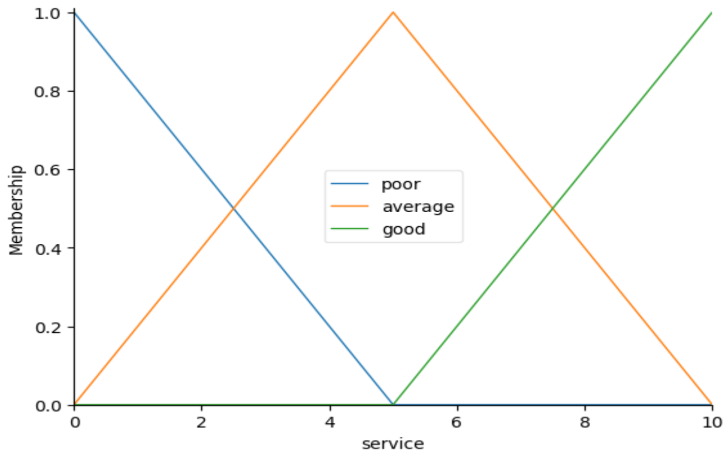
Most people would agree on these rules, but the rules are fuzzy. Mapping the imprecise rules into a defined, actionable tip is a challenge. This is the kind of task at which fuzzy logic excels.

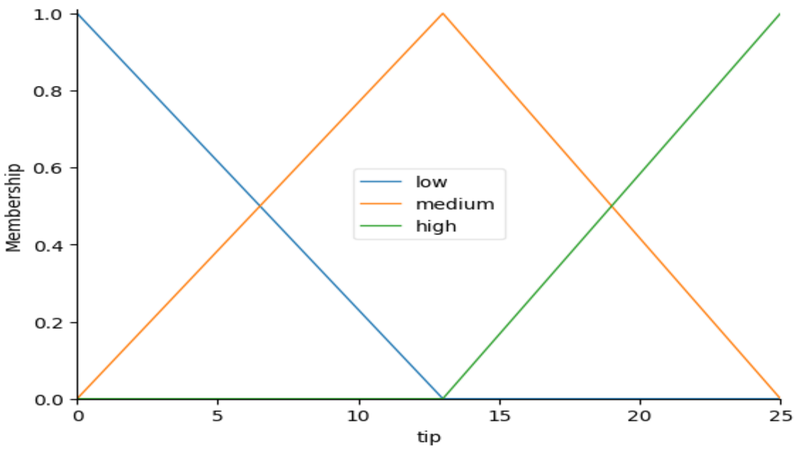
```
"""
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])
rule1.view()

"""
.. image:: PLOT2RST.current_figure
Control System Creation and Simulation
-------------------------------------
Now that we have our rules defined, we can simply create a control system via:
"""


tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])


tipping = ctrl.ControlSystemSimulation(tipping_ctrl)


tipping.input['quality'] = 6.5
tipping.input['service'] = 9.8
```

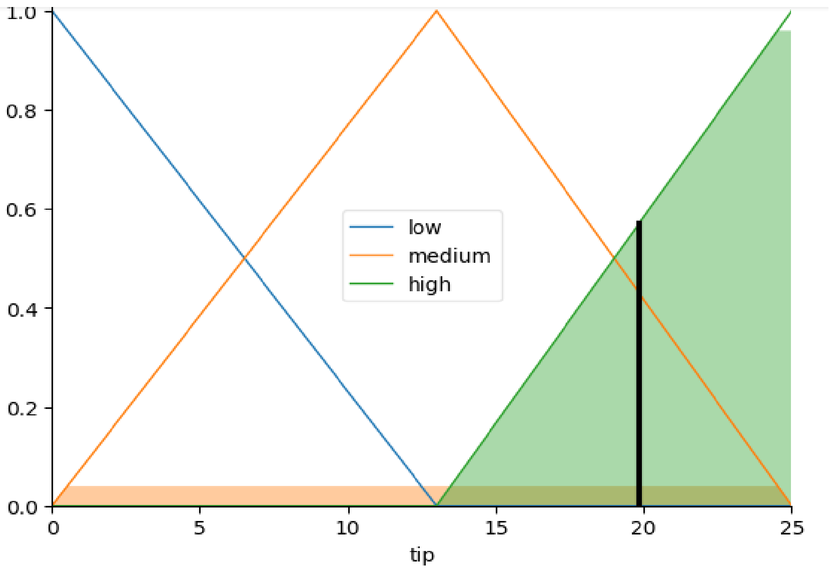
```
# Crunch the numbers
tipping.compute()

"""
Once computed, we can view the result as well as visualize it.
"""
print tipping.output['tip']
tip.view(sim=tipping)

"""
.. image:: PLOT2RST.current_figure

The resulting suggested tip is **20.24%**.

"""
```

