Data Science Lab

Student PRN:123M1H041

Student Name:DARSHAN SHASHIKANT PATHAK

Submission Date:07/10/2024

Advanced Data Science Lab - Practical Assignment: 5

Part 1: Assignment Based on Supervised Learning - Clustering and Principle Component Analysis

a) K-Means Clustering

What is Clustering?

Clustering is an unsupervised machine learning technique that groups data points based on their similarities. The objective is to partition a dataset into distinct clusters such that data points within the same cluster are more similar to each other than to those in other clusters. Clustering is widely used in applications like customer segmentation, image segmentation, and anomaly detection.

One of the most popular clustering algorithms is **K-Means Clustering**, which works as follows:

- The algorithm partitions data into (k) clusters by initializing (k) cluster centers.
- It assigns each data point to the nearest cluster center, then updates the cluster centers based on the mean of the assigned points.
- This process iterates until the cluster centers stabilize.

In this part, we'll generate synthetic 2D data using the `make_blobs` function from `sklearn.datasets` and apply K-Means clustering to group the data points.

Step 1: **Generate Synthetic Data:**

Using make_blobs to generate a dataset with a specified number of clusters.

```
In [1]: import matplotlib.pyplot as plt
        from sklearn.datasets import make_blobs
        from sklearn.cluster import KMeans

        # Step 1: Generate synthetic data
        X, y_true = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)
```

Step 2: **Implement K-Means Clustering**

Applying the KMeans class from sklearn.cluster to perform clustering.

```
In [2]: # Step 2: Apply K-Means clustering
        kmeans = KMeans(n_clusters=4)
        y_kmeans = kmeans.fit_predict(X)
```

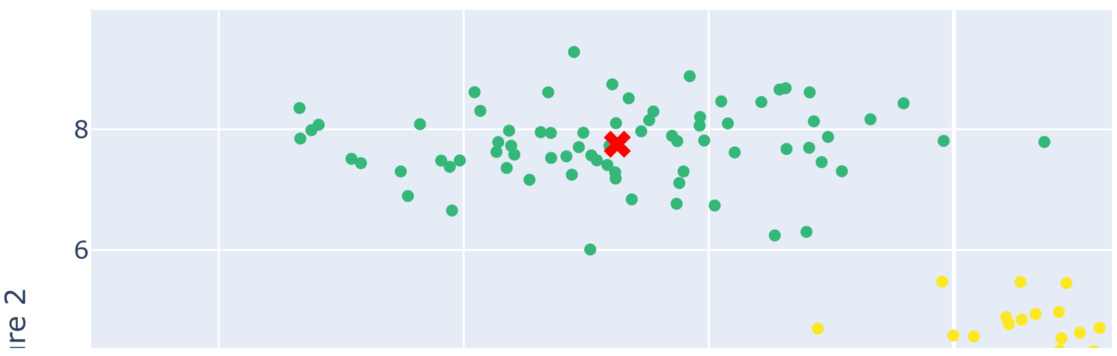
```
C:\Users\Darshan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2k
fra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\cluster\_kmeans.py:
1416: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in
1.4. Set the value of `n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
```

Step 3: **Visualize the Clusters**

Ploting the data points, with each point colored according to its assigned cluster label.

```
In [3]: import plotly.express as px
        from sklearn.datasets import make_blobs
        from sklearn.cluster import KMeans

        # Step 3: Visualize the clusters
        import pandas as pd
        df = pd.DataFrame(X, columns=['Feature 1', 'Feature 2'])
        df['Cluster'] = y_kmeans

        # Step 3: Visualize the clusters with Plotly
        fig = px.scatter(df, x='Feature 1', y='Feature 2', color='Cluster',
                         title="K-Means Clustering",
                         labels={'Feature 1': 'Feature 1', 'Feature 2': 'Feature 2'},
                         color_continuous_scale='Viridis')

        # Add cluster centers
        centers = kmeans.cluster_centers_
        fig.add_scatter(x=centers[:, 0], y=centers[:, 1], mode='markers',
                        marker=dict(size=12, color='red', symbol='x'),
                        name='Cluster Centers')

        fig.show()
```

K-Means Clustering



b) Principal Component Analysis (PCA)

What is PCA?

Principal Component Analysis (PCA) is a dimensionality reduction technique that transforms a dataset with many features into a smaller set of features, called principal components, which capture most of the variance in the data. PCA is widely used for data visualization, noise reduction, and speeding up machine learning algorithms by reducing the number of features.

