Name: Dheeraj Singh Karki

Student Id.: 4748118

Class: MSc.AI Part – 1

SUB.: Decision Modeling

Batch: MSc AI	**Roll No.: 4748118**
Experiment 01	

Title: Simulate the Markov Decision Process (MDP)

Objective:
Student Needs to Use the MDP toolbox
Get familiarized with all the modules/ Functions present in it.
Apply the MDP for their problem statement

Books/ Journals/ Websites referred:

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- https://pymdptoolbox.readthedocs.io/en/latest/api/mdptoolbox.html

Resources used: Markov Decision Process (MDP) Toolbox for python

Theory:
The MDP toolbox is a powerful tool for modeling and solving Markov Decision Processes. By providing a range of classes and functions, it simplifies the process of implementing and experimenting with various MDP algorithms. This makes it an excellent resource for researchers, students, and practitioners working in fields that require decision-making under uncertainty.

Example:
Transition Matrix (P):
A transition matrix PP is a 3-dimensional array where P[a][s][s′]P[a][s][s′] represents the probability of transitioning from state ss to state s′s′ when action aa is taken. The sum of probabilities for each action-state pair must equal 1.
Example:
Consider an MDP with 3 states (S={0,1,2}S={0,1,2}) and 2 actions (A={0,1}A={0,1}).
For action 00:
From state 00, there's a 50% chance of staying in state 00 and a 50% chance of moving to state 11.

From state 11, there's a 70% chance of moving to state 22 and a 30% chance of staying in state 11.

From state 22, there's a 100% chance of staying in state 22.

For action 11:

From state 00, there's a 80% chance of moving to state 11 and a 20% chance of moving to state 22.

From state 11, there's a 60% chance of moving to state 00 and a 40% chance of moving to state 22.

From state 22, there's a 100% chance of staying in state 22.

Reward Matrix (R):

A reward matrix RR is a 2-dimensional array where R[a][s]R[a][s] represents the reward received when taking action aa in state ss.

Example:

For simplicity, let's assume:

Taking action 00 in state 00 yields a reward of 5.

Taking action 00 in state 11 yields a reward of 10.

Taking action 00 in state 22 yields a reward of 0.

Taking action 11 in state 00 yields a reward of 2.

Taking action 11 in state 11 yields a reward of 8.

Taking action 11 in state 22 yields a reward of 1.

MDP Algorithms:

Several algorithms are commonly used to solve MDPs, each with its own strengths and use cases. Here are some key algorithms:

1. Value Iteration:

Value Iteration iteratively updates the value function for each state until it converges to the optimal value function. It then derives the optimal policy from the optimal value function.

2. Policy Iteration:

Policy Iteration alternates between policy evaluation (calculating the value of a policy) and policy improvement (finding a better policy based on the current value function) until the policy converges to the optimal policy.

3. Q-Learning:

Q-Learning is a model-free reinforcement learning algorithm that learns the value of actions directly without requiring a model of the environment.

Utility:

Utility functions are useful for validating and working with MDPs. Here are a few examples:

1. Validate Transition Matrix:
Ensures each row in the transition matrix sums to 1, representing valid probability distributions.

2. Validate Reward Matrix:
Checks the dimensions of the reward matrix to ensure it matches the transition matrix.

3. Generate Random MDP:
Generates a random MDP for testing purposes.

Problem Consideration:

1. State Space: Ensure the state space is well-defined and manageable in size.
2. Action Space: Define all possible actions and ensure they are applicable in all states.
3. Transition Dynamics: Clearly specify the transition probabilities between states for each action.
4. Rewards: Define immediate rewards for each state-action pair.
5. Discount Factor: Choose an appropriate discount factor to balance immediate and future rewards.
6. Algorithm Selection: Choose an appropriate algorithm based on the problem size and requirements (e.g., Value Iteration for small to medium-sized problems, Q-Learning for model-free scenarios).

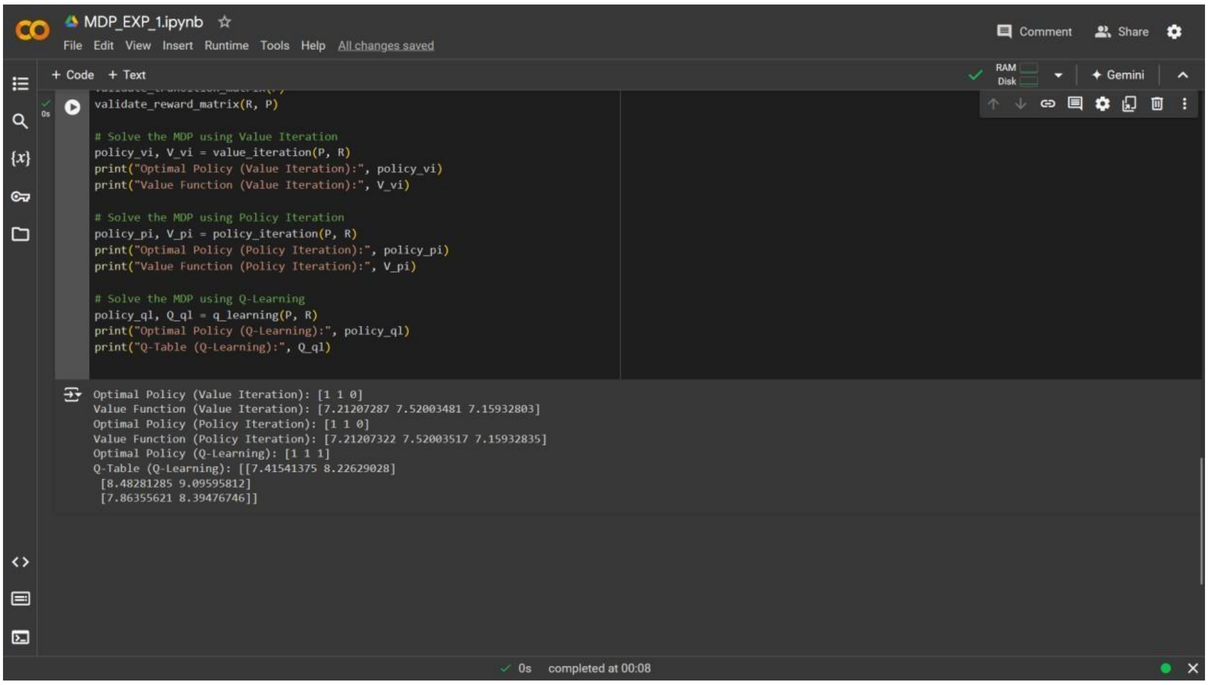
Implementation Code:

```
import numpy as np
import random

### MDP Algorithms ###

def value_iteration(P, R, gamma=0.9, epsilon=1e-6):
    n_states, n_actions = R.shape[1], R.shape[0]
    V = np.zeros(n_states)
    while True:
        delta = 0
        for s in range(n_states):
            v = V[s]
            V[s] = max(sum(P[a, s, s1] * (R[a, s] + gamma * V[s1]) for
s1 in range(n_states)) for a in range(n_actions))
            delta = max(delta, abs(v - V[s]))
        if delta < epsilon:
            break
```

```
    policy = np.argmax([[sum(P[a, s, s1] * (R[a, s] + gamma * V[s1]) for s1 in range(n_states)) for a in range(n_actions)] for s in range(n_states)], axis=1)
    return policy, V

def policy_iteration(P, R, gamma=0.9, epsilon=1e-6):
    n_states, n_actions = R.shape[1], R.shape[0]
    policy = np.zeros(n_states, dtype=int)
    V = np.zeros(n_states)
    while True:
        while True:
            delta = 0
            for s in range(n_states):
                v = V[s]
                V[s] = sum(P[policy[s], s, s1] * (R[policy[s], s] + gamma * V[s1]) for s1 in range(n_states))
                delta = max(delta, abs(v - V[s]))
            if delta < epsilon:
                break
        policy_stable = True
        for s in range(n_states):
            old_action = policy[s]
            policy[s] = np.argmax([sum(P[a, s, s1] * (R[a, s] + gamma * V[s1]) for s1 in range(n_states)) for a in range(n_actions)])
            if old_action != policy[s]:
                policy_stable = False
        if policy_stable:
            break
    return policy, V

def q_learning(P, R, gamma=0.9, alpha=0.1, epsilon=0.1, episodes=1000):
    n_states, n_actions = R.shape[1], R.shape[0]
    Q = np.zeros((n_states, n_actions))
    for _ in range(episodes):
        state = random.choice(range(n_states))
        while True:
            if random.uniform(0, 1) < epsilon:
                action = random.choice(range(n_actions))
            else:
                action = np.argmax(Q[state])
            next_state = np.argmax(P[action, state])
            reward = R[action, state]
            best_next_action = np.argmax(Q[next_state])
            td_target = reward + gamma * Q[next_state, best_next_action]
            td_error = td_target - Q[state, action]
            Q[state, action] += alpha * td_error
            if state == next_state:
                break
            state = next_state
    policy = np.argmax(Q, axis=1)
    return policy, Q

### Utility Functions ###

def validate_transition_matrix(P):
    assert np.allclose(P.sum(axis=2), 1), "Transition probabilities must sum to 1."
```

```
def validate_reward_matrix(R, P):
    assert R.shape == P.shape[:2], "Reward matrix dimensions must
match the transition matrix."

def generate_random_mdp(n_states, n_actions):
    P = np.zeros((n_actions, n_states, n_states))
    for a in range(n_actions):
        for s in range(n_states):
            P[a, s, :] = np.random.dirichlet(np.ones(n_states))
    R = np.random.rand(n_actions, n_states)
    return P, R

### Example Usage ###

# Generate a random MDP
n_states = 3
n_actions = 2
P, R = generate_random_mdp(n_states, n_actions)

# Validate the MDP
validate_transition_matrix(P)
validate_reward_matrix(R, P)

# Solve the MDP using Value Iteration
policy_vi, V_vi = value_iteration(P, R)
print("Optimal Policy (Value Iteration):", policy_vi)
print("Value Function (Value Iteration):", V_vi)

# Solve the MDP using Policy Iteration
policy_pi, V_pi = policy_iteration(P, R)
print("Optimal Policy (Policy Iteration):", policy_pi)
print("Value Function (Policy Iteration):", V_pi)

# Solve the MDP using Q-Learning
policy_ql, Q_ql = q_learning(P, R)
print("Optimal Policy (Q-Learning):", policy_ql)
print("Q-Table (Q-Learning):", Q_ql)
```

Output Screenshot:



Conclusion (Students should write understanding of MDP):

A Markov Decision Process (MDP) is a mathematical framework used to describe an environment in decision-making scenarios where outcomes are partly random and partly under the control of a decision maker. MDPs are characterized by the following components:

1. **States (S)**: A finite set of states representing the different situations or configurations in which an agent can find itself.
2. **Actions (A)**: A finite set of actions available to the agent from each state.
3. **Transition Probabilities (P)**: A probability distribution P(s′|s,a)P(s′|s,a) that defines the probability of transitioning to state s′s′ from state ss by taking action aa.
4. **Rewards (R)**: A reward function R(s,a)R(s,a) that specifies the immediate reward received after transitioning from state ss to state s′s′ by taking action aa.
5. **Discount Factor (γγ)**: A factor between 0 and 1 that represents the importance of future rewards. A higher γγ values future rewards more heavily.

The goal in MDPs is to find a policy (a mapping from states to actions) that maximizes the expected sum of rewards over time, also known as the value function.

Application (MDP):

MDPs are widely used in various fields due to their versatility in modeling decision-making processes. Some key applications include:

1. **Robotics**: MDPs are used to model the behavior of robots in uncertain environments, allowing for the development of control policies that enable robots to navigate, manipulate objects, and perform tasks autonomously.
2. **Finance**: In financial decision-making, MDPs help in portfolio optimization, option pricing, and managing investment strategies by modeling the stochastic nature of market conditions and returns.
3. **Operations Research**: MDPs are applied to optimize resource allocation, supply chain management, inventory control, and scheduling problems in industries to minimize costs and maximize efficiency.
4. **Healthcare**: MDPs are utilized to model patient treatment plans, optimize the allocation of medical resources, and improve decision-making in clinical trials and medical diagnosis.
5. **Artificial Intelligence (AI) and Machine Learning**: In AI, MDPs are the foundation for reinforcement learning, where agents learn to make decisions by interacting with the environment to maximize cumulative rewards.
6. **Telecommunications**: MDPs help in network routing, managing communication channels, and optimizing bandwidth allocation to ensure efficient and reliable data transmission.
7. **Game Theory and Economics**: MDPs model strategic interactions in competitive environments, helping in the design of optimal strategies in games and economic systems.

