



# Machine Learning SPRING 2023-2024

**SUBMITTED BY:**

**NAME: SUMIYA HUR TASNIM**

**ID: 21-44851-2**

**SECTION: B**

**SUPERVISED BY**

**Taiman Arham Siddique**

**Faculty**

**Submission Date: May 15, 2024**

# Theory

**Q1) Explain how locally weighted regression differs from linear regression, including their formulas. What is an advantage of locally weighted regression over linear regression? [2 points]**

**Ans:**

**Formulas:**

-Linear Regression: This model fits a straight line or hyperplane to the data points. The formula for a linear regression line is given by:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n + \epsilon$$

**Locally Weighted Regression (Loess or Lowess):** This is a non-parametric method that fits multiple regressions in the local neighborhood of each point. The formula is based on a weighting function that decreases the influence of observations as they are farther from the point of interest where the prediction is being made. The prediction  $\hat{y}$  at point  $x$  is calculated as:

$$\hat{y}(x) = \theta^T(x) \cdot x$$

where  $\theta(x)$  is calculated for each point  $x$  using weighted least squares, emphasizing points closer to  $x$ :

$$\theta(x) = (X^T W(x) X)^{-1} X^T W(x) y$$

Here,  $W(x)$  is a diagonal matrix where each diagonal element  $w_{ii}$  is a weight for  $x_i$  derived from a weighting function like the tri-cube function, which depends on the distance of  $x_i$  from  $x$ .

**Advantage of Locally Weighted Regression over Linear Regression:**

- The main advantage of locally weighted regression is its flexibility to model complex relationships without assuming a linear relationship across the entire data set. It adapts to changes in the local structure of the data, providing a more accurate fit, especially in cases where the underlying data relationships vary across different regions of the dataset. This feature makes it superior in handling non-linear relationships, leading to better performance on datasets with local variations in trends.

**Q2) Given you want to apply a model to predict whether a patient has malignant or benign tumor, where model output  $y = 1$  means malignant and  $y = 0$  means benign. Explain how the binary logistic regression model is used to train on patient data and then predict tumor of a new patient. Include formulas and learning algorithm used in your answer. [2 points]**

ANS:

**Model Overview:**

Binary logistic regression is a statistical method used to predict a binary outcome, such as classifying tumors as malignant ( $y = 1$ ) or benign ( $y = 0$ ), based on one or more predictor variables (features). It is a form of regression that models the probability of a binary response based on predictor variables.

**Formulas:**

- **Probability Prediction:** The logistic function is used to model the probability that the dependent variable equals a certain value (e.g., 1 for malignant). The probability  $p$  that  $y = 1$  given input features  $x$  is:

$$p(x) = \frac{1}{1 + e^{-z}}$$

where  $z$  is the linear combination of the input features and the regression coefficients:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n$$

Here,  $\beta_0, \beta_1, \dots, \beta_n$  are the coefficients of the model.

-**Decision Boundary:** A classification decision can be made based on the probability  $p(x)$ . If  $p(x) \geq 0.5$ , the tumor is classified as malignant ( $y = 1$ ); if  $p(x) < 0.5$ , it is classified as benign ( $y = 0$ ).

**Learning Algorithm:**

-**Cost Function:** The parameters (coefficients) of the model are estimated by minimizing the cost function, often implemented as the log-loss (binary cross-entropy) for logistic regression:

$$J(\beta) = -\frac{1}{m} \sum_{i=1}^m [y_i \log(p(x_i)) + (1 - y_i) \log(1 - p(x_i))]$$

where  $m$  is the number of training samples,  $y_i$  is the observed class label for the  $i$ th sample, and  $p(x_i)$  is the predicted probability of the  $i$ th sample being malignant based on the logistic model.

- **Optimization:** This cost function is typically minimized using an optimization algorithm such as gradient descent. In gradient descent, the parameters  $\beta$  are updated iteratively:

$$\beta_j := \beta_j - \alpha \frac{\partial J}{\partial \beta_j}$$

where  $\alpha$  is the learning rate, and  $\frac{\partial J}{\partial \beta_j}$  is the gradient of the cost function with respect to the  $j$ th coefficient.