Output:









Result:

Thus, the Python program to create a fuzzy control system for modelling how to choose to tip at a restaurant was executed and the output was verified successfully.

Ex. No.5 **Formulating a Decision Tree**

Date:

Aim:

To write a python program to formulate a decision tree to predict whether or not a patient has diabetes.

Algorithm:

Step 1: Start

Step 2: Explore the data

Step 3: Determine if it requires any cleaning and if there are any correlations in the data

Step 4: Apply the decision tree classification algorithm (using sklearn)

Step 5: Visualise the decision tree

Step 6: Evaluate the accuracy of the model

Step 7: Optimise the model to improve accuracy

Step 8: Predict the output.

Step 9: Stop

Program:

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
# Import Decision Tree Classifier
from sklearn.model_selection import train_test_split
# Import train_test_split function
from sklearn import metrics
#Import scikit-learn metrics module for accuracy calculation
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age', 'label']
# load dataset
pima = pd.read_csv("diabetes.csv", header=None, names=col_names)

feature_cols = ['pregnant', 'insulin', 'bmi', 'age', 'glucose', 'bp',' pedigree']
```

```
X = pima[feature_cols] # Features
y = pima.label # Target variable
# Split dataset into training set and test set 70% training and 30% testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
df = pd.read_csv('diabetes.csv')
x = df.drop('Outcome', axis=1)
y = df['Outcome']
X_train, X_test, Y_train, Y_test = train_test_split(x, y, test_size = 0.3)
model = DecisionTreeClassifier( )
model.fit(X_train, Y_train)
Y_pred = model.predict(X_test)
if model.predict([[1, 85, 66, 29, 0,26.6 ,0.351, 31]])[0] == 1:
  print("Having diabetes")
else:
  print("Not having diabetes")

pima.head()
```

Output:

Not having diabetes

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
1	6	148	72	35	0	33.6	0.627	50	1
2	1	85	66	29	0	26.6	0.351	31	0
3	8	183	64	0	0	23.3	0.672	32	1
4	1	89	66	23	94	28.1	0.167	21	0

Result:

Thus, the Python program to formulate a decision tree to predict whether or not a patient has diabetes was executed and the output was verified successfully.

Ex. No.6 **Fruit Classification Problem**
Date:

Aim:

To write a python program to implement adaptive boosting for a simple fruit classification problem.

Algorithm:

Step 1: Start.

Step 2: Import necessary libraries: pandas, NumPy, DecisionTreeClassifier, AdaBoostClassifier, cross_val_score, LabelEncoder.

Step 3: Load fruit classification dataset from a specified URL into a panda DataFrame.

Step 4: Preprocess the dataset by converting 'fruit_name' to numeric values using LabelEncoder.

Step 5: Split the dataset into features (X) and labels (y).

Step 6: Create a DecisionTreeClassifier instance for the decision tree classifier.

Step 7: Perform k-fold cross-validation using cross_val_score and calculate the mean accuracy score for the decision tree classifier.

Step 8: Create an AdaBoostClassifier instance using the decision tree classifier as the base estimator and 100 estimators.

Step 9: Perform k-fold cross-validation using cross_val_score and calculate the mean accuracy score for the AdaBoost classifier.

Step 10: Print the mean accuracy scores for both classifiers.

Step 11: Stop.

Program:

```
import pandas as pd

import numpy as np

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import AdaBoostClassifier

from sklearn.model_selection import cross_val_score, GridSearchCV

from sklearn.preprocessing import LabelEncoder

from sklearn import tree
```

```
# Load the fruit classification dataset from the URL
fruits = pd.read_csv('fruit data.csv')
# Explore the dataset
print("Dataset Summary:")
print(fruits.info())
print("\nFirst few rows of the dataset:")
print(fruits.head())