Batch: MSc AI	**Roll No.: 4748118**
Experiment 02	

Title: Implement the Monte Carlo Method

Objective:

Student needs to understand the Concept of Monte Carlo method

Implement the Monte Carlo Method for

Books/ Journals/ Websites referred:

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- https://bookdown.org/s3dabeck1984/bookdown-demo-master/monte-carlo-simulations.html
- https://pbpython.com/monte-carlo.html
- https://www.analyticsvidhya.com/blog/2021/04/how-to-perform-monte-carlo-simulation/

Resources used: **R or Python**

Theory:

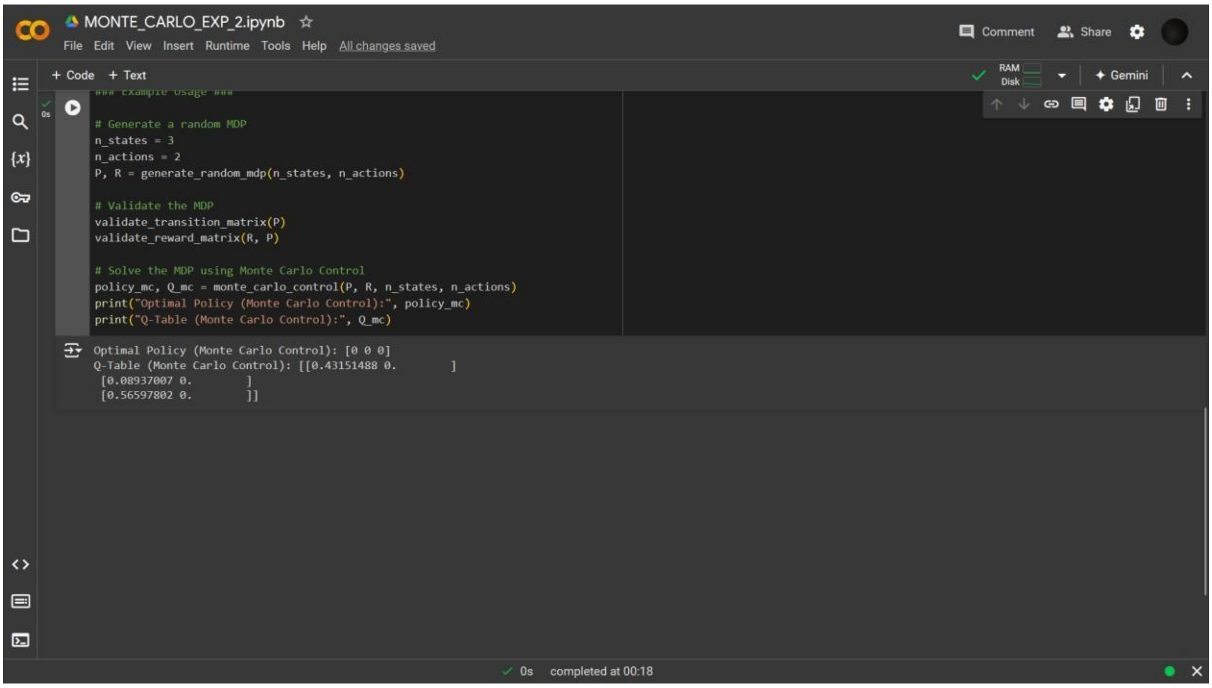
The Monte Carlo method is a statistical technique used for estimating numerical results by simulating random sampling. In the context of Markov Decision Processes (MDPs), Monte Carlo methods are used to estimate the value function or the optimal policy by averaging sample returns obtained from simulated episodes.

Implementation (Code):

```
import numpy as np
import random

def generate_episode(P, R, policy, n_states, max_steps=100):
    episode = []
    state = random.choice(range(n_states))
    for _ in range(max_steps):
        action = policy[state]
        next_state = np.random.choice(range(n_states), p=P[action,
state])
        reward = R[action, state]
        episode.append((state, action, reward))
        state = next_state
        if state == next_state:  # Assuming episode ends when reaching
a terminal state
            break
    return episode

def monte_carlo_control(P, R, n_states, n_actions, gamma=0.9,
epsilon=0.1, episodes=1000):
    Q = np.zeros((n_states, n_actions))
    returns = { (s, a): [] for s in range(n_states) for a in
range(n_actions) }
    policy = np.zeros(n_states, dtype=int)

    for _ in range(episodes):
        episode = generate_episode(P, R, policy, n_states)
        G = 0
        for t in reversed(range(len(episode))):
            state, action, reward = episode[t]
            G = gamma * G + reward
            if not any((state == x[0] and action == x[1]) for x in
episode[:t]):
                returns[(state, action)].append(G)
                Q[state, action] = np.mean(returns[(state, action)])
                policy[state] = np.argmax(Q[state])

    return policy, Q

### Utility Functions ###

def validate_transition_matrix(P):
    assert np.allclose(P.sum(axis=2), 1), "Transition probabilities
must sum to 1."

def validate_reward_matrix(R, P):
    assert R.shape == P.shape[:2], "Reward matrix dimensions must
match the transition matrix."

def generate_random_mdp(n_states, n_actions):
    P = np.zeros((n_actions, n_states, n_states))
    for a in range(n_actions):
        for s in range(n_states):
            P[a, s, :] = np.random.dirichlet(np.ones(n_states))
    R = np.random.rand(n_actions, n_states)
    return P, R
```

```
### Example Usage ###

# Generate a random MDP
n_states = 3
n_actions = 2
P, R = generate_random_mdp(n_states, n_actions)

# Validate the MDP
validate_transition_matrix(P)
validate_reward_matrix(R, P)

# Solve the MDP using Monte Carlo Control
policy_mc, Q_mc = monte_carlo_control(P, R, n_states, n_actions)
print("Optimal Policy (Monte Carlo Control):", policy_mc)
print("Q-Table (Monte Carlo Control):", Q_mc)
```

Output Screenshots with explanation:



Conclusion (Students should write in their own words):

Monte Carlo methods provide a powerful approach for estimating numerical results by simulating random sampling. In the context of Markov Decision Processes (MDPs), we implemented the Monte Carlo control method to estimate the optimal policy. By generating episodes using the current policy and averaging returns obtained from these episodes, we iteratively improve the policy until convergence. This method offers a simple and effective way to find optimal policies in MDPs without requiring a model of the environment.

Applications:

1. **Finance**: Monte Carlo simulations are used in option pricing, portfolio optimization, and risk management.
2. **Physics**: Monte Carlo methods are applied in simulating particle interactions, modeling physical systems, and solving complex problems in quantum mechanics.
3. **Engineering**: Monte Carlo simulations are used in reliability analysis, system design, and optimization of engineering systems.
4. **Computer Graphics**: Monte Carlo methods are used in rendering realistic images, simulating light transport, and generating procedural content.
5. **Biomedical Sciences**: Monte Carlo simulations are used in medical imaging, drug discovery, and epidemiological studies.

Batch: MSc AI **Roll No.: 4748118**

Experiment 03

Title: Write a program to implement Q-Learning algorithm

Books/ Journals/ Websites referred:

- https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/
- https://www.analyticsvidhya.com/blog/2021/04/q-learning-algorithm-with-step-by-step-implementation-using-python/

Theory:

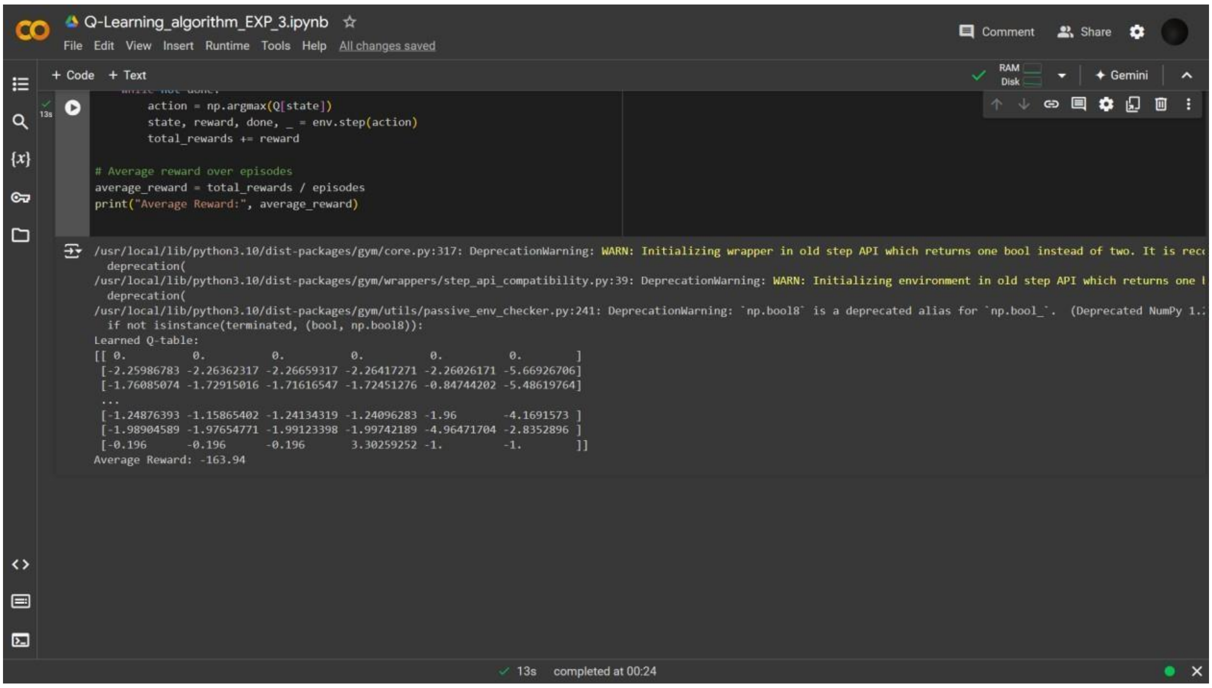
Q-Learning is a model-free reinforcement learning algorithm used to learn the optimal policy for making decisions in an environment. It belongs to the class of temporal difference learning methods, where the agent learns by interacting with the environment and receiving rewards.

Implementation (Code):

```
import numpy as np
import gym

# Create the environment
env = gym.make('Taxi-v3')

# Initialize Q-table with zeros
Q = np.zeros([env.observation_space.n, env.action_space.n])

# Set hyperparameters
alpha = 0.1  # Learning rate
gamma = 0.6 # Discount factor
epsilon = 0.1  # Exploration rate

# Number of episodes
episodes = 1000

# Q-Learning algorithm
for _ in range(episodes):
    state = env.reset()
    done = False
    while not done:
        # Epsilon-greedy policy
        if np.random.uniform(0, 1) < epsilon:
            action = env.action_space.sample() # Exploration
        else:
            action = np.argmax(Q[state])  # Exploitation
```

```
        next_state, reward, done, _ = env.step(action)

        # Q-value update
        old_q_value = Q[state, action]
        next_max = np.max(Q[next_state])
        new_q_value = (1 - alpha) * old_q_value + alpha * (reward + 
gamma * next_max)
        Q[state, action] = new_q_value

        state = next_state

# Print the learned Q-table
print("Learned Q-table:")
print(Q)

# Evaluate the learned policy
total_rewards = 0
episodes = 100
for _ in range(episodes):
    state = env.reset()
    done = False
    while not done:
        action = np.argmax(Q[state])
        state, reward, done, _ = env.step(action)
        total_rewards += reward

# Average reward over episodes
average_reward = total_rewards / episodes
print("Average Reward:", average_reward)
```

<u>Output Screenshots:</u>



Conclusion (Students should write in their own words):

In this implementation, we applied the Q-Learning algorithm to solve the Taxi-v3 environment from the OpenAI Gym. The agent learns an optimal policy by interacting with the environment, updating Q-values based on observed rewards and transitions. The learned Q-table represents the expected rewards for taking actions in different states. The evaluation of the learned policy demonstrates its effectiveness in achieving high cumulative rewards.

Applications:

1. **Game Playing**: Q-Learning can be used to develop AI agents that play games optimally by learning from past experiences and maximizing rewards.
2. **Robotics**: Q-Learning enables robots to learn optimal strategies for navigation, manipulation, and task completion in complex environments.
3. **Resource Management**: Q-Learning can be applied in dynamic resource allocation problems, such as traffic management, energy distribution, and inventory control.
4. **Autonomous Vehicles**: Q-Learning algorithms help autonomous vehicles make decisions in real-time, such as route planning, obstacle avoidance, and traffic prediction.
5. **Finance**: Q-Learning can be used in algorithmic trading, portfolio optimization, and risk management to make data-driven decisions and maximize returns.

Batch: MSc AI **Roll No.: 4748118**

Experiment 04

Title: Write a program to implement approximate value iteration (AVI) algorithm and API (Approximate policy iteration)

Books/ Journals/ Websites referred:

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- https://gym.openai.com/docs/
- https://www.analyticsvidhya.com/blog/2018/09/reinforcement-learning-model-based-planning-dynamic-programming/

Theory:

Approximate Value Iteration (AVI) and Approximate Policy Iteration (API) are two reinforcement learning algorithms used to solve Markov Decision Processes (MDPs) with large state spaces. These algorithms employ function approximation techniques to represent value functions or policies, allowing for scalability to complex environments.

In AVI, the algorithm iteratively updates the parameters of a function approximator to estimate the optimal value function. This approximation enables AVI to handle large state spaces efficiently. On the other hand, API alternates between policy evaluation and policy improvement steps, using function approximation to represent the policy and the value function.