### **Prediction for a New Patient:**

Once the model is trained, predicting the class of a tumor for a new patient involves:

1. Collecting the new patient's data (features).
2. Applying the logistic regression model to compute the probability  $p(x)$  that the tumor is malignant.
3. Using the decision boundary (e.g.,  $p(x) \geq 0.5$ ) to classify the tumor as malignant or benign.

By following these steps, the model provides a probabilistic assessment that can be used directly or converted into a binary class prediction to aid in medical decision-making.

**Q3.a) Given the output,  $y(n)$ , of 3 training items of softmax regression are represented by the following one-hot vectors where  $y \in \{1,2,3\}$ :  $y_1 = [1 \ 0 \ 0]$ ,  $y_2 = [0 \ 1 \ 0]$  and  $y_3 = [0 \ 0 \ 1]$ . Write the expanded form of the softmax cost function  $J(w)$  for these 3 items, and the softmax output function  $f(x;w)$ . [2 points] b) What is the relationship between softmax and binary logistic regression? [1 point]**

**Ans:**

**Softmax Output Function:** The softmax function outputs a probability distribution over  $K$  classes for a given input vector  $x$ . In the case of three classes as specified, the softmax output function  $f(x;w)$  for an input  $x$  is given by:

$$f(x; w) = \text{softmax}(w^T x) = \left[ \frac{e^{w_1^T x}}{Z}, \frac{e^{w_2^T x}}{Z}, \frac{e^{w_3^T x}}{Z} \right]$$

where  $w_1, w_2, w_3$  are the weight vectors for each class,  $w^T x$  denotes the matrix-vector product of weights and features, and  $Z$  is the normalization constant, defined as:

$$Z = e^{w_1^T x} + e^{w_2^T x} + e^{w_3^T x}$$

The cost function for softmax regression, when dealing with multi-class classification, is typically the cross-entropy loss. For the given one-hot encoded outputs  $y_1=[1,0,0]$ ,  $y_2=[0,1,0]$ , and  $y_3=[0,0,1]$ , the cost function  $J(w)$  for these 3 items is:

$$J(w) = -[y_1 \cdot \log(f(x_1; w)) + y_2 \cdot \log(f(x_2; w)) + y_3 \cdot \log(f(x_3; w))]$$

Expanding this with the one-hot vectors:

$$J(w) = - \left[ \log \left( \frac{e^{w_1^T x_1}}{Z_1} \right) + \log \left( \frac{e^{w_2^T x_2}}{Z_2} \right) + \log \left( \frac{e^{w_3^T x_3}}{Z_3} \right) \right]$$

where  $Z_1 = e^{w_1^T x_1} + e^{w_2^T x_1} + e^{w_3^T x_1}$ ,  $Z_2 = e^{w_1^T x_2} + e^{w_2^T x_2} + e^{w_3^T x_2}$ , and  $Z_3 = e^{w_1^T x_3} + e^{w_2^T x_3} + e^{w_3^T x_3}$ .

### 3.b) What is the relationship between softmax and binary logistic regression? [1 point]

**Ans:**

The relationship between SoftMax regression and binary logistic regression lies in the generalization capabilities of the SoftMax function. Binary logistic regression is a special case of SoftMax regression where the number of classes  $k$  is 2. In binary logistic regression, the output probabilities for the two classes (say, class 1 and class 0) can be modeled as:

$$p(y = 1|x; w) = \frac{1}{1 + e^{-w^T x}}$$

$$p(y = 0|x; w) = 1 - p(y = 1|x; w)$$

In softmax regression, when  $K = 2$ , the softmax function simplifies to:

$$\text{softmax}(w^T x) = \left[ \frac{e^{w_1^T x}}{e^{w_1^T x} + e^{w_2^T x}}, \frac{e^{w_2^T x}}{e^{w_1^T x} + e^{w_2^T x}} \right]$$

This can be seen as directly equivalent to the logistic function, where  $w_1$  and  $w_2$  represent the parameters for the two classes. Thus, softmax regression extends the concept of binary logistic regression to handle multiple classes seamlessly, sharing the same fundamental principles but allowing for classification among more than two categories.