PCA works by:

- Finding the directions (principal components) that maximize the variance in the data.
- Transforming the data into these new dimensions, ordered by the amount of variance they explain.

In this part, we'll use PCA to reduce the dimensionality of the famous Iris dataset, which consists of four features, to two principal components. This allows us to visualize the data in two dimensions while retaining as much of the original variance as possible.

Step 1: **Loading the Iris Dataset**

Using load_iris from sklearn.datasets to load the data.

In [4]:
```
import plotly.express as px
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
import pandas as pd

# Step 1: Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
species = iris.target_names
```

Step 2: Implementing PCA

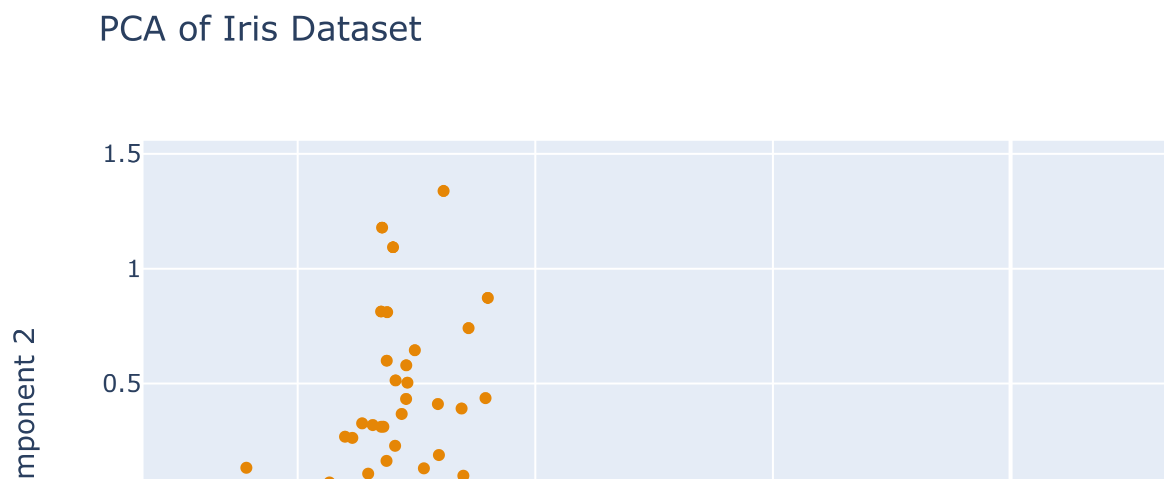
Using PCA from sklearn.decomposition to reduce the dataset to two dimensions.

In [5]:
```
# Step 2: Apply PCA to reduce the data to 2 dimensions
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
```

Step 3: Visualizing the Reduced Data

Ploting the two principal components with coloring to differentiate the species.

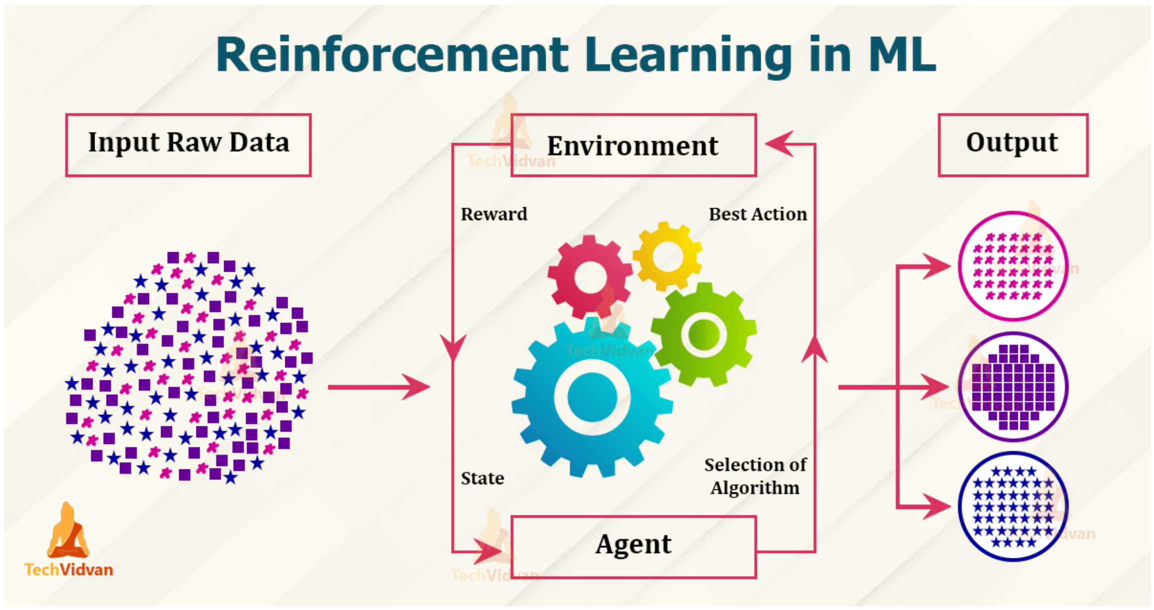
In [6]:
```
# Creating a DataFrame for plotting
df_pca = pd.DataFrame(X_pca, columns=['Principal Component 1', 'Principal Component
df_pca['Species'] = [species[i] for i in y]

# Step 3: Visualize the PCA results with Plotly
fig_pca = px.scatter(df_pca, x='Principal Component 1', y='Principal Component 2',
                     title="PCA of Iris Dataset",
                     labels={'Principal Component 1': 'Principal Component 1',
                             'Principal Component 2': 'Principal Component 2'},
                     color_discrete_sequence=px.colors.qualitative.Vivid)

fig_pca.show()
```