# Preprocessing: Convert fruit_name to numeric values
label_encoder = LabelEncoder()
fruits['fruit_label'] = label_encoder.fit_transform(fruits['fruit_name'])
# Split the dataset into features and labels
X = fruits[['mass', 'width', 'height', 'color_score']]
#y = fruits['fruit_label']
y = fruits['fruit_name']

# Create a decision tree classifier
decision_tree = DecisionTreeClassifier()

# Train the decision tree classifier using k-fold cross-validation
decision_tree_scores = cross_val_score(decision_tree, X, y, cv=5)

# Training classifier
classifier = tree.DecisionTreeClassifier()  # using decision tree classifier
classifier = classifier.fit(X, y)  # Find patterns in data

# Making predictions
print (classifier.predict([[210,9.4,6.3,0.6]]))
print("Decision Tree Accuracy (Cross-Validation):", np.mean(decision_tree_scores))

# Create an AdaBoost classifier using the decision tree as the base estimator
```

```
adaboost = AdaBoostClassifier(base_estimator=decision_tree, n_estimators=100)
# Train the best AdaBoost classifier using k-fold cross-validation
adaboost_scores = cross_val_score(adaboost, X, y, cv=5)

# Training classifier
classifier = tree.DecisionTreeClassifier()  # using decision tree classifier
classifier = classifier.fit(X, y)  # Find patterns in data

# Making predictions
print (classifier.predict([[210,9.4,6.3,0.6]]))
print("AdaBoost Accuracy (Cross-Validation):", np.mean(adaboost_scores))
```

Output:

```
Dataset Summary:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 59 entries, 0 to 58
Data columns (total 7 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   fruit_label    59 non-null     int64
 1   fruit_name     59 non-null     object
 2   fruit_subtype  59 non-null     object
 3   mass           59 non-null     int64
 4   width          59 non-null     float64
 5   height         59 non-null     float64
 6   color_score    59 non-null     float64
dtypes: float64(3), int64(2), object(2)
memory usage: 3.4+ KB
None
```

First few rows of the dataset:

	fruit_label	fruit_name	fruit_subtype	mass	width	height	color_score
0	1	apple	granny_smith	192	8.4	7.3	0.55
1	1	apple	granny_smith	180	8.0	6.8	0.59
2	1	apple	granny_smith	176	7.4	7.2	0.60
3	2	mandarin	mandarin	86	6.2	4.7	0.80
4	2	mandarin	mandarin	84	6.0	4.6	0.79

['apple']

Decision Tree Accuracy (Cross-Validation): 0.8651515151515151515

['apple']

AdaBoost Accuracy (Cross-Validation): 0.881818181818181818

Result:

Thus, the Python program to implement adaptive boosting for a simple fruit classification problem was executed and the output was verified successfully.

Ex. No.7 **Expectation Maximization Algorithm for Coin Toss Problem**
Date:

Aim:

To write a python program to implement expectation maximization algorithm for a coin toss problem.

Algorithm:

Step 1: Start.

Step 2: Guess the random initial estimates of θ_A and θ_B between 0 and 1.

Step 3: Use the likelihood function use the parameter values to see how probable the data is.

Step 4: Use this likelihood to generate a weighting for indicating the probability of each sequence produced using θA and θB. This is called the Expectation Step.

Step 5: Add up the total number of weighted counts for heads and tails across all sequences (call these counts H′ and T′) for both parameter estimates. Produce new estimates for θA and θB using the maximum likelihood formula H′ / H′+T′ (the Maximisation step).

Step 6: Repeat steps 3 to 5 until each parameter estimate has converged, or a set number of iterations has been reached. The total weight for each sequence should be normalised to 1.

Step 7: Stop.

Program:

```
import numpy as np


def coin_em(rolls, theta_A=None, theta_B=None, maxiter=10):
    # Initial Guess
    theta_A = theta_A or random.random()
    theta_B = theta_B or random.random()
    thetas = [(theta_A, theta_B)]
    # Iterate
    for c in range(maxiter):
        print("#%d:\t%0.2f %0.2f" % (c, theta_A, theta_B))
```