#### Q4) What is the penalty term of ridge/L2 regularization and how does it reduce overfitting? [1 point]

Ans:

The penalty term of ridge regression, also known as L2 regularization, is given by the sum of the squares of the coefficients multiplied by a regularization parameter. Mathematically, the L2 penalty term in the cost function is expressed as:

$$\lambda \sum_{j=1}^n \beta_j^2$$

Here,  $\beta_j$  represents the regression coefficients (excluding the intercept),  $n$  is the number of features, and  $\lambda$  is the regularization parameter, a non-negative hyperparameter that controls the strength of the regularization effect. This term is added to the standard regression cost function, such as the sum of squared residuals in linear regression.

**Reduction of Overfitting:** Ridge regularization helps in reducing overfitting through the following mechanisms:

1. **Shrinking Coefficients:** The L2 penalty shrinks the values of the coefficients towards zero but does not set them exactly to zero. By penalizing the magnitude of the coefficients, ridge regression ensures that the model does not rely too heavily on any single or a small group of features, which might be artifacts of the training data and not true indicators of the underlying patterns.
2. **Bias-Variance Trade-off:** The addition of the regularization term introduces a bias into the estimates by moving them towards zero, but it significantly reduces their variance. This trade-off generally results in a model that performs better on new, unseen data, as it discourages fitting the noise in the training dataset.
3. **Handling Collinearity:** In cases where there is multicollinearity in the data (i.e., high correlations among predictor variables), ridge regression helps stabilize the coefficient estimates. By adding the penalty term, the problem of having a non-invertible (or poorly conditioned) matrix in the normal equation (from linear regression) is mitigated, as the regularization term adds diagonal dominance to the matrix, thereby ensuring invertibility.
4. **Controlling Complexity:** By adjusting  $\lambda$ , the complexity of the model can be controlled. A higher value of  $\lambda$  increases the penalty, leading to simpler models, while a lower  $\lambda$  reduces the impact of the penalty, allowing more complex models. This flexibility allows the model to be tuned to find the optimal balance between fitting the training data and maintaining the ability to generalize to new data.

Overall, ridge regularization is a powerful technique to make regression models more robust and prevent them from overfitting, especially when dealing with large numbers of predictors or correlated features.

**Q5. a) Write the pseudocode/steps of applying Policy iteration to solve an MDP, including the equations. [1 point]**

**Ans:**

Policy iteration is an algorithm used to determine an optimal policy for a Markov Decision Process (MDP). It alternates between two main steps: policy evaluation and policy improvement. Below is the pseudocode and relevant equations for applying policy iteration:

**Pseudocode for Policy Iteration:**

**1. Initialize:**

- Start with an arbitrary policy  $\pi$  (often a random policy).
- Initialize  $V(s)$  arbitrarily for all states  $s$  (commonly set to zero).

**2. Policy Evaluation:**

- Repeat until  $V$  converges:
  - For each state  $s$  in the state space:

$$V(s) \leftarrow \sum_{a \in A(s)} \pi(a|s) \sum_{s', r} P(s', r|s, a) [r + \gamma V(s')]$$

Here,  $P(s', r|s, a)$  is the probability of transitioning to state  $s'$  and receiving reward  $r$  from state  $s$  and action  $a$ ,  $\gamma$  is the discount factor, and  $\pi(a|s)$  is the probability of taking action  $a$  in state  $s$  under policy  $\pi$ .

**Policy Improvement:**

- For each state  $s$ :
  - Find the best action  $a^*$  by:

$$a^* = \arg \max_a \sum_{s', r} P(s', r|s, a) [r + \gamma V(s')]$$

- Update the policy:

$$\pi'(s) \leftarrow a^*$$



- If  $\pi'(s)$  is the same as  $\pi(s)$  for all  $s$ , then stop and return  $\pi$  and  $V$ .
- Otherwise, set  $\pi \leftarrow \pi'$ .

### 5. Repeat Steps 2 and 3:

- Continue alternating between policy evaluation and policy improvement until the policy remains unchanged after an improvement step.

#### Explanation:

- **Policy Evaluation:** Computes the state-value function  $V$  for a fixed policy  $\pi$ . This step involves calculating the expected return starting from each state and following the current policy.
- **Policy Improvement:** Uses the results of the policy evaluation to update the policy by making it greedy with respect to  $V$ . This step involves examining possible actions to determine which action would yield the highest expected return under the current value estimates.