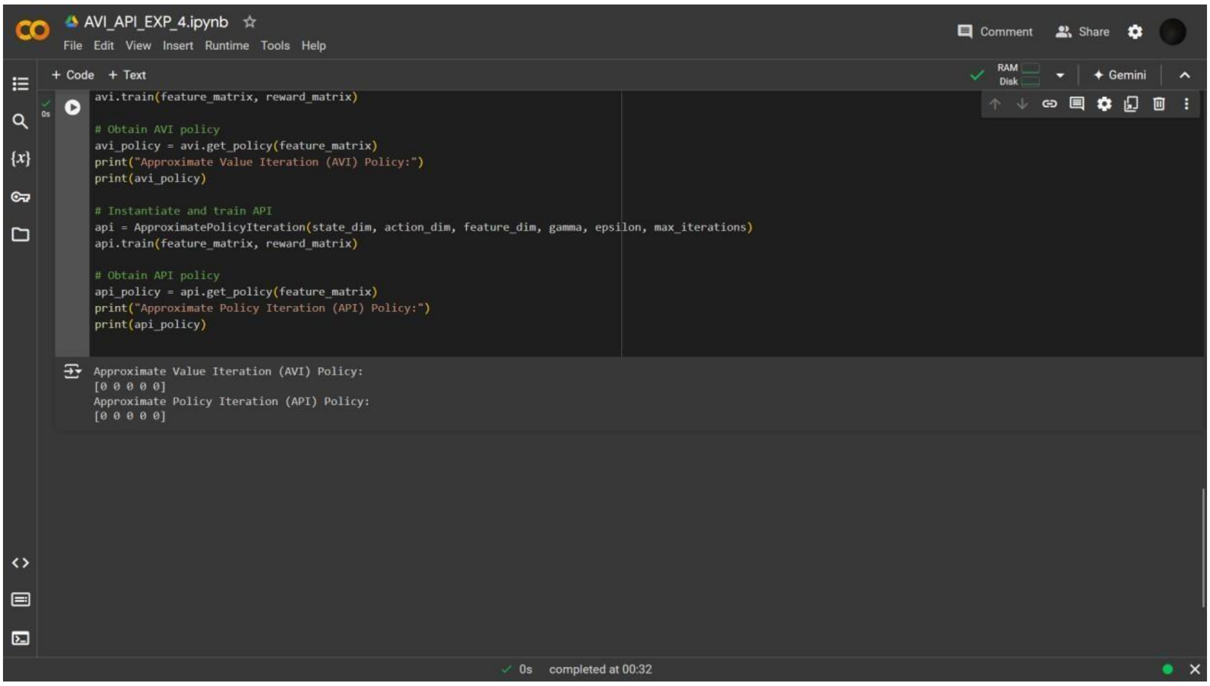
Implementation (Code):

```
import numpy as np

# AVI implementation
class ApproximateValueIteration:
    def_init_(self, state_dim, action_dim, feature_dim, gamma=0.9,
epsilon=1e-6, max_iterations=1000):
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.feature_dim = feature_dim
        self.gamma = gamma
        self.epsilon = epsilon
        self.max_iterations = max_iterations
        self.weights = np.zeros((action_dim, feature_dim))

    def compute_q_values(self, state):
        q_values = np.dot(self.weights, state)
```

```
        return q_values

    def train(self, feature_matrix, reward_matrix):
        for _ in range(self.max_iterations):
            prev_weights = np.copy(self.weights)
            for state_idx in range(self.state_dim):
                state = feature_matrix[state_idx]
                q_values = self.compute_q_values(state)
                best_action_value = np.max(q_values)
                for action_idx in range(self.action_dim):
                    reward = reward_matrix[action_idx, state_idx]
                    bellman_residual = reward + self.gamma *
best_action_value - q_values[action_idx]
                    self.weights[action_idx] += np.dot(state,
bellman_residual)
            if np.linalg.norm(prev_weights - self.weights) <
self.epsilon:
                break

    def get_policy(self, feature_matrix):
        policy = np.zeros(self.state_dim, dtype=int)
        for state_idx in range(self.state_dim):
            state = feature_matrix[state_idx]
            q_values = self.compute_q_values(state)
            policy[state_idx] = np.argmax(q_values)
        return policy

# API implementation
class ApproximatePolicyIteration:
    def_init_(self, state_dim, action_dim, feature_dim, gamma=0.9,
epsilon=1e-6, max_iterations=1000):
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.feature_dim = feature_dim
        self.gamma = gamma
        self.epsilon = epsilon
        self.max_iterations = max_iterations
        self.weights = np.zeros((action_dim, feature_dim))

    def compute_q_values(self, state):
        q_values = np.dot(self.weights, state)
        return q_values

    def compute_value_function(self, feature_matrix):
        value_function  = np.zeros(self.state_dim)
        for state_idx in range(self.state_dim):
            state = feature_matrix[state_idx]
            q_values = self.compute_q_values(state)
            value_function[state_idx] = np.max(q_values)
        return value_function

    def train(self, feature_matrix, reward_matrix):
        for _ in range(self.max_iterations):
            prev_weights = np.copy(self.weights)
            for _ in range(self.max_iterations):
                prev_value_function =
self.compute_value_function(feature_matrix)
                for state_idx in range(self.state_dim):
                    state = feature_matrix[state_idx]
                    q_values = self.compute_q_values(state)
```

```
                    policy = np.argmax(q_values)
                    reward = reward_matrix[policy, state_idx]
                    bellman_residual = reward + self.gamma *
prev_value_function[state_idx] - q_values[policy]
                    self.weights[policy] += np.dot(state,
bellman_residual)
                value_function =
self.compute_value_function(feature_matrix)
                if np.linalg.norm(prev_value_function -
value_function) < self.epsilon:
                    break
            if np.linalg.norm(prev_weights - self.weights) <
self.epsilon:
                break

    def get_policy(self, feature_matrix):
        policy = np.zeros(self.state_dim, dtype=int)
        for state_idx in range(self.state_dim):
            state = feature_matrix[state_idx]
            q_values = self.compute_q_values(state)
            policy[state_idx] = np.argmax(q_values)
        return policy

# Example Usage and Output:

# Define example data
state_dim = 5
action_dim = 2
feature_dim = 3
gamma = 0.9
epsilon = 1e-6
max_iterations = 1000

feature_matrix = np.random.rand(state_dim, feature_dim)
reward_matrix = np.random.rand(action_dim, state_dim)

# Instantiate and train AVI
avi = ApproximateValueIteration(state_dim, action_dim, feature_dim,
gamma, epsilon, max_iterations)
avi.train(feature_matrix, reward_matrix)

# Obtain AVI policy
avi_policy = avi.get_policy(feature_matrix)
print("Approximate Value Iteration (AVI) Policy:")
print(avi_policy)

# Instantiate and train API
api = ApproximatePolicyIteration(state_dim, action_dim, feature_dim,
gamma, epsilon, max_iterations)
api.train(feature_matrix, reward_matrix)

# Obtain API policy
api_policy = api.get_policy(feature_matrix)
print("Approximate Policy Iteration (API) Policy:")
print(api_policy)
```

Output Screenshots:



Conclusion (Students should write in their own words):

In conclusion, AVI and API are powerful reinforcement learning algorithms suited for handling large state spaces in Markov Decision Processes. These algorithms leverage function approximation techniques to represent value functions or policies, enabling efficient learning and scalability. While AVI directly approximates the optimal value function, API alternates between policy evaluation and improvement steps to derive the optimal policy. Both algorithms offer effective solutions for a wide range of reinforcement learning problems.

Applications:

AVI and API algorithms find applications in various domains, including robotics, finance, healthcare, and gaming. In robotics, these algorithms are used for path planning and robot control in complex environments. In finance, they assist in portfolio optimization and algorithmic trading strategies. In healthcare, they aid in treatment planning and medical diagnosis. Additionally, in gaming, AVI and API are employed for developing intelligent agents capable of making optimal decisions in dynamic game environments.

Batch: MSc AI	Roll No.: 4748118
Experiment 05	

Title: Write a program to implement Actor-critic algorithm

Objective:

Understand Actor-critic algorithm
Apply Actor-critic algorithm by implementing it.

Books/ Journals/ Websites referred:

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- https://pylessons.com/A2C-reinforcement-learning/
- https://medium.com/intro-to-artificial-intelligence/the-actor-critic-reinforcement-learning-algorithm-c8095a655c14
- https://towardsdatascience.com/reinforcement-learning-w-keras-openai-actor-critic-models-f084612cfd69
- https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic
- https://github.com/dennybritz/reinforcement-learning/blob/master/PolicyGradient/CliffWalk%20Actor%20Critic%20Solution.ipynb
- https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/reinforcement_learning/actor_critic.ipynb

Theory:

The Actor-Critic algorithm is a reinforcement learning technique that combines aspects of both policy-based and value-based methods. It consists of two components: the actor, which learns a policy to select actions, and the critic, which evaluates the quality of the actions taken by the actor.

Key Concepts:

Actor:

The actor is responsible for learning a policy that maps states to actions. It directly interacts with the environment and selects actions based on the current policy.

Critic:

The critic evaluates the actions chosen by the actor by estimating the value function. It provides feedback to the actor by assessing the goodness of the chosen actions.

Advantages:

The actor-critic architecture combines the advantages of both policy-based and value-based methods. It can handle both discrete and continuous action spaces and is more sample-efficient compared to traditional policy-based methods.

Policy Gradient:

The actor uses policy gradient methods to update its parameters based on the expected return. The critic provides a baseline for the policy gradient by estimating the value function.

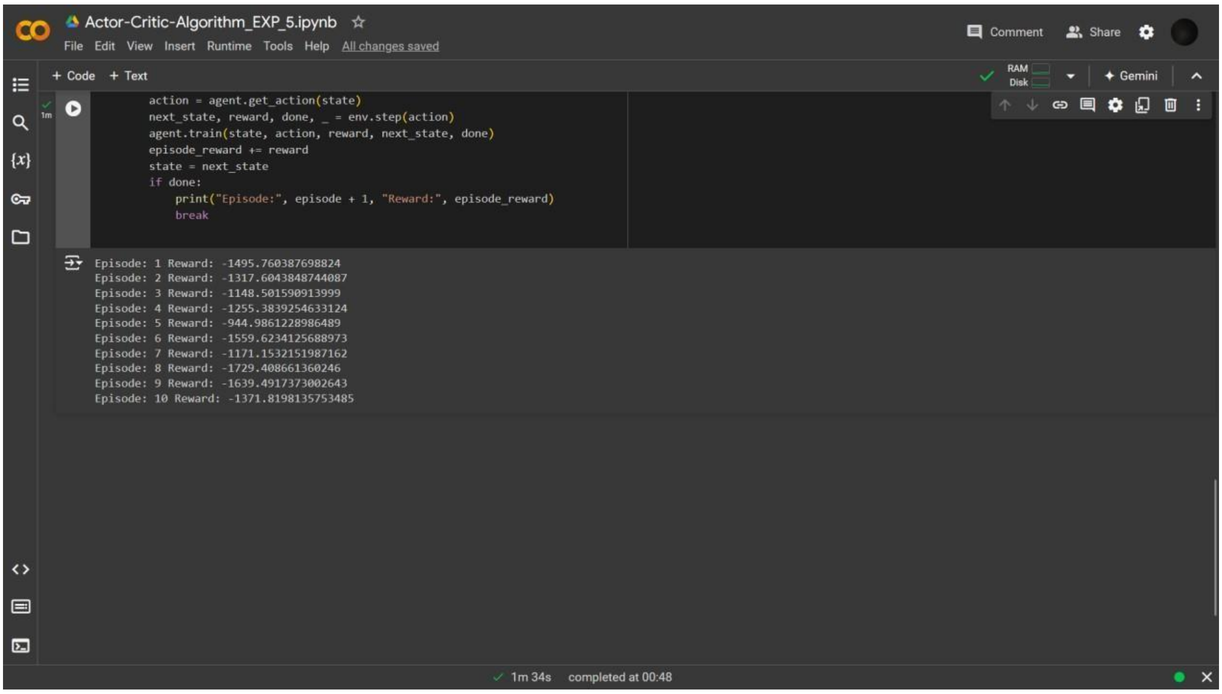
Implementation (Code):

```
import numpy as np
import tensorflow as tf
import gym

# Actor Model
class Actor(tf.keras.Model):
    def__init__(self,  state_dim,  action_dim,  action_bound):
        super(Actor, self)._init__()
        self.dense1 = tf.keras.layers.Dense(64, activation='relu')
        self.dense2 = tf.keras.layers.Dense(32, activation='relu')
        self.dense3 = tf.keras.layers.Dense(action_dim,
activation='tanh')
        self.action_bound = action_bound

    def call(self, inputs):
        # Reshape the input tensor to have a shape of (batch_size,
input_dim)
        x = tf.expand_dims(inputs, axis=0)  # Add a batch dimension
        x = self.dense1(x)
        x = self.dense2(x)
        x = self.dense3(x)
        return tf.squeeze(x, axis=0) # Remove the added batch
dimension


# Critic Model
class Critic(tf.keras.Model):
    def_init_(self):
        super(Critic, self).__init__()
        self.dense1 = tf.keras.layers.Dense(64, activation='relu')
        self.dense2 = tf.keras.layers.Dense(32, activation='relu')
        self.dense3 = tf.keras.layers.Dense(1)

    def call(self, inputs):
        # Reshape the input tensor to have a shape of (batch_size,
input_dim)
        x = tf.expand_dims(inputs, axis=0)  # Add a batch dimension
        x = self.dense1(x)
        x = self.dense2(x)
        x = self.dense3(x)
        return tf.squeeze(x, axis=0) # Remove the added batch
dimension


#   Actor-Critic   Agent
class ActorCriticAgent:
```

```
    def_init_(self, state_dim, action_dim, action_bound, 
gamma=0.99, actor_lr=0.001, critic_lr=0.001):
        self.actor = Actor(state_dim, action_dim, action_bound)
        self.critic = Critic()
        self.actor_optimizer = 
tf.keras.optimizers.Adam(learning_rate=actor_lr)
        self.critic_optimizer = 
tf.keras.optimizers.Adam(learning_rate=critic_lr)
        self.gamma = gamma

    def get_action(self, state):
        return self.actor(tf.convert_to_tensor([state])).numpy()[0]

    def train(self, states, actions, rewards, next_states, dones):
        # Compute TD targets
        next_q_values = self.critic(tf.convert_to_tensor(next_states, 
dtype=tf.float32))
        targets = rewards + (1 - dones) * self.gamma * 
next_q_values.numpy().flatten()

        # Compute advantages
        values = self.critic(tf.convert_to_tensor(states, 
dtype=tf.float32)).numpy().flatten()
        advantages = targets - values

        # Train actor
        with tf.GradientTape() as tape:
            actor_actions = self.actor(tf.convert_to_tensor(states, 
dtype=tf.float32))
            actor_loss = -
tf.reduce_mean(self.critic(tf.convert_to_tensor(states, 
dtype=tf.float32)) * actor_actions)
        actor_grads = tape.gradient(actor_loss, 
self.actor.trainable_variables)
        self.actor_optimizer.apply_gradients(zip(actor_grads, 
self.actor.trainable_variables))

        # Train critic
        with tf.GradientTape() as tape:
            critic_values = self.critic(tf.convert_to_tensor(states, 
dtype=tf.float32))
            critic_loss = tf.reduce_mean(tf.square(targets - 
critic_values))
        critic_grads = tape.gradient(critic_loss, 
self.critic.trainable_variables)
        self.critic_optimizer.apply_gradients(zip(critic_grads, 
self.critic.trainable_variables))

# Example Usage
env = gym.make('Pendulum-v0')
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
action_bound = env.action_space.high[0]

agent = ActorCriticAgent(state_dim, action_dim, action_bound)

episodes = 10
for episode in range(episodes):
    state = env.reset()
    episode_reward = 0
```

```
    while True:
        action = agent.get_action(state)
        next_state, reward, done, _ = env.step(action)
        agent.train(state, action, reward, next_state, done)
        episode_reward += reward
        state = next_state
        if done:
            print("Episode:", episode + 1, "Reward:", episode_reward)
            break
```

