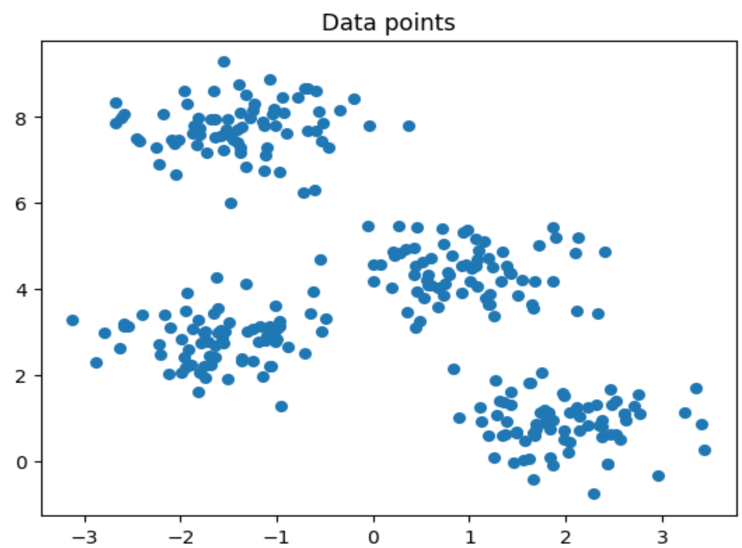
Experiment – 1

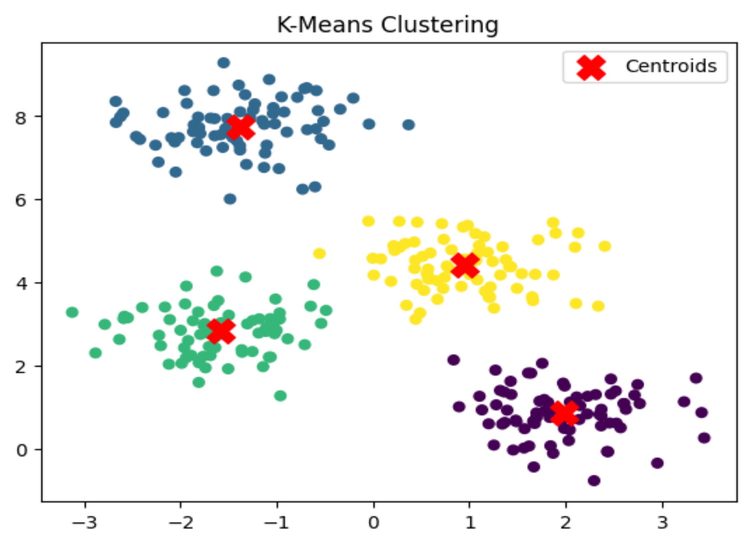
Aim : Implementation of the **K-Means clustering** algorithm.

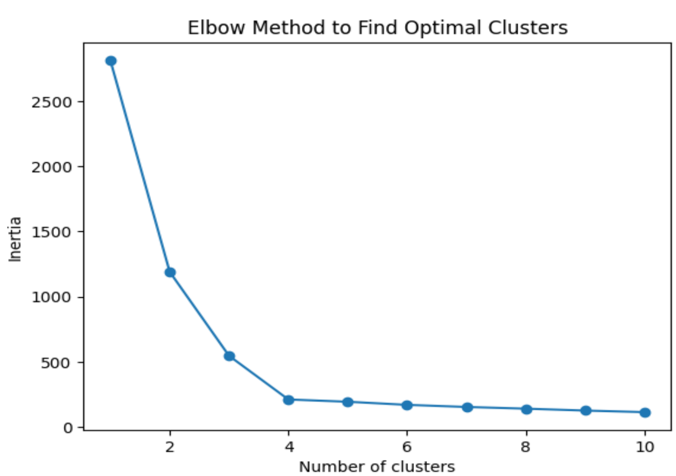
Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

plt.scatter(X[:, 0], X[:, 1], s=30, cmap='viridis')
plt.title("Data points")
plt.show()

kmeans = KMeans(n_clusters=4)
kmeans.fit(X)

centers = kmeans.cluster_centers_

labels = kmeans.labels_

plt.scatter(X[:, 0], X[:, 1], c=labels, s=30, cmap='viridis')
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, marker='X', label='Centroids')
plt.title("K-Means Clustering")
plt.legend()
plt.show()