Policy iteration guarantees convergence to the optimal policy and value function, assuming the MDP satisfies standard conditions (finite states and actions, and certain regularity conditions on the transition probabilities and rewards). It effectively balances between evaluation of the current policy and continuous improvement, leading to an optimal solution for decision-making in stochastic environments.

## Q5. b) What is the advantage of using an exploration-based policy like $\epsilon$ -greedy, to solve an MDP? [1 point]

Ans:

**Exploration vs. Exploitation:** The fundamental challenge in solving an MDP, especially through learning algorithms like Q-learning, involves balancing between exploration and exploitation. Exploration involves visiting and discovering new states and actions to understand their effects, while exploitation involves using the known information to maximize rewards.

**$\epsilon$ -Greedy Policy:** An  $\epsilon$ -greedy policy is a common strategy to address this balance. In an  $\epsilon$ -greedy policy:

- With probability  $\epsilon$  (where  $\epsilon$  is a small positive number, typically less than 0.1), an agent selects a random action (exploration).
- With probability  $1-\epsilon$ , the agent selects the best-known action based on current Q-values (exploitation).

**Advantage of  $\epsilon$ -Greedy Policy:**

### 1.Ensures Continuous Exploration:

- **Avoids Local Maxima:** By continually exploring,  $\epsilon$ -greedy prevents the agent from getting stuck in local maxima early in learning. It helps discover potentially better actions that might initially appear suboptimal.
- **State Coverage:** It ensures that over time, every state-action pair is visited multiple times, which is crucial for the accurate estimation of the state-action value function (Q-values). This comprehensive coverage allows the learning algorithm to construct a more accurate model of the environment.

### 2.Balanced Approach:

- **Dynamic Adaptation:** As learning progresses, the need for exploration typically decreases. An  $\epsilon$ -greedy policy can be adapted by reducing  $\epsilon$  over time (annealing), thus shifting focus from exploration to exploitation as the agent becomes more confident in its estimates.
- **Immediate Best Action Utilization:** By exploiting known information most of the time, the  $\epsilon$ -greedy policy ensures that the agent often takes actions that it currently believes are best, leading to higher immediate rewards while still exploring enough to potentially find better options.

### 3.Simplicity and Effectiveness:

- **Easy to Implement:** The  $\epsilon$ -greedy policy is straightforward to implement and does not require complex mechanisms to decide between exploring and exploiting, making it a popular choice in many practical applications.

- **Robust Performance:** This policy has been proven effective across a wide range of environments and scenarios in reinforcement learning, providing a good balance of performance and computational efficiency.

Overall, the  $\epsilon$ -greedy policy is a practical solution that allows an agent to explore the environment sufficiently while exploiting its growing knowledge to make informed decisions. This balance is crucial for the success of learning in complex and uncertain environments characterized by MDPs.

## Q6.a) What makes Q-learning an off-policy algorithm? [1 point]

**Ans: Definition of Off-policy:** In reinforcement learning, an algorithm is described as "off-policy" if it learns the optimal policy independently of the agent's actions. This means the learning process can evaluate the optimal policy while the agent explores the environment using a different, possibly exploratory policy.

**Q-learning as Off-policy:** Q-learning is classified as an off-policy algorithm because it learns the optimal policy even as the agent behaves under a different policy. Here's how Q-learning achieves this:

### 1. Learning the Optimal Policy:

- **Q-value Updates:** In Q-learning, the update rule for the Q-values, which estimate the value of taking a particular action in a particular state, is given by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[ r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Here,  $s_t$  and  $a_t$  are the current state and action,  $r_{t+1}$  is the reward received after taking action  $a_t$ ,  $s_{t+1}$  is the next state,  $\alpha$  is the learning rate, and  $\gamma$  is the discount factor.

### 2. Policy Independence:

- **Action Selection for Learning:** The term  $\max_a Q(s_{t+1}, a)$  represents the maximum Q-value for the next state  $s_{t+1}$ , considering all possible actions. This term signifies the best possible future reward expected from the best action as per the current estimate, independent of the agent's actual chosen action  $a_{t+1}$  in the next state.
- **Agent's Exploratory Policy:** While the Q-value update uses the maximum Q-value to compute the update (indicative of following an optimal policy), the agent itself can follow an exploratory policy like  $\epsilon$ -greedy for action selection. This means the agent may choose suboptimal actions to explore the environment, which are not necessarily the actions with the maximum Q-value.