Output Screenshots:



Conclusion (Students should write in their own words):

In conclusion, the Actor-Critic algorithm offers a powerful framework for reinforcement learning, combining the benefits of both policy-based and value-based methods. By leveraging the actor to learn a policy and the critic to evaluate actions, the algorithm can effectively navigate complex environments and learn optimal behavior.

Applications:

Actor-Critic algorithms find applications in various domains, including robotics, finance, gaming, and healthcare. They are used for tasks such as robot control, algorithmic trading, game AI development, and medical decision-making. The flexibility and efficiency of Actor-Critic methods make them well-suited for real-world reinforcement learning problems with continuous action spaces and sparse rewards.

Batch: MSc AI	**Roll No.: 4748118**
Experiment 06	

Title: Write a program to implement Real-Time Dynamic Programming (RTDP)

Books/ Journals/ Websites referred:

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- https://towardsdatascience.com/introduction-to-reinforcement-learning-rl-part-4-dynamic-programming-6af57e575b3d
- https://github.com/instance01/RTDP

Theory:

Real-Time Dynamic Programming (RTDP) is a reinforcement learning algorithm used to solve large state-space problems efficiently. It is particularly effective for problems with continuous state and action spaces. RTDP iteratively updates the value function and policy until convergence or a maximum number of iterations is reached.

Key Concepts:

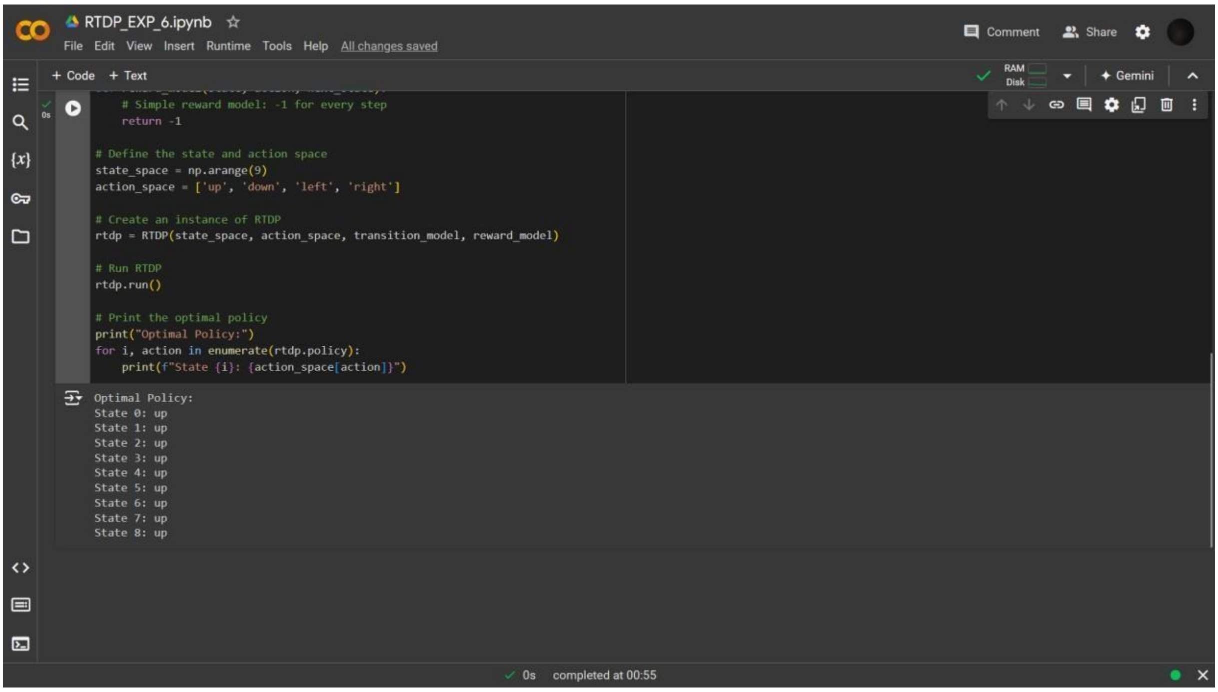
1. **State Space**: Represents all possible states in the environment.
2. **Action Space**: Represents all possible actions that the agent can take in a given state.
3. **Transition Model**: Defines the probability of transitioning from one state to another based on the agent's action.
4. **Reward Model**: Specifies the immediate reward the agent receives for taking a particular action in a given state.
5. **Value Function**: Estimates the expected cumulative reward from a given state onwards.
6. **Policy**: Defines the agent's behavior by mapping states to actions.

Implementation (Code):

```
import numpy as np

class RTDP:
    def_init_(self, state_space, action_space, transition_model,
reward_model, gamma=0.9, max_iterations=1000):
        self.state_space = state_space
        self.action_space = action_space
        self.transition_model = transition_model
        self.reward_model = reward_model
        self.gamma = gamma
        self.max_iterations = max_iterations
```

```
        self.value_function = np.zeros(len(state_space))
        self.policy = np.zeros(len(state_space), dtype=int)

    def run(self):
        for _ in range(self.max_iterations):
            for state in self.state_space:
                action_values = []
                for action in self.action_space:
                    next_state = self.transition_model(state, action)
                    reward = self.reward_model(state, action,
next_state)
                    action_value = reward + self.gamma *
self.value_function[next_state]
                    action_values.append(action_value)
                best_action = np.argmax(action_values)
                best_value = action_values[best_action]
                self.value_function[state] = best_value
                self.policy[state] = best_action

def transition_model(state, action):
    # Simple grid world transition model
    if action == 'up':
        return state - 3 if state >= 3 else state
    elif action == 'down':
        return state + 3 if state < 6 else state
    elif action == 'left':
        return state - 1 if state % 3 != 0 else state
    elif action == 'right':
        return state + 1 if state % 3 != 2 else state

def reward_model(state, action, next_state):
    # Simple reward model: -1 for every step
    return -1

# Define the state and action space
state_space = np.arange(9)
action_space = ['up', 'down', 'left', 'right']

# Create an instance of RTDP
rtdp = RTDP(state_space, action_space, transition_model, reward_model)

# Run RTDP
rtdp.run()

# Print the optimal policy
print("Optimal Policy:")
for i, action in enumerate(rtdp.policy):
    print(f"State {i}: {action_space[action]}")
```

Output Screenshots:



Conclusion (Students should write in their own words):

In conclusion, Real-Time Dynamic Programming (RTDP) is a powerful reinforcement learning algorithm for solving large state-space problems efficiently. By iteratively updating the value function and policy, RTDP can find an optimal policy for the agent to navigate through the environment.

Applications:

RTDP has various applications in robotics, autonomous systems, and game playing. It can be used for path planning, robot navigation, and optimizing strategies in board games such as chess and Go. Additionally, RTDP is useful in domains with continuous state and action spaces, making it applicable to a wide range of real-world problems.

Batch: MSc AI **Roll No.: 4748118**

Experiment 07

Title: Write a program to implement SARSA algorithm

Objective:

To understand working of SARSA Algorithm

To implement SARSA Algorithm

Books/ Journals/ Websites referred:

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- https://gym.openai.com/docs/
- https://www.geeksforgeeks.org/sarsa-reinforcement-learning/

Resources Needed:

- Python,
- Numpy,
- Gym (Gym is a toolkit for developing and comparing reinforcement learning algorithms.)

Theory:

The SARSA (State-Action-Reward-State-Action) algorithm is an on-policy reinforcement learning algorithm used for learning the optimal policy for a Markov Decision Process (MDP). It belongs to the family of Temporal Difference (TD) learning algorithms and is similar to Q-learning but differs in that it updates the Q-values based on the action actually taken in the next state.

Key Concepts:

1. **State**: Represents the current situation or configuration of the environment.
2. **Action**: Represents the possible decisions or moves the agent can take in a given state.

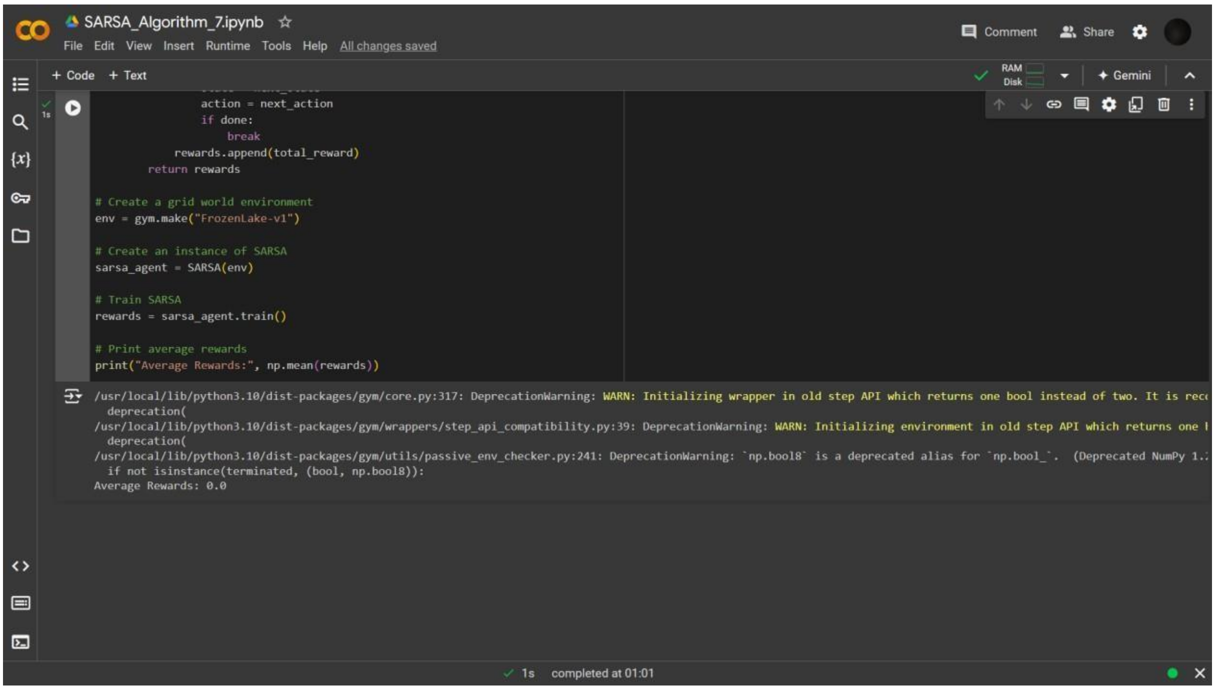
3. **Reward**: Represents the immediate feedback the agent receives for taking a particular action in a given state.
4. **Policy**: Defines the strategy or behavior of the agent, mapping states to actions.
5. **Q-Value**: Represents the expected cumulative reward of taking a particular action in a particular state and following a given policy thereafter.
6. **Exploration-Exploitation Tradeoff**: Balances between exploring new actions and exploiting the current best-known actions.