2) Assignments Based on Reinforcement Learning Algorithms

a) Implement the Q-Learning algorithm to train an agent to navigate a simple grid environment. The agent will learn to reach a goal while avoiding obstacles:

You can create a simple 5x5 grid where the agent can move up, down, left, or right. The goal is to reach a specific cell while avoiding obstacles.



Overview of Q-Learning

What is Q-Learning?

Q-Learning is a popular **Reinforcement Learning** (RL) algorithm that enables an agent to learn how to act optimally in a given environment to maximize a reward over time. It is a model-free algorithm, which means it doesn't require a model of the environment. Instead, it learns an optimal policy through interactions with the environment.

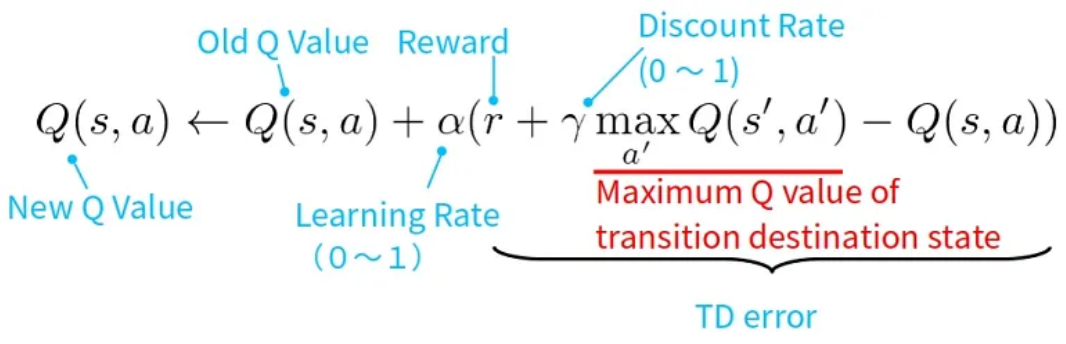
The main components of Q-Learning are:

1. **States (S)**: All possible configurations or positions the agent can be in.
2. **Actions (A)**: All possible actions the agent can take in each state (e.g., move up, down, left, right).
3. **Reward (R)**: A feedback signal given to the agent based on the action taken. Positive rewards encourage the agent to reach its goal, while negative rewards discourage undesired actions.
4. **Q-Table (Q)**: A table that stores the expected future rewards for each state-action pair. This table is updated iteratively as the agent interacts with the environment.

Q-Learning Algorithm:

1. **Initialize** the Q-table with zeros for all state-action pairs.
2. For each episode:
    - Start from an initial state.
    - For each step until the goal or max steps are reached:
        A. **Choose an action** using an exploration strategy (e.g., ε-greedy).
        B. **Take the action** and observe the new state and reward.

   C. **Update the Q-value** for the state-action pair based on the observed reward and estimated future rewards.
   D. **Update the state** to the new state.
3. **Decay** the exploration rate over time to allow the agent to s an optimal path to reach the goal. This approach provides a foundational understanding of Reinforcement Learning and Q-Learning's capabilities in navigating environments.