### 6. Policy Evaluation vs. Policy Improvement:

- Q-learning directly estimates the optimal policy's Q-values (policy evaluation) regardless of the agent's current policy used to interact with the environment. It does not require the current policy to converge or to be stable, making it inherently off-policy.

The off-policy nature of Q-learning allows it to be robust in a variety of settings and to effectively learn optimal policies from exploratory data, making it a powerful tool in fields where obtaining optimal policies directly from optimal behavior is challenging or impractical. This characteristic is crucial for scenarios where safe or cost-effective exploration is not possible, allowing for learning from historical data generated by different policies or even from suboptimal actions.

## 6b) What is the difference between on-policy and off-policy algorithms? [1 point]

Ans:

### On-policy Algorithms:

- **Definition:** On-policy algorithms learn the value of the policy being executed by the agent, meaning the learning process and policy used to make decisions are the same. The algorithm evaluates and improves the policy that the agent enacts while it interacts with the environment.
- **Example:** SARSA (State-Action-Reward-State-Action) is a classic example of an on-policy algorithm. In SARSA, the updates to the action-value function (Q-value) depend on the action taken by the policy currently being improved. The update rule in SARSA reflects this:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

- Here,  $a_{t+1}$  is the action taken according to the current policy at the next state  $s_{t+1}$  indicating the dependency on the policy's own choices.

### Off-policy Algorithms:

- **Definition:** Off-policy algorithms learn the value of the optimal policy independently of the agent's actions. This means the learning algorithm evaluates the optimal policy using data gathered from a policy that can be different from the one it's improving.
- **Example:** Q-learning, as discussed, is an off-policy algorithm. It uses the greedy action (action with the highest current Q-value) for updates regardless of the action actually taken by the policy in use (like an  $\epsilon$ -greedy policy):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

This update rule uses  $\max_a Q(s_{t+1}, a)$ , which represents the best expected reward achievable from the next state, independent of the action actually chosen by the agent's exploration policy.

### Key Differences:

#### 1. Policy Dependency:

- **On-policy:** Learns from the actions decided by the policy it's trying to improve. The value estimate is directly tied to the actions taken by the policy.

- **Off-policy:** Learns potentially from all actions, including those not taken by the current policy, aiming to determine the value of the best possible policy.

## **2.Data Utilization:**

- **On-policy:** Updates depend only on actions and states that are sampled from the specific policy being optimized.
- **Off-policy:** Can utilize data collected from any policy (an older policy, a different policy, or a suboptimal exploratory policy), which is beneficial for learning from a broader range of experiences.

## **3.Flexibility and Efficiency:**

- **On-policy:** Generally simpler and more straightforward but may require more interactions with the environment as it needs to explore and exploit using the same policy.
- **Off-policy:** More complex but provides the flexibility of learning from past or hypothetical decisions, which can lead to faster convergence to the optimal policy if diverse and informative actions are recorded.

Understanding these differences is crucial when selecting the appropriate algorithm for a given reinforcement learning task, depending on the specific requirements for policy dependence and data utilization.