inertia = []
for i in range(1, 11):
   kmeans = KMeans(n_clusters=i)
   kmeans.fit(X)
   inertia.append(kmeans.inertia_)

plt.plot(range(1, 11), inertia, marker='o')
plt.title("Elbow Method to Find Optimal Clusters")
plt.xlabel("Number of clusters")
plt.ylabel("Inertia")
plt.show()
```

Output:


Data points
8
6
4
2
0
−3
−2
−1
0
1
2
3


K-Means Clustering
Centroids
8
6
4
2
0
−3
−2
−1
0
1
2
3


Elbow Method to Find Optimal Clusters
Inertia
2500
2000
1500
1000
500
0
2
4
6
8
10
Number of clusters

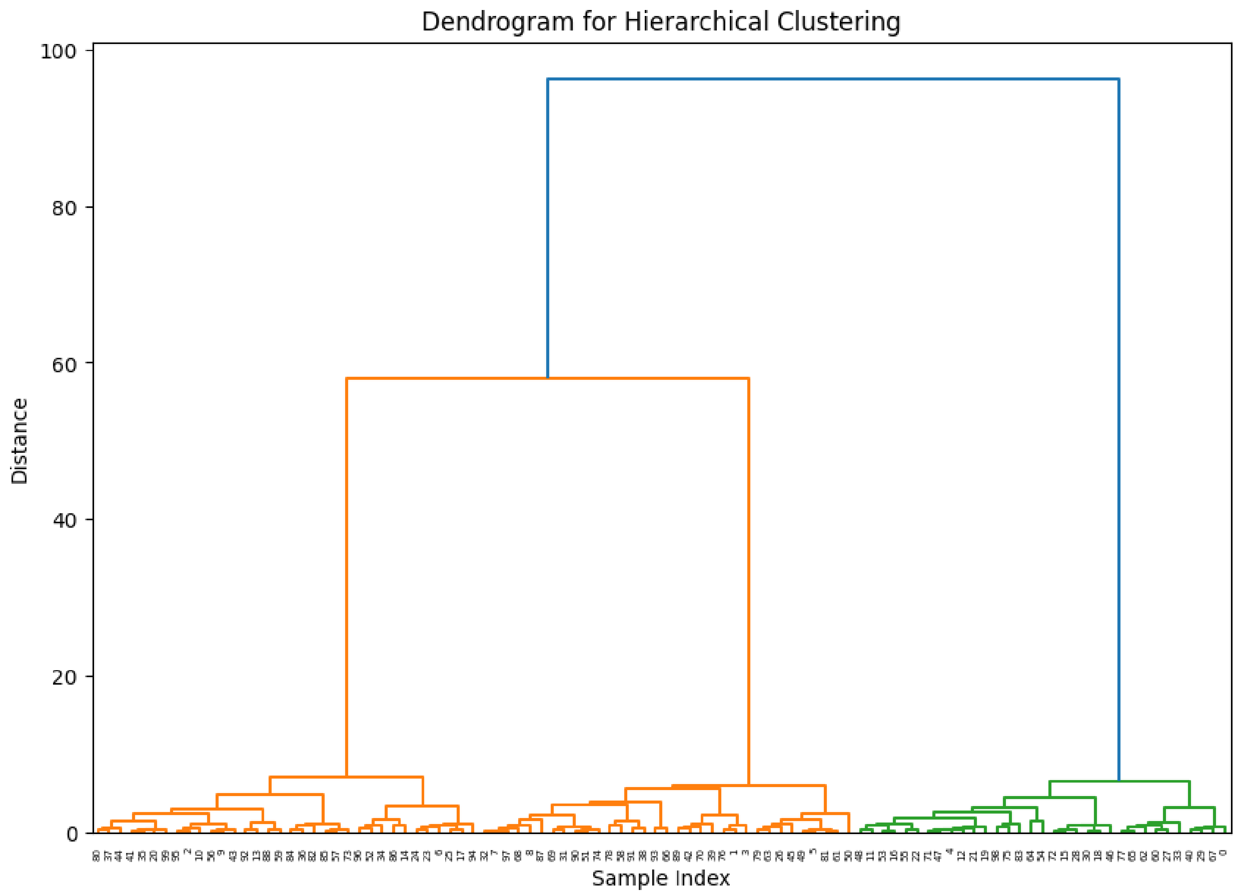
Experiment – 2

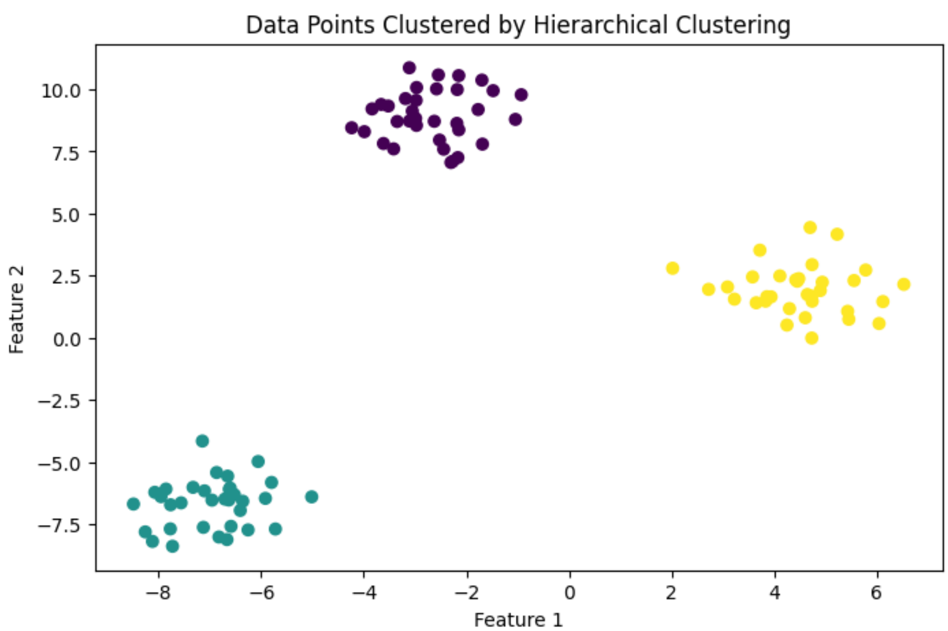
Aim : Write a program to implement the Hierarchical Clustering algorithm.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.datasets import make_blobs
from sklearn.cluster import AgglomerativeClustering