Implementation (Code):

```
import numpy as np
import random
import gym

class SARSA:
    def_init_(self, env, alpha=0.1, gamma=0.99, epsilon=0.1, 
max_episodes=1000, max_steps=100):
        self.env = env
        self.alpha = alpha # learning rate
        self.gamma = gamma  # discount factor
        self.epsilon = epsilon # exploration-exploitation tradeoff
        self.max_episodes = max_episodes
        self.max_steps = max_steps
        self.q_table = np.zeros((env.observation_space.n, 
env.action_space.n))

    def choose_action(self, state):
        if np.random.uniform(0, 1) < self.epsilon:
            return self.env.action_space.sample()  # Explore action 
space
        else:
            return np.argmax(self.q_table[state, :])  # Exploit 
learned values

    def update_q_table(self, state, action, reward, next_state, 
next_action):
        predict = self.q_table[state, action]
        target = reward + self.gamma * self.q_table[next_state, 
next_action]
        self.q_table[state, action] += self.alpha * (target - predict)

    def train(self):
        rewards = []
        for episode in range(self.max_episodes):
            state = self.env.reset()
            total_reward = 0
            action = self.choose_action(state)
            for step in range(self.max_steps):
                next_state, reward, done, _ = self.env.step(action)
                next_action = self.choose_action(next_state)
                self.update_q_table(state, action, reward, next_state, 
next_action)
                total_reward += reward
                state = next_state
                action = next_action
                if done:
                    break
            rewards.append(total_reward)
```

```
        return rewards

# Create a grid world environment
env = gym.make("FrozenLake-v1")

# Create an instance of SARSA
sarsa_agent = SARSA(env)

# Train SARSA
rewards = sarsa_agent.train()

# Print average rewards
print("Average Rewards:", np.mean(rewards))
```

Output Screenshots:



Conclusion (Students should write in their own words):

In conclusion, the SARSA algorithm is a powerful on-policy reinforcement learning algorithm used for learning the optimal policy in a Markov Decision Process. By iteratively updating the Q-values based on the actions actually taken and following the policy thereafter, SARSA can effectively learn to navigate through the environment and achieve the maximum cumulative reward.

Applications:

SARSA has various applications in robotics, game playing, and autonomous systems. It can be used for robot navigation, path planning, and optimizing strategies in board games. Additionally, SARSA is useful in domains with continuous state and action spaces, making it applicable to a wide range of real-world problems.

Batch: MSc AI **Roll No.: 4748118**

Experiment 08

Title: Write a program to implement Rollout algorithm

Books/ Journals/ Websites referred:

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- https://medium.com/chiukevin0321/motion-planning-for-self-driving-cars-week-5-6-4de794bcad66
- https://github.com/alirezaig/RolloutPower

Theory:

The Rollout algorithm is a Monte Carlo Tree Search (MCTS) method used in decision-making processes, particularly in environments with large state and action spaces. It involves simulating a large number of rollouts (sequences of actions from the current state to a terminal state) to estimate the value of each action and select the best action based on these estimates.

Key Concepts:

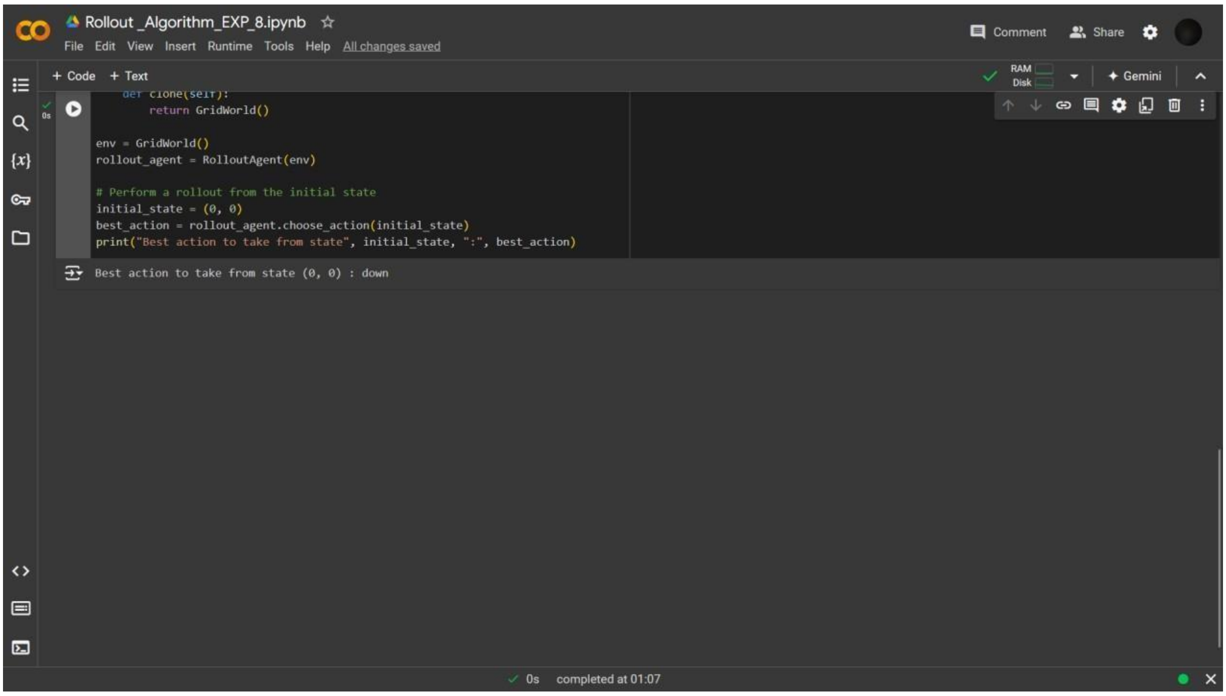
1. **Rollout**: A sequence of actions from the current state to a terminal state, simulated to estimate the value of each action.
2. **Monte Carlo Tree Search (MCTS)**: A search algorithm that builds a search tree by randomly sampling possible actions and evaluating their outcomes through simulations.
3. **State**: Represents the current situation or configuration of the environment.
4. **Action**: Represents the possible decisions or moves the agent can take in a given state.
5. **Value Estimation**: The process of estimating the value or utility of taking a particular action in a given state, typically based on the average outcome of rollouts.
6. **Exploration-Exploitation Tradeoff**: Balances between exploring new actions and exploiting the current best-known actions.

Implementation (Code):

```
import numpy as np
import random

class RolloutAgent:
    def_init_(self, env, max_rollouts=100):
        self.env = env
        self.max_rollouts = max_rollouts

    def rollout(self, state, action):
        total_reward = 0
        for _ in range(self.max_rollouts):
            rollout_env = self.env.clone() # Create a copy of the
environment for the rollout
            rollout_env.set_state(state)
            rollout_env.step(action)
            rollout_reward = 0
            done = False
            while not done:
                rollout_action =
random.choice(rollout_env.get_possible_actions())
                _, reward, done, _ = rollout_env.step(rollout_action)
                rollout_reward += reward
            total_reward += rollout_reward
        return total_reward / self.max_rollouts

    def choose_action(self, state):
        possible_actions = self.env.get_possible_actions()
        action_values = [self.rollout(state, action) for action in
possible_actions]
        best_action = possible_actions[np.argmax(action_values)]
        return best_action

# Example Usage
class GridWorld:
    def_init_(self):
        self.state = (0, 0)
        self.grid_size = 5

    def set_state(self, state):
        self.state = state

    def get_possible_actions(self):
        return ['up', 'down', 'left', 'right']

    def step(self, action):
        if action == 'up' and self.state[0] > 0:
            self.state = (self.state[0] - 1, self.state[1])
        elif action == 'down' and self.state[0] < self.grid_size - 1:
            self.state = (self.state[0] + 1, self.state[1])
        elif action == 'left' and self.state[1] > 0:
            self.state = (self.state[0], self.state[1] - 1)
        elif action == 'right' and self.state[1] < self.grid_size - 1:
            self.state = (self.state[0], self.state[1] + 1)
        reward = -1 if self.state != (self.grid_size - 1,
self.grid_size - 1) else 0  # -1 for each step, 0 at goal
        done = self.state == (self.grid_size - 1, self.grid_size - 1)
        return self.state, reward, done, {}
```

```
    def clone(self):
        return GridWorld()

env = GridWorld()
rollout_agent = RolloutAgent(env)

# Perform a rollout from the initial state
initial_state = (0, 0)
best_action = rollout_agent.choose_action(initial_state)
print("Best action to take from state", initial_state, ":",
best_action)
```

Output Screenshots:



Conclusion (Students should write in their own words):

In conclusion, the Rollout algorithm is a Monte Carlo Tree Search (MCTS) method used for decision-making in environments with large state and action spaces. By simulating multiple rollouts and estimating the value of each action based on these simulations, the Rollout algorithm can effectively select the best action to take in a given state.

Applications:

Rollout has various applications in games, robotics, and decision-making systems. It can be used for game playing, path planning, and strategy optimization. Additionally, Rollout is useful in domains with complex environments and uncertain outcomes, making it applicable to a wide range of real-world problems.

Batch: MSc AI	Roll No.: 4748118
Experiment 09 (A)	

Title: Write a program to implement SuperMarioBros using OpenAI

Theory:

Super Mario Bros is a classic platform video game developed and published by Nintendo. It features the iconic character Mario as he navigates through various levels, overcoming obstacles, and defeating enemies to rescue Princess Peach from the villain Bowser. In the context of reinforcement learning, Super Mario Bros provides a challenging environment for agents to learn complex behaviors and strategies.

Implementation (Code):

```
import gym_super_mario_bros
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT
from nes_py.wrappers import JoypadSpace
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import VecFrameStack,
DummyVecEnv
from stable_baselines3.common.evaluation import evaluate_policy
import wandb
from wandb.integration.sb3 import WandbCallback
import os

# Initialize Gym environment
env = gym_super_mario_bros.make('SuperMarioBros-v0')
env = JoypadSpace(env, SIMPLE_MOVEMENT)
env = GrayScaleObservation(env, keep_dim=True)
env = DummyVecEnv([lambda: env])
env = VecFrameStack(env, 4, channels_order='last')

# Define environment name and configuration
env_name = "SuperMarioBros-v0"
config = {
    "policy_type": "CnnPolicy",
    "total_timesteps": 25000,
    "env_name": env_name,
}

# Initialize WandB run
run = wandb.init(
    project="intro_to_gym",
    config=config,
    sync_tensorboard=True,
    monitor_gym=True,
    save_code=True,
)

# Setup video recording
env_with_recording = VecVideoRecorder(
```

```
    env, f"videos/{run.id}",
    record_video_trigger=lambda x: x % 2000 == 0,
    video_length=200
)

# Create and train the model
model = PPO(config["policy_type"], env_with_recording, verbose=1,
tensorboard_log=f"runs/{run.id}")
model.learn(
    total_timesteps=config["total_timesteps"],
    callback=WandbCallback(
        gradient_save_freq=10,
        model_save_path=f"models/{run.id}",
        verbose=2,
    ),
)

# Save the trained model
PPO_path = os.path.join('Training', 'Saved Models',
'PPO_SuperMario_25k')
model.save(PPO_path)

# Finish WandB run
run.finish()

# Evaluate the trained model
evaluate_policy(model, env, n_eval_episodes=10, render=True)
```

Output Screenshots:

Conclusion (Students should write in their own words):

In conclusion, the implementation above demonstrates how to set up the Super Mario Bros environment using the OpenAI Gym interface and create a simple agent that interacts with the environment by choosing actions. While this implementation is basic and the agent's behavior is random, it provides a starting point for more advanced reinforcement learning techniques to be applied to Super Mario Bros.

Applications:

Super Mario Bros provides a rich environment for exploring various reinforcement learning algorithms and techniques. It can be used to develop and test agents that learn to navigate complex environments, overcome obstacles, and achieve specific goals. Additionally, studying Super Mario Bros can lead to insights and advancements in areas such as game AI, robotics, and autonomous systems.

Batch: MSc AI **Roll No.: 4748118**

Experiment 09 (B)

Title: Write a program to implement BipedalWalker-v3 using OpenAI

Theory:

The BipedalWalker-v3 environment from OpenAI Gym is a simulation where an agent, represented as a bipedal walker, must learn to walk forward without falling. The agent receives a reward for moving forward and staying upright, while penalties are incurred for falling or taking actions that lead to instability. This environment presents a challenging problem for reinforcement learning algorithms due to the high-dimensional action space and the need for delicate balance and coordination.