## Implementation (18 points)

**Value iteration [8 points]:** Implement code as needed to solve the value iteration problem in the notebook file value iteration. Instructions to implement your code are given in the notebook file. The notebook file should produce expected results when run with your code.

```
import numpy as np
# Define the size of the environment
environment_rows = 11
environment_columns = 11

# Define rewards
rewards = np.array([
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 1, 1, 1, 0, 3, 3, 3, 3, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 3, 1, 1, 1, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 1, 1, 0, 3, 3, 3],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 1, 1, 1, 0, 3, 3, 3, 3, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 0, 3, 1, 1, 0, 3, 3, 3],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
])

# Initialize the value function
V = np.zeros((environment_rows, environment_columns))
V += rewards # Incorporate rewards into the initial values

# Define actions: up, right, down, left
actions = ['up', 'right', 'down', 'left']

# Determine if a location is a terminal state
def is_terminal_state(current_row_index, current_column_index):
    return rewards[current_row_index, current_column_index] != -1.

# Get the next location based on the chosen action
def get_next_location(current_row_index, current_column_index, action_index):
    new_row_index, new_column_index = current_row_index, current_column_index
    if actions[action_index] == 'up' and current_row_index > 0:
        new_row_index -= 1
    elif actions[action_index] == 'right' and current_column_index < environment_columns - 1:
        new_column_index += 1
    elif actions[action_index] == 'down' and current_row_index < environment_rows - 1:
        new_row_index += 1
```

```

elif actions[action_index] == 'left' and current_column_index > 0:
    new_column_index -= 1
return new_row_index, new_column_index

# Value iteration function
def value_iterations(V, threshold=0.01, discount_factor=0.9):
    number_iterations = 0
    while True:
        oldV = V.copy()
        for row_index in range(environment_rows):
            for column_index in range(environment_columns):
                if not is_terminal_state(row_index, column_index):
                    Q_values = np.zeros(4)
                    for action_index in range(len(actions)):
                        new_row_index, new_column_index =
get_next_location(row_index, column_index, action_index)
                        Q_values[action_index] = rewards[new_row_index,
new_column_index] + discount_factor * V[new_row_index, new_column_index]
                    V[row_index, column_index] = np.max(Q_values)
                number_iterations += 1
            if np.max(np.abs(oldV - V)) < threshold:
                break
    return V, number_iterations

V, number_iterations = value_iterations(V)
print("Value Function:")
print(V)
print("Number of Iterations:")
print(number_iterations)

```

## Output :

```

value_iteration x
E:\Python\python.exe "D:\Uni_Semester\Semester 9\Final\Machine Learning\Final_assignment\value_iteration.py"
Value Function:
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 1. 1. 1. 0. 3. 3. 3. 3.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 3. 1. 1. 1. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 0. 3. 3. 3.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 1. 1. 1. 0. 3. 3. 3. 3.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 0. 3. 1. 1. 0. 3. 3.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
Number of Iterations:
1

```



**Q-Learning [10 points].** Implement code as needed to solve the Q-learning problem in the notebook file q-learning. Instructions to implement your code are given in the notebook file. The notebook file should produce expected results when run with your code.

```
import numpy as np

# Define the shape of the environment (its states)
environment_rows = 11
environment_columns = 11

# Create a 3D numpy array to hold the current Q-values for each state and
action pair
q_values = np.zeros((environment_rows, environment_columns, 4))

# Define actions
# Numeric action codes: 0 = up, 1 = right, 2 = down, 3 = left
actions = ['up', 'right', 'down', 'left']

# Create a 2D numpy array to hold the rewards for each state
rewards = np.full((environment_rows, environment_columns), -100.)
rewards[0, 5] = 100. # Set the reward for the goal to 100

# Define aisle locations (white squares)
aisles = {i: [] for i in range(1, 10)}
aisles[1] = [i for i in range(1, 10)]
aisles[2] = [1, 7, 9]
aisles[3] = [i for i in range(1, 8)] + [9]
aisles[4] = [3, 7]
aisles[5] = [i for i in range(11)]
aisles[6] = [5]
aisles[7] = [i for i in range(1, 10)]
aisles[8] = [3, 7]
aisles[9] = [i for i in range(11)]

# Set the rewards for all aisle locations
for row_index in range(1, 10):
    for column_index in aisles[row_index]:
        rewards[row_index, column_index] = -1.

# Function to check if a location is a terminal state
def is_terminal_state(current_row_index, current_column_index):
    return rewards[current_row_index, current_column_index] != -1.

# Function to get a random non-terminal starting location
def get_starting_location():
    current_row_index, current_column_index =
np.random.randint(environment_rows), np.random.randint(environment_columns)
    while is_terminal_state(current_row_index, current_column_index):
        current_row_index, current_column_index =
np.random.randint(environment_rows), np.random.randint(environment_columns)
    return current_row_index, current_column_index
```

```

# Function for epsilon-greedy policy
def get_next_action(current_row_index, current_column_index, epsilon):
    if np.random.random() < epsilon:
        return np.argmax(q_values[current_row_index, current_column_index])
    else:
        return np.random.randint(4)

# Function to get the next location based on the action
def get_next_location(current_row_index, current_column_index, action_index):
    new_row_index, new_column_index = current_row_index, current_column_index
    if actions[action_index] == 'up' and current_row_index > 0:
        new_row_index -= 1
    elif actions[action_index] == 'right' and current_column_index <
environment_columns - 1:
        new_column_index += 1
    elif actions[action_index] == 'down' and current_row_index <
environment_rows - 1:
        new_row_index += 1
    elif actions[action_index] == 'left' and current_column_index > 0:
        new_column_index -= 1
    return new_row_index, new_column_index

# Function to find the shortest path from any location to the goal
def get_shortest_path(start_row_index, start_column_index):
    if is_terminal_state(start_row_index, start_column_index):
        return []
    else:
        current_row_index, current_column_index = start_row_index,
start_column_index
        shortest_path = []
        while not is_terminal_state(current_row_index, current_column_index):
            action_index = get_next_action(current_row_index,
current_column_index, 1.)
            current_row_index, current_column_index =
get_next_location(current_row_index, current_column_index, action_index)
            shortest_path.append([current_row_index, current_column_index])
        return shortest_path

# Training parameters
epsilon = 0.9
discount_factor = 0.9
learning_rate = 0.9

# Training loop
for episode in range(1000):
    row_index, column_index = get_starting_location()
    while not is_terminal_state(row_index, column_index):
        action_index = get_next_action(row_index, column_index, epsilon)
        old_row_index, old_column_index = row_index, column_index
        row_index, column_index = get_next_location(old_row_index,
old_column_index, action_index)
        reward = rewards[row_index, column_index]
        old_q_value = q_values[old_row_index, old_column_index, action_index]
        temporal_difference = reward + (discount_factor *
np.max(q_values[row_index, column_index])) - old_q_value
        q_values[old_row_index, old_column_index, action_index] +=