# Generate sample data
X, y = make_blobs(n_samples=100, centers=3, random_state=42, cluster_std=1.0)

# Perform hierarchical clustering using linkage method
linked = linkage(X, method='ward')

# Plot the dendrogram
plt.figure(figsize=(10, 7))
dendrogram(linked, orientation='top', distance_sort='descending', show_leaf_counts=True)
plt.title('Dendrogram for Hierarchical Clustering')
plt.xlabel('Sample Index')
plt.ylabel('Distance')
plt.show()

# Apply Agglomerative Clustering to assign cluster labels
cluster = AgglomerativeClustering(n_clusters=3, affinity='euclidean', linkage='ward')
labels = cluster.fit_predict(X)

# Plot the clustered data
plt.figure(figsize=(8, 5))
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.title('Data Points Clustered by Hierarchical Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

Output:


Dendrogram for Hierarchical Clustering
100
80
60
40
20
0
Distance
Sample Index


Data Points Clustered by Hierarchical Clustering
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
Feature 2
−8
−6
−4
−2
0
2
4
6
Feature 1

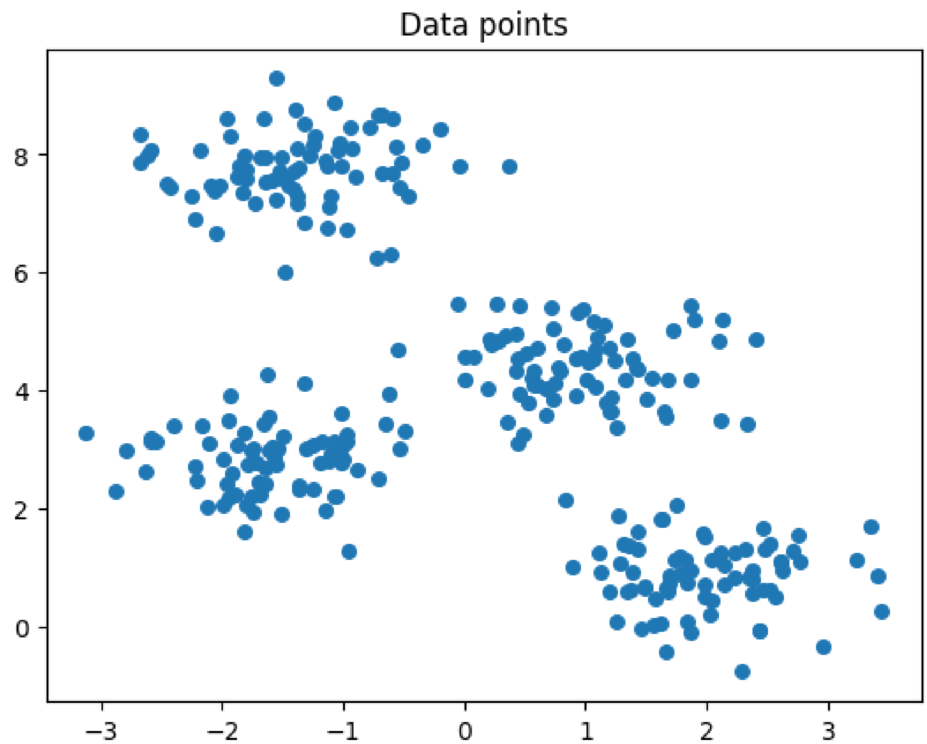
Experiment – 3

Aim : Implementation of the DBScan algorithm.

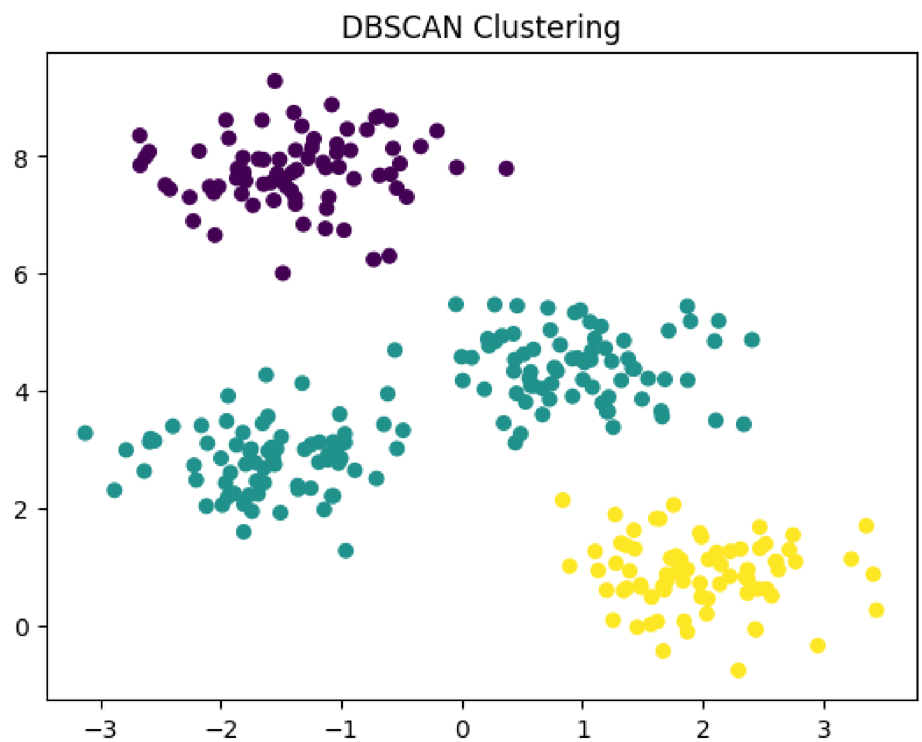
Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs

X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

plt.scatter(X[:, 0], X[:, 1], s=30, cmap='viridis')
plt.title("Data points")
plt.show()

dbscan = DBSCAN(eps=0.8, min_samples=5)
dbscan.fit(X)

labels = dbscan.labels_

plt.scatter(X[:, 0], X[:, 1], c=labels, s=30, cmap='viridis')
plt.title("DBSCAN Clustering")
plt.show()

unique_labels = set(labels)
n_clusters = len(unique_labels) - (1 if -1 in unique_labels else 0)
print(f"Number of clusters: {n_clusters}")
```