```
        heads_A, tails_A, heads_B, tails_B = e_step(rolls, theta_A, theta_B)
        theta_A, theta_B = m_step(heads_A, tails_A, heads_B, tails_B)

    thetas.append((theta_A, theta_B))
    return thetas, (theta_A, theta_B)

def e_step(rolls, theta_A, theta_B):
    """Produce the expected value for heads_A, tails_A, heads_B, tails_B
    over the rolls given the coin biases"""

    heads_A, tails_A = 0,0
    heads_B, tails_B = 0,0
    for trial in rolls:
        likelihood_A = coin_likelihood(trial, theta_A)
        likelihood_B = coin_likelihood(trial, theta_B)
        p_A = likelihood_A / (likelihood_A + likelihood_B)
        p_B = likelihood_B / (likelihood_A + likelihood_B)
        heads_A += p_A * trial.count("H")
        tails_A += p_A * trial.count("T")
        heads_B += p_B * trial.count("H")
        tails_B += p_B * trial.count("T")
    return heads_A, tails_A, heads_B, tails_B

def m_step(heads_A, tails_A, heads_B, tails_B):
    """Produce the values for theta that maximize the expected number of heads/tails"""
    # Replace dummy values with your implementation
    theta_A = heads_A / (heads_A + tails_A)
    theta_B = heads_B / (heads_B + tails_B)
    return theta_A, theta_B
```

```
def coin_likelihood(roll, bias):
    # P(X | Z, theta)
    numHeads = roll.count("H")
    flips = len(roll)
    return pow(bias, numHeads) * pow(1-bias, flips-numHeads)

rolls = [ "HTTTHHTHTH", "HHHHTHHHHH", "HTHHHHHTHH",
          "HTHTTTHHTT", "THHHTHHHTH" ]
thetas, _ = coin_em(rolls, 0.4, 0.3, maxiter=10)
```

Output:

```
#0:     0.40 0.30
#1:     0.69 0.55
#2:     0.74 0.56
#3:     0.77 0.55
#4:     0.78 0.53
#5:     0.79 0.53
#6:     0.79 0.52
#7:     0.80 0.52
#8:     0.80 0.52
#9:     0.80 0.52
```

Result:

Thus, the Python program to implement expectation maximization algorithm for a coin toss problem was executed and the output was verified successfully.

Ex. No.8 **Sentiment Analysis using Classifier**
Date:

Aim:

To write a python program to predict whether a given movie review is positive or negative.

Algorithm:

Step 1: Start

Step 2: Split your data into training and evaluation sets.

Step 3: Select a model architecture.

Step 4: Use training data to train your model.

Step 5: Use test data to evaluate the performance of your model.

Step 6: Use your trained model on new data to generate predictions,

Step 7: Stop

Program:

```
import pandas
d = pandas.read_csv("IMDB Dataset.csv", delimiter=",")
split = 0.7
d_train = d[:int(split*len(d))]
d_test = d[int((1-split)*len(d)):]
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
features = vectorizer.fit_transform(d_train.review)
test_features = vectorizer.transform(d_test.review)
i = 45000
j = 10
words = vectorizer.get_feature_names_out()[i:i+10]
pandas.DataFrame(features[j:j+7,i:i+10].todense(), columns=words)
```

```
from sklearn.naive_bayes import MultinomialNB
model1 = MultinomialNB()
model1.fit(features, d_train.sentiment)
pred1 = model1.predict_proba(test_features)

review = "I love this movie"
print (model1.predict(vectorizer.transform([review]))[0])
review = "This movie is bad"
print (model1.predict(vectorizer.transform([review]))[0])
review = "I was going to say something awesome, but I simply can't because the movie is so bad."
print (model1.predict(vectorizer.transform([review]))[0])
```

Output:

	legislators	legislature	legislatures	legit	legitamit	legitimacy	legitimate	legitimated	legitimately	legitimates
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0

positive

negative

negative

Result:

Thus, the Python program to predict whether a given movie review is positive or negative was executed and the output was verified successfully.

Ex. No.9 **Robot Movement**
Date:

Aim:

To write a python program to determine the movement of the robot in four directions.

Algorithm:

Step 1: Start

Stop 2: Import the math module.

Stop 3: Define a function called distance_moved() that takes the x-coordinate, y-coordinate, number of steps, and direction of the robot as input, and returns the distance moved by the robot.

Step 4: In the distance_moved() function, check the direction of the robot's movement and update the x-coordinate and y-coordinate accordingly.

Step 5: Use the math.sqrt() function to calculate the distance between the robot's original position and its current position.

Step 6: Return the distance moved by the robot.

Step 7: In the main function, prompt the user to enter the number of steps and the direction of the robot's movement.

Step 8: Call the distance_moved() function with the x-coordinate, y-coordinate, number of steps, and direction of the robot as input.

Step 9: Print the updated x- coordinate, y-coordinate and the distance moved by the robot.

Step 10: Stop

Program:

```
import math
def newposition(x, y, steps, direction):
  """Calculates the distance moved by a robot from its current position.
  Args:
    x: The x-coordinate of the robot's current position.
    y: The y-coordinate of the robot's current position.
    steps: The number of steps taken by the robot.
    direction: The direction of the robot's movement.
```

```
  Returns:
    The distance moved by the robot.
  """
        if direction == "UP":
            y += steps
        elif direction == "DOWN":
            y -= steps
        elif direction == "LEFT":
            x -= steps
        elif direction == "RIGHT":
            x += steps
        return(x,y)
if __name__ == "__main__":
        x = 0
        y = 0
        print("Position of x and y are")
        print(x,y)
        steps = int(input("Enter the number of steps: "))
        direction = input("Enter the direction (UP, DOWN, LEFT, RIGHT): ")
        print("New position of x and y are")
        position = newposition(x, y, steps, direction)
        print(position)
```

Output:

Position of x and y are 0 0

Enter the number of steps: 15

Enter the direction (UP, DOWN, LEFT, RIGHT): RIGHT

New position of x and y are

(15, 0)

Result:

Thus, the Python program to determine the movement of the robot in four directions was executed and the output was verified successfully.

Ex. No.10 **Robot Distance Travelled Problem**
Date:

Aim:

To write a python program to determine the distance moved by a robot in four directions.

Algorithm:

Step 1: Start

Stop 2: Import the math module.

Stop 3: Define a function called distance() that takes the x-coordinate, y-coordinate, number of steps, and direction of the robot as input, and returns the distance moved by the robot.

Step 4: In the distance() function, check the direction of the robot's movement and update the x-coordinate and y-coordinate accordingly.

Step 5: Use the math.sqrt() function to calculate the distance between the robot's original position and its current position.

Step 6: Return the distance moved by the robot.

Step 7: In the main function, prompt the user to enter the number of steps and the direction of the robot's movement.

Step 8: Call the distance() function with the x-coordinate, y-coordinate, number of steps, and direction of the robot as input.

Step 9: Print the updated x- coordinate, y-coordinate and the distance moved by the robot.

Step 10: Stop

Program:

```
import math
def distance(x, y, steps, direction):
  """Calculates the distance moved by a robot from its current position.
  Args:
    x: The x-coordinate of the robot's current position.
    y: The y-coordinate of the robot's current position.
    steps: The number of steps taken by the robot.
    direction: The direction of the robot's movement.
  Returns:
```

```
    The distance moved by the robot.
  """
        if direction == "UP":
                y += steps
        elif direction == "DOWN":
                y -= steps
        elif direction == "LEFT":
                x -= steps
        elif direction == "RIGHT":
                x += steps
        print(x,y)
        return math.sqrt(x**2 + y**2)
if __name__ == "__main__":
        x = 0
        y = 0
        print("Position of x and y are", x , y)
        steps = int(input("Enter the number of steps: "))
        direction = input("Enter the direction (UP, DOWN, LEFT, RIGHT): ")
        print("New position of x and y are")
        distancetravelled = distance(x, y, steps, direction)
        print("The distance moved is:", distancetravelled)
```

Output:

```
Position of x and y are 0 0
Enter the number of steps: 15
Enter the direction (UP, DOWN, LEFT, RIGHT): DOWN
New position of x and y are
0 -15
The distance moved is: 15.0
```

Result:

Thus, the Python program to determine the distance moved by a robot in four directions was executed and the output was verified successfully.