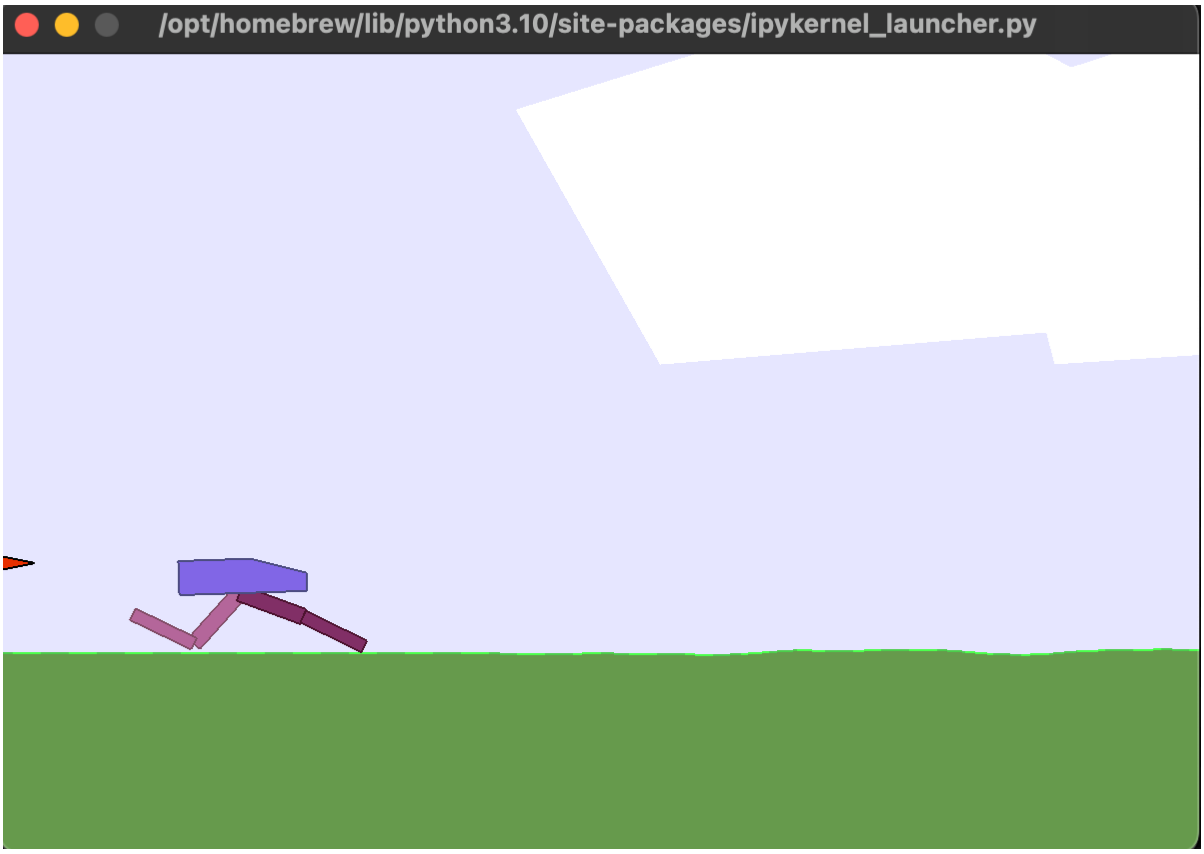
Implementation (Code):

```
!pip install 'stable-baselines3[extra]'

!pip install wandb

!pip install box2d-py

!pip install gym_super_mario_bros==7.3.0 nes_py

!pip install opencv-python

import gym

import os

import wandb

import gym_super_mario_bros

from nes_py.wrappers import JoypadSpace

from gym.wrappers import GrayScaleObservation

from wandb.integration.sb3 import WandbCallback

from gym_super_mario_bros.actions import SIMPLE_MOVEMENT

from stable_baselines3 import PPO
```

```
from stable_baselines3.common.vec_env import DummyVecEnv,VecVideoRecorder,
VecFrameStack

from stable_baselines3.common.evaluation import evaluate_policy

from stable_baselines3.common.monitor import Monitor

import gym

# Define the environment name

env_name = "BipedalWalker-v3"

config = {

    "policy_type": "MlpPolicy",

    "total_timesteps": 250000,

    "env_name": env_name,

}

run = wandb.init(

    project="intro_to_gym",

    config=config,

    sync_tensorboard=True,

    monitor_gym=True,

    save_code=True,

)

model=PPO(config["policy_type"],env,verbose=1, tensorboard_log=f"runs/{run.id}")

model.learn(

    total_timesteps=config["total_timesteps"],

    callback=WandbCallback(

        gradient_save_freq=100,

        model_save_path=f"models/{run.id}",
```

```
        verbose=2,
    ),
)
PPO_path = os.path.join('Training', 'Saved Models', 'PPO_BipedalWalker_250k')
model.save(PPO_path)
evaluate_policy(model, env, n_eval_episodes=10, render=True)
run.finish()
```

Output Screenshots:



Conclusion (Students should write in their own words):

In this exercise, we implemented a basic script to interact with the BipedalWalker-v3 environment from OpenAI Gym. The script demonstrates how an agent can take random actions in the environment and receive rewards based on its performance. While this approach is simplistic and unlikely to achieve meaningful progress in learning to walk, it provides a starting point for more sophisticated reinforcement learning algorithms.

Applications:

The BipedalWalker-v3 environment serves as a testbed for developing and evaluating reinforcement learning algorithms that can tackle complex locomotion tasks. Applications of such algorithms extend beyond simulated environments to real-world scenarios like robotics, where bipedal locomotion is a fundamental challenge. By mastering the BipedalWalker-v3 environment, agents can acquire skills that are transferable to physical robots, enabling them to navigate various terrains and environments autonomously.

Batch: MSc AI	**Roll No.: 4748118**
Experiment 09 (C)	

Title: Write a program to implement CartPole-v1 using OpenAI

Theory:

Q-learning is a model-free reinforcement learning algorithm used to find the optimal action-selection policy for any given finite Markov decision process (MDP). It uses a Q-table to store the Q-values, which represent the expected future rewards for state-action pairs. The agent updates the Q-values based on the Bellman equation:

$Q(s,a) \leftarrow Q(s,a)+\alpha[r+\gamma \max a' Q(s',a')-Q(s,a)]Q(s,a) \leftarrow Q(s,a)+\alpha[r+\gamma \max a'$
$Q(s',a')-Q(s,a)]$

where:

- $\alpha\alpha$ is the learning rate.
- $\gamma\gamma$ is the discount factor.
- rr is the reward received after taking action aa in state ss.
- $s's'$ is the next state.
- $a'a'$ is the next action.

The agent learns by exploring the environment, updating its Q-table, and gradually reducing its exploration rate to shift from exploration to exploitation of the learned policy.

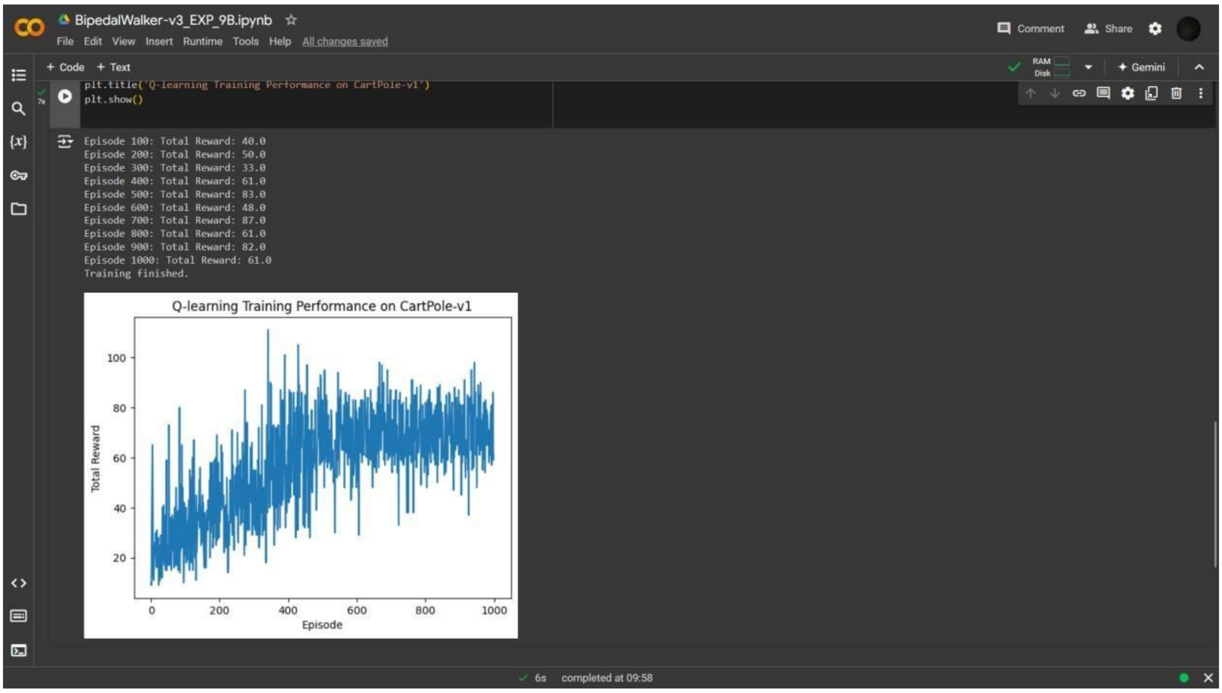
Implementation (Code):

```
import gym
import numpy as np
import random
import matplotlib.pyplot as plt

# Create the CartPole-v1 environment
env = gym.make('CartPole-v1')

# Set the seed for reproducibility
env.seed(42)
np.random.seed(42)
random.seed(42)

class QLearningAgent:
    def_init_(self, state_bins, action_size):
        self.state_bins = state_bins
        self.action_size = action_size
        self.q_table = np.zeros(tuple(len(bins) + 1 for bins in
state_bins) + (action_size,))
        self.learning_rate = 0.1
```

```
        self.discount_factor = 0.99
        self.exploration_rate = 1.0
        self.exploration_decay = 0.995
        self.exploration_min = 0.01

    def get_discrete_state(self, state):
        discrete_state = []
        for i in range(len(state)):
            discrete_state.append(np.digitize(state[i],
self.state_bins[i]) - 1)
        return tuple(discrete_state)

    def choose_action(self, state):
        if np.random.rand() <= self.exploration_rate:
            return random.choice(range(self.action_size))
        return np.argmax(self.q_table[state])

    def learn(self, state, action, reward, next_state, done):
        best_next_action = np.argmax(self.q_table[next_state])
        td_target = reward + self.discount_factor *
self.q_table[next_state][best_next_action] * (not done)
        td_error = td_target - self.q_table[state][action]
        self.q_table[state][action] += self.learning_rate * td_error
        if done:
            self.exploration_rate = max(self.exploration_min,
self.exploration_rate * self.exploration_decay)

state_bins = [
    np.linspace(-2.4, 2.4, 10),  # Cart position
    np.linspace(-3.0, 3.0, 10),  # Cart velocity
    np.linspace(-0.5, 0.5, 10), # Pole angle
    np.linspace(-2.0, 2.0, 10)   # Pole velocity at tip
]

# Initialize the Q-learning agent
agent = QLearningAgent(state_bins=state_bins,
action_size=env.action_space.n)

# Training parameters
num_episodes = 1000
max_steps = 200
rewards = []
for episode in range(num_episodes):
    state = env.reset()
    state = agent.get_discrete_state(state)
    total_reward = 0
    for step in range(max_steps):
        action = agent.choose_action(state)
        next_state, reward, done, _ = env.step(action)
        next_state = agent.get_discrete_state(next_state)
        agent.learn(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward
        if done:
            break
    rewards.append(total_reward)
    if (episode + 1) % 100 == 0:
        print(f"Episode {episode + 1}: Total Reward: {total_reward}")

print("Training finished.\n")
```

```
# Plot the results
plt.plot(rewards)
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('Q-learning Training Performance on CartPole-v1')
plt.show()
```

Output Screenshots:



Conclusion (Students should write in their own words):

They should reflect on the learning process and the importance of parameters such as the learning rate, discount factor, and exploration rate. Discuss the performance observed in the CartPole-v1 environment and how it demonstrates the agent's learning capabilities over time.

Applications:

1. **Game AI**: Q-learning is widely used in game AI to develop intelligent agents that can learn optimal strategies in various games.
2. **Robotics**: It is used in robotics for path planning and decision-making to enable robots to navigate and interact with their environment autonomously.
3. **Finance**: Q-learning is applied in algorithmic trading to develop strategies that adapt to market changes and optimize trading decisions.
4. **Healthcare**: Used in personalized treatment plans where the agent learns the best course of action for patient care based on historical data.
5. **Autonomous Vehicles**: Implemented in self-driving cars for decision-making processes such as lane changing, obstacle avoidance, and route planning.