Old Q Value
Reward
Discount Rate
(0 ~ 1)
$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
New Q Value
Learning Rate
(0 ~ 1)
Maximum Q value of
transition destination state
TD error

Implementation of the Q-Learning Algorithm

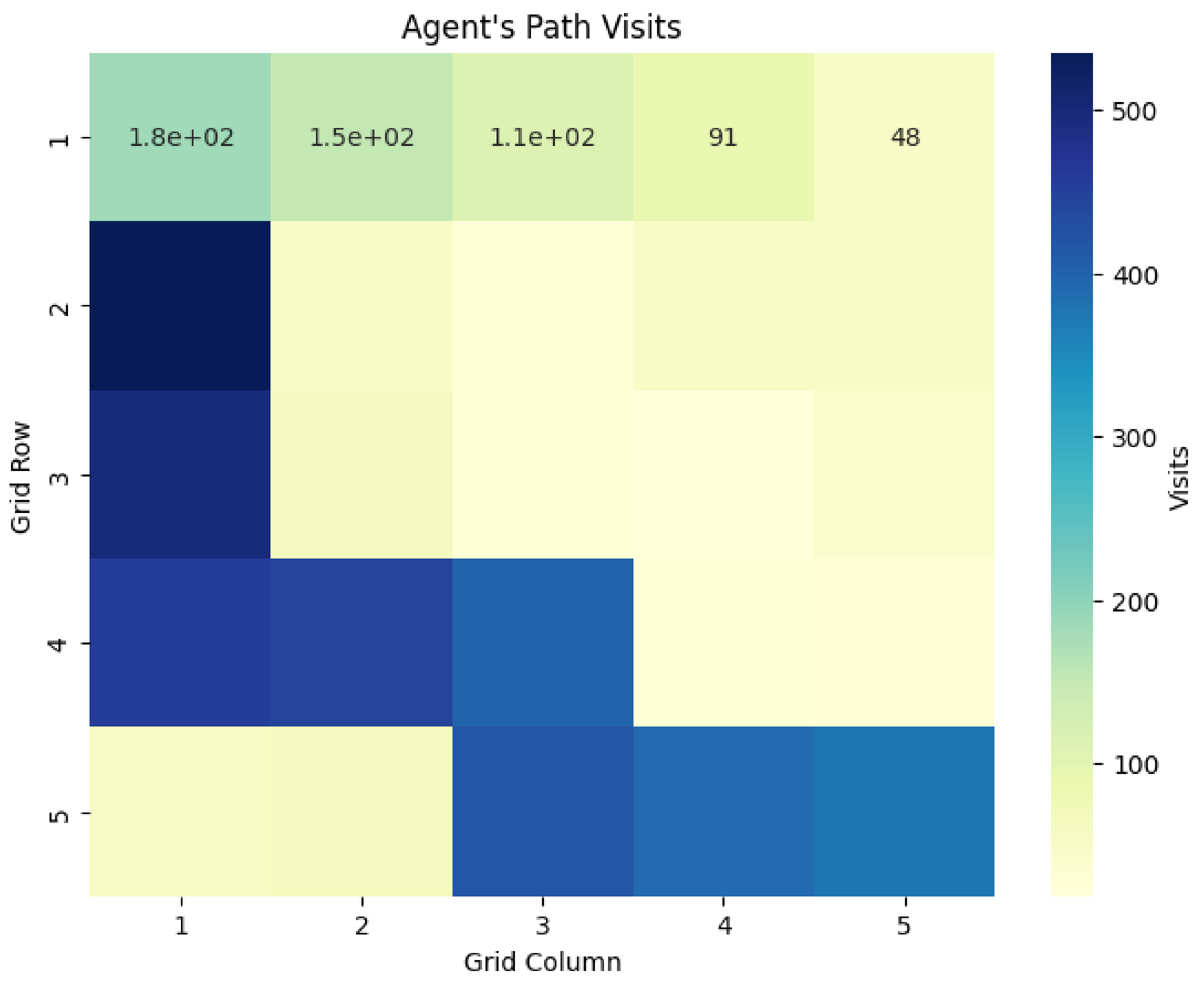
Step-by-Step Guide:

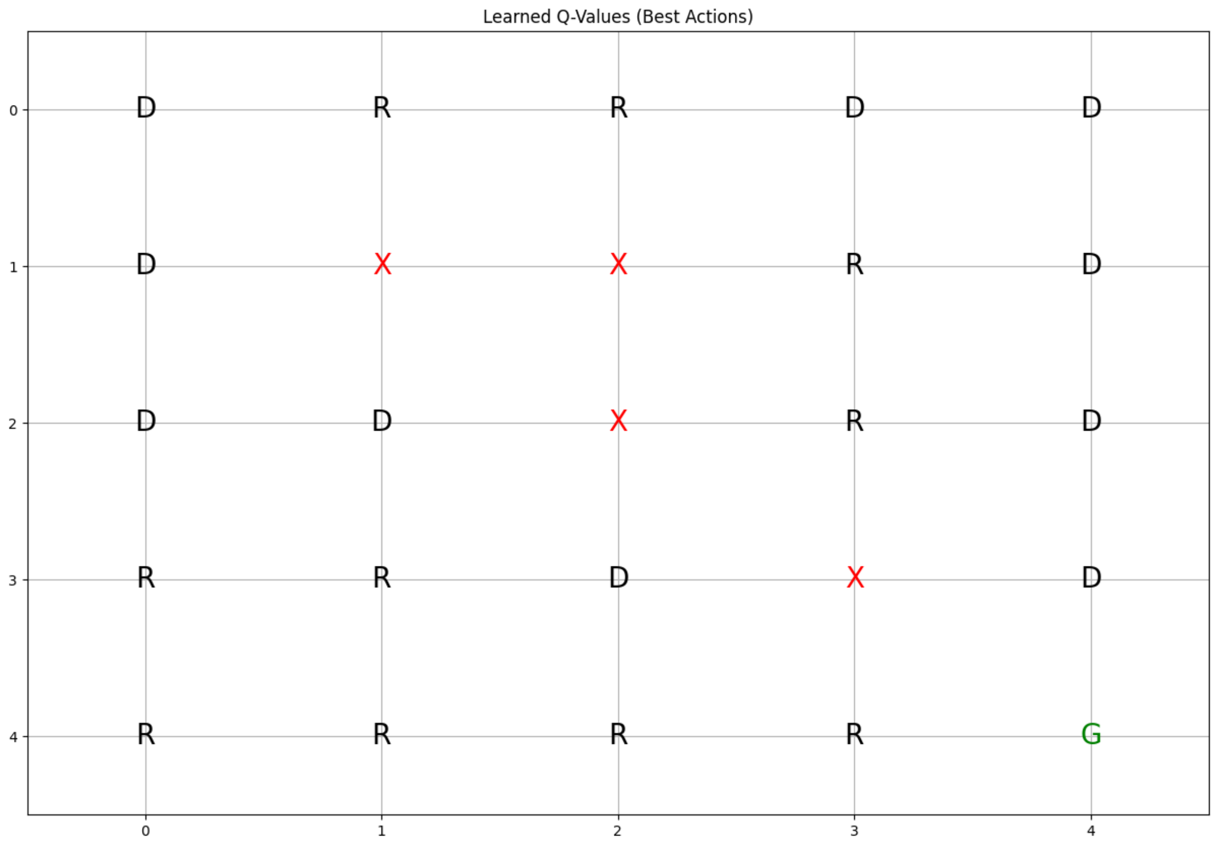
1. **Define the Environment**:
   - We will create a 5x5 grid where the agent can move up, down, left, or right.
   - The grid will include a start position, a goal position, and a few obstacles.
2. **Initialize Parameters**:
   - Q-Table: Initialize with zeros for each state-action pair.
   - Hyperparameters: Set learning rate ((\alpha)), discount factor ((\gamma)), and exploration rate ((\epsilon)).
3. **Implement Q-Learning**:
   - Iterate through episodes, using the Q-Learning update rule to learn an optimal policy.
4. **Visualize the Learned Policy**:
   - Show the path the agent takes from the start to the goal while avoiding obstacles.