```

```

learning_rate * temporal_difference
    epsilon = max(0.1, epsilon * 0.99) # Decay epsilon

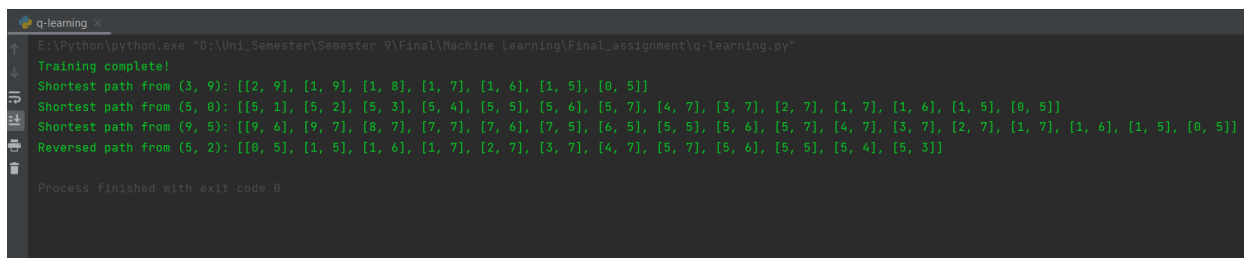
print('Training complete!')

# Display shortest paths
print("Shortest path from (3, 9):", get_shortest_path(3, 9))
print("Shortest path from (5, 0):", get_shortest_path(5, 0))
print("Shortest path from (9, 5):", get_shortest_path(9, 5))

# Display an example of reversed shortest path
path = get_shortest_path(5, 2)
path.reverse()
print("Reversed path from (5, 2):", path)

```

## Output :



```

q-learning x
E:\Python\python.exe "D:\Uni_Semester\Semester 9\Final\Machine Learning\Final_assignment\q-learning.py"
Training complete!
Shortest path from (3, 9): [[2, 9], [1, 9], [1, 8], [1, 7], [1, 6], [1, 5], [0, 5]]
Shortest path from (5, 0): [[5, 1], [5, 2], [5, 3], [5, 4], [5, 5], [5, 6], [5, 7], [4, 7], [3, 7], [2, 7], [1, 7], [1, 6], [1, 5], [0, 5]]
Shortest path from (9, 5): [[9, 6], [9, 7], [8, 7], [7, 7], [7, 6], [7, 5], [6, 5], [5, 5], [5, 6], [5, 7], [4, 7], [3, 7], [2, 7], [1, 7], [1, 6], [1, 5], [0, 5]]
Reversed path from (5, 2): [[0, 5], [1, 5], [1, 6], [1, 7], [2, 7], [3, 7], [4, 7], [5, 7], [5, 6], [5, 5], [5, 4], [5, 3]]

Process finished with exit code 0

```