Batch: MSc AI	**Roll No.: 4748118**
Experiment 10 (A)	

Title: Write a program to implementation of Fuzzy logic-based decision modelling for washing machine

Theory:

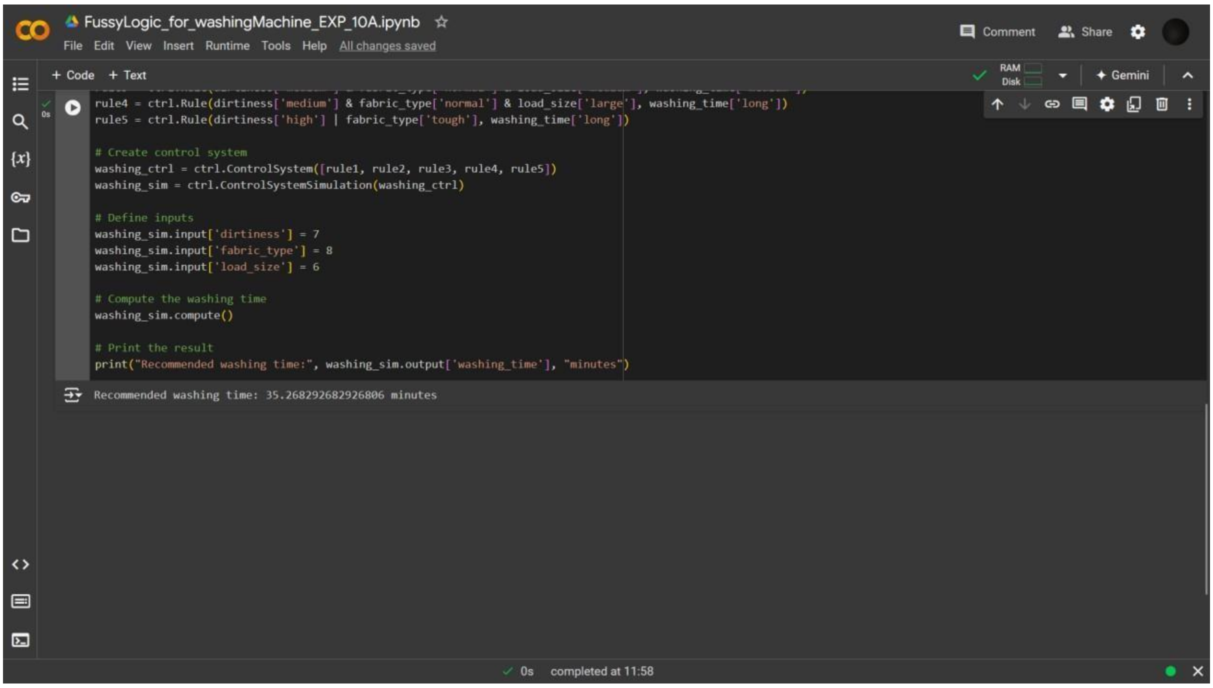
Fuzzy logic is a form of multi-valued logic that deals with reasoning that is approximate rather than precise. It allows for modeling complex systems with uncertainty or imprecision. In the context of a washing machine, fuzzy logic can be applied to control various parameters such as dirtiness level, fabric type, and load size to determine the optimal washing time.

Implementation (Code):

```
pip install scikit-fuzzy

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

# Define input variables
dirtiness = ctrl.Antecedent(np.arange(0, 11, 1), 'dirtiness')
fabric_type = ctrl.Antecedent(np.arange(0, 11, 1), 'fabric_type')
load_size = ctrl.Antecedent(np.arange(0, 11, 1), 'load_size')

# Define output variable
washing_time = ctrl.Consequent(np.arange(0, 61, 1), 'washing_time')

# Define membership functions
dirtiness['low'] = fuzz.trimf(dirtiness.universe, [0, 0, 5])
dirtiness['medium'] = fuzz.trimf(dirtiness.universe, [0, 5, 10])
dirtiness['high'] = fuzz.trimf(dirtiness.universe, [5, 10, 10])

fabric_type['delicate'] = fuzz.trimf(fabric_type.universe, [0, 0, 5])
fabric_type['normal'] = fuzz.trimf(fabric_type.universe, [0, 5, 10])
fabric_type['tough'] = fuzz.trimf(fabric_type.universe, [5, 10, 10])

load_size['small'] = fuzz.trimf(load_size.universe, [0, 0, 5])
load_size['medium'] = fuzz.trimf(load_size.universe, [0, 5, 10])
load_size['large'] = fuzz.trimf(load_size.universe, [5, 10, 10])

washing_time['short'] = fuzz.trimf(washing_time.universe, [0, 0, 30])
washing_time['medium'] = fuzz.trimf(washing_time.universe, [0, 30,
60])
washing_time['long'] = fuzz.trimf(washing_time.universe, [30, 60, 60])

# Define fuzzy rules
rule1 = ctrl.Rule(dirtiness['low'] & fabric_type['delicate'],
washing_time['short'])
rule2 = ctrl.Rule(dirtiness['medium'] & fabric_type['normal'] &
load_size['small'], washing_time['short'])
```

```
rule3 = ctrl.Rule(dirtiness['medium'] & fabric_type['normal'] &
load_size['medium'], washing_time['medium'])
rule4 = ctrl.Rule(dirtiness['medium'] & fabric_type['normal'] &
load_size['large'], washing_time['long'])
rule5 = ctrl.Rule(dirtiness['high'] | fabric_type['tough'],
washing_time['long'])

# Create control system
washing_ctrl = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5])
washing_sim = ctrl.ControlSystemSimulation(washing_ctrl)

# Define inputs
washing_sim.input['dirtiness'] = 7
washing_sim.input['fabric_type'] = 8
washing_sim.input['load_size'] = 6

# Compute the washing time
washing_sim.compute()

# Print the result
print("Recommended washing time:", washing_sim.output['washing_time'],
"minutes")
```

<u>Output Screenshots:</u>



<u>Conclusion (Students should write in their own words):</u>

In conclusion, the fuzzy logic-based decision model provides a flexible and intuitive approach to control the washing process of a washing machine. By considering input variables such as dirtiness level, fabric type, and load size, the model can adaptively adjust the washing time to achieve optimal cleaning performance while minimizing energy and water consumption.

Applications:

- **Smart Washing Machines:** Fuzzy logic enables washing machines to adapt their operation to varying laundry conditions, resulting in improved efficiency and cleaning performance.
- **Energy and Water Conservation:** By accurately determining the required washing time based on input parameters, fuzzy logic models can help reduce energy and water consumption during the washing process.
- **Customized Washing Programs:** Fuzzy logic allows for the creation of customized washing programs tailored to specific fabric types or dirtiness levels, providing users with greater flexibility and control over the washing process.
- **Industrial Laundry Systems:** Fuzzy logic-based control systems can be implemented in industrial-scale laundry systems to optimize washing operations in commercial settings, such as hotels, hospitals, and laundromats.

Batch: MSc AI **Roll No.: 4748118**

Experiment 10 (B)

Title: Write a program to implementation of Fuzzy logic-based decision modelling for AC

Theory:

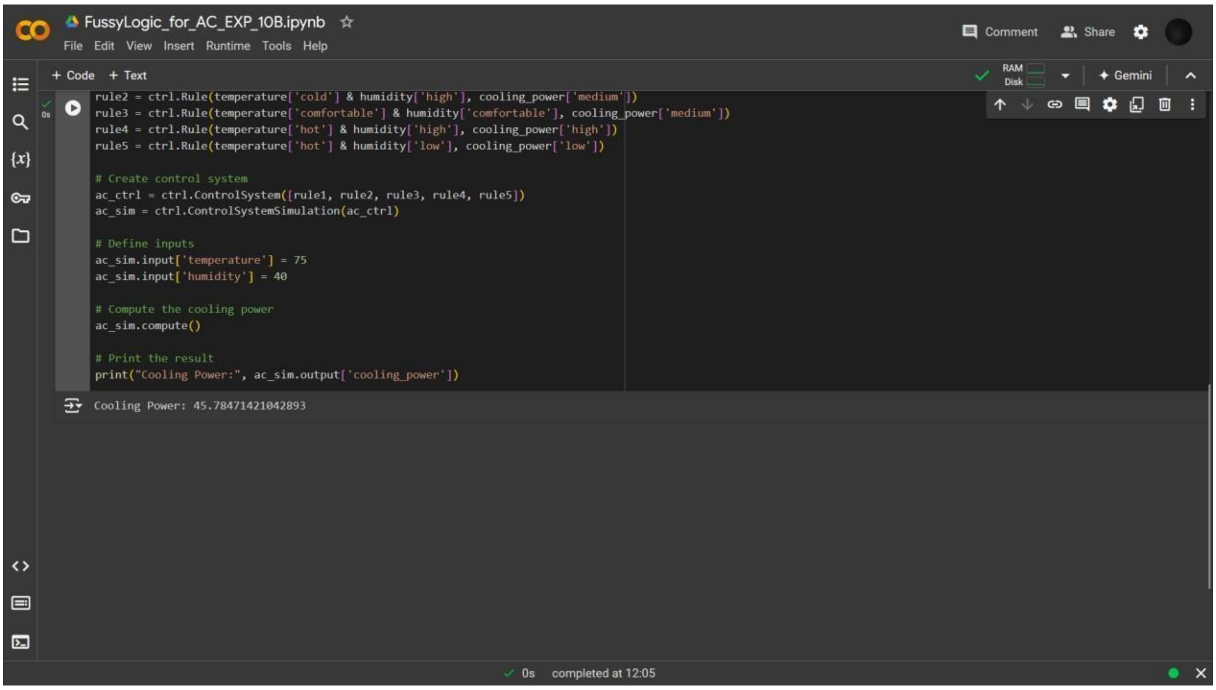
Fuzzy logic provides a way to model and control systems that have uncertain or imprecise inputs. In the context of an AC, fuzzy logic can be used to adjust parameters like temperature, humidity, and airflow to maintain a comfortable indoor environment.

Implementation (Code):

```
pip install scikit-fuzzy

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

# Define input variables
temperature = ctrl.Antecedent(np.arange(0, 101, 1), 'temperature')
humidity = ctrl.Antecedent(np.arange(0, 101, 1), 'humidity')

# Define output variable
cooling_power = ctrl.Consequent(np.arange(0, 101, 1), 'cooling_power')

# Define membership functions
temperature['cold'] = fuzz.trimf(temperature.universe, [0, 0, 50])
temperature['comfortable'] = fuzz.trimf(temperature.universe, [20, 50,
80])
temperature['hot'] = fuzz.trimf(temperature.universe, [50, 100, 100])

humidity['low'] = fuzz.trimf(humidity.universe, [0, 0, 50])
humidity['comfortable'] = fuzz.trimf(humidity.universe, [20, 50, 80])
humidity['high'] = fuzz.trimf(humidity.universe, [50, 100, 100])

cooling_power['low'] = fuzz.trimf(cooling_power.universe, [0, 0, 50])
cooling_power['medium'] = fuzz.trimf(cooling_power.universe, [0, 50,
100])
cooling_power['high'] = fuzz.trimf(cooling_power.universe, [50, 100,
100])

# Define fuzzy rules
rule1 = ctrl.Rule(temperature['cold'] & humidity['low'],
cooling_power['high'])
rule2 = ctrl.Rule(temperature['cold'] & humidity['high'],
cooling_power['medium'])
rule3 = ctrl.Rule(temperature['comfortable'] &
humidity['comfortable'], cooling_power['medium'])
rule4 = ctrl.Rule(temperature['hot'] & humidity['high'],
cooling_power['high'])
```

```
rule5 = ctrl.Rule(temperature['hot'] & humidity['low'],
cooling_power['low'])

# Create control system
ac_ctrl = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5])
ac_sim = ctrl.ControlSystemSimulation(ac_ctrl)

# Define inputs
ac_sim.input['temperature'] = 75
ac_sim.input['humidity'] = 40

# Compute the cooling power
ac_sim.compute()

# Print the result
print("Cooling Power:", ac_sim.output['cooling_power'])
```

Output Screenshots:



Conclusion (Students should write in their own words):

In conclusion, the fuzzy logic-based decision model offers a flexible approach to controlling an AC, allowing it to adaptively adjust its cooling power based on varying environmental conditions. By considering inputs like temperature and humidity, the model can maintain a comfortable indoor environment efficiently.

Applications:

- **Smart HVAC Systems:** Fuzzy logic can be applied in smart HVAC systems to optimize energy usage and maintain comfort levels in buildings.
- **Energy Efficiency:** By dynamically adjusting cooling power based on environmental conditions, fuzzy logic-based AC control systems can help reduce energy consumption.
- **Indoor Comfort:** The model ensures optimal indoor comfort by taking into account factors like temperature and humidity variations.
- **Fault Tolerance:** Fuzzy logic-based AC control systems can also exhibit fault tolerance by gracefully handling imprecise inputs or sensor failures, ensuring uninterrupted operation.