Code Implementation:

In [7]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import random
import requests
from PIL import Image
from io import BytesIO
```

```python
# Set up the grid environment
grid_size = 5
goal_position = (4, 4)
obstacles = [(1, 1), (1, 2), (2, 2), (3, 3)]
start_position = (0, 0)

# Define actions
actions = ['up', 'down', 'left', 'right']
n_actions = len(actions)

# Define parameters for Q-learning
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor
epsilon = 1.0  # Exploration rate
epsilon_decay = 0.99
min_epsilon = 0.01
episodes = 500

# Initialize Q-table
Q = np.zeros((grid_size, grid_size, n_actions))

# Reward function
def get_reward(state):
    if state == goal_position:
        return 100  # Reward for reaching the goal
    elif state in obstacles:
        return -10  # Penalty for hitting an obstacle
    else:
        return -1  # Small penalty for each step taken

# Function to get the next position based on the action
def get_next_position(state, action):
    row, col = state
    if action == 'up' and row > 0:
        row -= 1
    elif action == 'down' and row < grid_size - 1:
        row += 1
    elif action == 'left' and col > 0:
        col -= 1
    elif action == 'right' and col < grid_size - 1:
        col += 1
    return (row, col)

# Function to choose an action based on epsilon-greedy policy
def get_action(state):
    if np.random.rand() < epsilon:
        return np.random.choice(actions)  # Explore
    else:
        return actions[np.argmax(Q[state[0], state[1]])]  # Exploit

# Training the agent
agent_positions = []
for episode in range(episodes):
    state = start_position
    done = False
    while not done:
        action = get_action(state)
        next_state = get_next_position(state, action)
        reward = get_reward(next_state)

        # Update Q-value
        action_index = actions.index(action)
        best_next_action = np.max(Q[next_state[0], next_state[1]])
        Q[state[0], state[1], action_index] += alpha * (reward + gamma * best_next_

        # Move to the next state
        state = next_state

        # Store agent's position
        agent_positions.append(state)

        # Check if the episode is done
        if state == goal_position or state in obstacles:
            done = True

    # Decay the exploration rate
    epsilon = max(min_epsilon, epsilon * epsilon_decay)

# Visualize the agent's path
def visualize_path(agent_positions):
    grid = np.zeros((grid_size, grid_size))
    for pos in agent_positions:
        grid[pos] += 1  # Count visits

    plt.figure(figsize=(8, 6))
    sns.heatmap(grid, annot=True, cmap='YlGnBu', cbar_kws={'label': 'Visits'})
    plt.title("Agent's Path Visits")
    plt.xlabel("Grid Column")
    plt.ylabel("Grid Row")
    plt.xticks(ticks=np.arange(0.5, grid_size, 1), labels=np.arange(1, grid_size +
    plt.yticks(ticks=np.arange(0.5, grid_size, 1), labels=np.arange(1, grid_size +
    plt.show()

visualize_path(agent_positions)

# Visualize Q-values
def visualize_q_values(Q):
    plt.figure(figsize=(15, 10))

    for i in range(grid_size):
        for j in range(grid_size):
            if (i, j) == goal_position:
                plt.text(j, i, 'G', ha='center', va='center', fontsize=20, color='g
            elif (i, j) in obstacles:
                plt.text(j, i, 'X', ha='center', va='center', fontsize=20, color='r
            else:
                # Display Q-values for the best action
                best_action_index = np.argmax(Q[i, j])
                action_symbol = actions[best_action_index][0].upper()  # First Lett
                plt.text(j, i, action_symbol, ha='center', va='center', fontsize=20
    plt.title("Learned Q-Values (Best Actions)")
    plt.xlim(-0.5, grid_size - 0.5)
    plt.ylim(grid_size - 0.5, -0.5)
    plt.xticks(np.arange(grid_size))
    plt.yticks(np.arange(grid_size))
    plt.grid()
    plt.show()

visualize_q_values(Q)
```





Explanation:

1. **Environment Setup**:
   - A 5x5 grid environment is defined with a start position, goal position, and obstacles.
2. **Q-Table Initialization**:
   - The Q-Table is initialized with zeros. This table will store the expected future rewards for each state-action pair.
3. **Training Loop**:
   - For each episode, the agent starts at the initial state and explores the environment by choosing actions.
   - The Q-values are updated based on the reward received and the estimated value of the next state.
   - The exploration rate (\epsilon) decays over time, allowing the agent to shift from exploration to exploitation.
4. **Visualization**:
   - The optimal policy (i.e., the best action to take in each state) is visualized. Obstacles, start, and goal positions are marked.

This implementation demonstrates how Q-Learning can be used to train an agent to navigate a simple grid environment while avoiding obstacles. Through iterative learning and updating the Q-values, the agent learns an optimal path to reach the goal. This approach provides a foundational understanding of Reinforcement Learning and Q-Learning's capabilities in navigating environments.