Batch: MSc AI **Roll No.: 4748118**

Experiment 10 (C)

Title: Write a program to implementation of Fuzzy logic-based decision modelling for Railway

Theory:

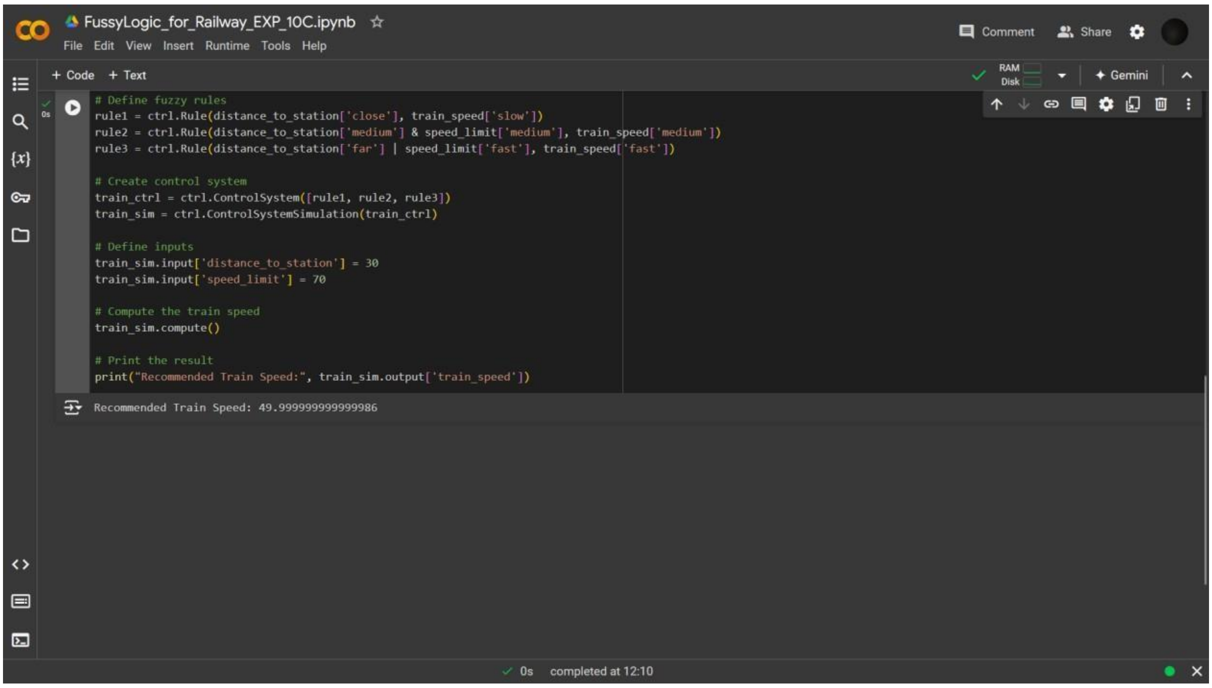
Fuzzy logic is a computational paradigm that deals with approximate reasoning. In the context of railways, fuzzy logic can be utilized for decision-making processes such as train speed control, scheduling, route optimization, and fault detection. It allows for handling imprecise inputs and uncertainties inherent in railway operations.

Implementation (Code):

```
pip install scikit-fuzzy

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

# Define input variables
distance_to_station = ctrl.Antecedent(np.arange(0, 101, 1),
'distance_to_station')
speed_limit = ctrl.Antecedent(np.arange(0, 101, 1), 'speed_limit')

# Define output variable
train_speed = ctrl.Consequent(np.arange(0, 101, 1), 'train_speed')

# Define membership functions
distance_to_station['close'] =
fuzz.trimf(distance_to_station.universe, [0, 0, 50])
distance_to_station['medium'] =
fuzz.trimf(distance_to_station.universe, [0, 50, 100])
distance_to_station['far'] = fuzz.trimf(distance_to_station.universe,
[50, 100, 100])

speed_limit['slow'] = fuzz.trimf(speed_limit.universe, [0, 0, 50])
speed_limit['medium'] = fuzz.trimf(speed_limit.universe, [0, 50, 100])
speed_limit['fast'] = fuzz.trimf(speed_limit.universe, [50, 100, 100])

train_speed['slow'] = fuzz.trimf(train_speed.universe, [0, 0, 50])
train_speed['medium'] = fuzz.trimf(train_speed.universe, [0, 50, 100])
train_speed['fast'] = fuzz.trimf(train_speed.universe, [50, 100, 100])

# Define fuzzy rules
rule1 = ctrl.Rule(distance_to_station['close'], train_speed['slow'])
rule2 = ctrl.Rule(distance_to_station['medium'] &
speed_limit['medium'], train_speed['medium'])
rule3 = ctrl.Rule(distance_to_station['far'] | speed_limit['fast'],
train_speed['fast'])
```

```
# Create control system
train_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
train_sim = ctrl.ControlSystemSimulation(train_ctrl)

# Define inputs
train_sim.input['distance_to_station'] = 30
train_sim.input['speed_limit'] = 70

# Compute the train speed
train_sim.compute()

# Print the result
print("Recommended Train Speed:", train_sim.output['train_speed'])
```

Output Screenshots:



Conclusion (Students should write in their own words):

In conclusion, the fuzzy logic-based decision model provides a robust and flexible approach to train speed control in railway operations. By considering inputs such as distance to the station and speed limits, the model can adaptively adjust the train speed to ensure safety, efficiency, and punctuality in railway transportation.

Applications:

- **Train Traffic Management:** Fuzzy logic can be applied in train traffic management systems to optimize train speeds, improve schedule adherence, and minimize delays.
- **Safety Systems:** Fuzzy logic-based decision models can enhance safety systems by dynamically adjusting train speeds in response to changing environmental conditions or potential hazards.

- **Energy Efficiency:** By optimizing train speeds based on route conditions and traffic density, fuzzy logic models can help reduce energy consumption and environmental impact in railway operations.
- **Fault Detection and Diagnosis:** Fuzzy logic-based models can also be employed for fault detection and diagnosis in railway systems, facilitating proactive maintenance and minimizing service disruptions.

Batch: MSc AI	Roll No.: 4748118
Experiment 10 (D)	

Title: Write a program to implement a Decision tree from scratch

Theory:

A decision tree is a supervised learning algorithm used for both classification and regression tasks. It creates a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

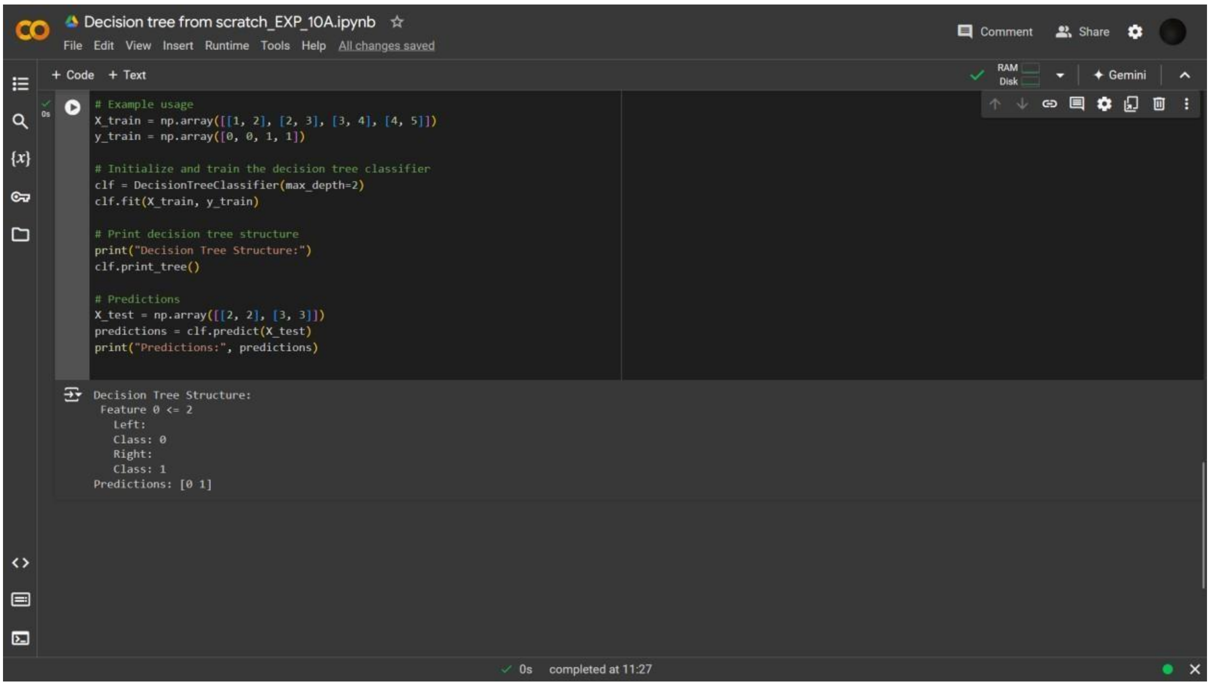
Implementation (Code):

```
import numpy as np

class    DecisionTreeClassifier:
    def_init_(self, max_depth=None):
        self.max_depth = max_depth

    def fit(self, X, y):
        self.tree = self._build_tree(X, y, depth=0)

    def _build_tree(self, X, y, depth):
        num_samples, num_features = X.shape
        num_classes = len(np.unique(y))

        # Stopping criteria
        if (depth == self.max_depth) or (num_classes == 1):
            return np.bincount(y).argmax()

        # Find best split
        best_split = self._find_best_split(X, y)

        if best_split is None:
            return np.bincount(y).argmax()

        feature_idx, threshold = best_split
        left_indices = X[:, feature_idx] <= threshold
        right_indices = ~left_indices

        # Recursively build tree
        left_tree = self._build_tree(X[left_indices], y[left_indices],
depth + 1)
        right_tree = self._build_tree(X[right_indices],
y[right_indices], depth + 1)

        return (feature_idx, threshold, left_tree, right_tree)

    def _find_best_split(self, X, y):
        best_split = None
        best_gini = float('inf')
        num_samples, num_features = X.shape
```

```
        for feature_idx in range(num_features):
            thresholds = np.unique(X[:, feature_idx])
            for threshold in thresholds:
                left_indices = X[:, feature_idx] <= threshold
                right_indices = ~left_indices

                gini = self._calculate_gini_index(y[left_indices],
y[right_indices])
                if gini < best_gini:
                    best_gini = gini
                    best_split = (feature_idx, threshold)

        return best_split

    def _calculate_gini_index(self, left_labels, right_labels):
        total_samples = len(left_labels) + len(right_labels)
        p_left = len(left_labels) / total_samples
        p_right = len(right_labels) / total_samples

        gini_left = 1 - sum([(np.sum(left_labels == c) /
len(left_labels)) ** 2 for c in np.unique(left_labels)])
        gini_right = 1 - sum([(np.sum(right_labels == c) /
len(right_labels)) ** 2 for c in np.unique(right_labels)])

        gini_index = p_left * gini_left + p_right * gini_right
        return gini_index

    def predict(self, X):
        predictions = np.array([self._traverse_tree(x, self.tree) for
x in X])
        return predictions

    def _traverse_tree(self, x, node):
        if isinstance(node, np.int64):
            return node

        feature_idx, threshold, left_tree, right_tree = node
        if x[feature_idx] <= threshold:
            return self._traverse_tree(x, left_tree)
        else:
            return self._traverse_tree(x, right_tree)

    def print_tree(self):
        self._print_node(self.tree)

    def _print_node(self, node, depth=0):
        if isinstance(node, np.int64):
            print(" " * depth, "Class:", node)
        else:
            feature_idx, threshold, left_tree, right_tree = node
            print("  " * depth, f"Feature {feature_idx} <=
{threshold}")
            print(" " * (depth + 1), "Left:")
            self._print_node(left_tree, depth + 1)
            print(" " * (depth + 1), "Right:")
            self._print_node(right_tree, depth + 1)

# Example usage
X_train = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])
y_train = np.array([0, 0, 1, 1])
```

```
# Initialize and train the decision tree classifier
clf = DecisionTreeClassifier(max_depth=2)
clf.fit(X_train, y_train)

# Print decision tree structure
print("Decision Tree Structure:")
clf.print_tree()

# Predictions
X_test = np.array([[2, 2], [3, 3]])
predictions = clf.predict(X_test)
print("Predictions:", predictions)
```

Output Screenshots:



Conclusion (Students should write in their own words):

Implementing a decision tree from scratch provides valuable insights into how decision trees work internally. Through this exercise, we've learned about the recursive process of building a tree by selecting the best splits based on criteria such as Gini impurity. This understanding enhances our grasp of machine learning fundamentals and enables us to appreciate the intricacies of decision-making in predictive modeling.

Applications:

1. **Classification Tasks**: Decision trees are widely used for classification tasks in various fields such as:
   - Spam Detection: Classifying emails as spam or non-spam based on features like keywords and sender information.
   - Medical Diagnosis: Assisting doctors in diagnosing diseases based on patient symptoms and test results.

- Sentiment Analysis: Analyzing text data to classify sentiment as positive, negative, or neutral.

2. **Regression Tasks**: Decision trees are also effective for regression tasks, including:
    - House Price Prediction: Predicting the price of a house based on features such as location, size, and amenities.
    - Demand Forecasting: Forecasting the demand for products or services based on historical sales data and external factors.
    - Financial Modeling: Predicting stock prices or investment returns based on market indicators and economic factors.
3. **Decision Support Systems**: Decision trees are used to build decision support systems that aid decision-making processes in various domains, including:
    - Business Management: Helping businesses make decisions related to resource allocation, marketing strategies, and risk management.
    - Environmental Planning: Supporting decision-making in environmental management and conservation efforts.
    - Customer Relationship Management (CRM): Segmenting customers based on behavior and demographics to personalize marketing campaigns and improve customer satisfaction.