Output:



Data points
8
6
4
2
0
−3
−2
−1
0
1
2
3



DBSCAN Clustering
8
6
4
2
0
−3
−2
−1
0
1
2
3

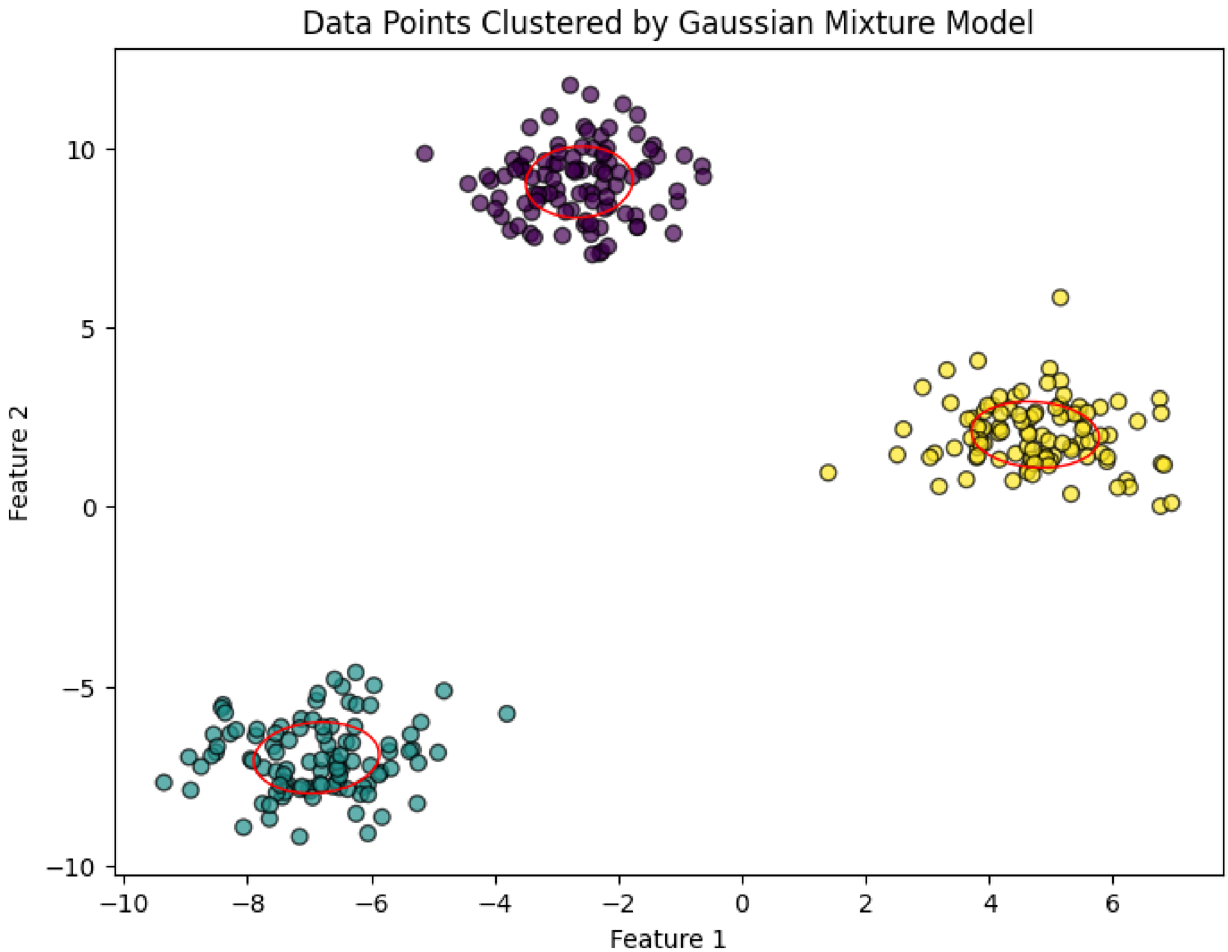
Experiment – 4

Aim : Implementation of the Gaussian Mixture Model.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.mixture import GaussianMixture

# Generate synthetic dataset with clusters
X, y = make_blobs(n_samples=300, centers=3, random_state=42, cluster_std=1.0)

# Apply Gaussian Mixture Model
gmm = GaussianMixture(n_components=3, covariance_type='full', random_state=42)
gmm.fit(X)
labels = gmm.predict(X)

# Plot the clustered data
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', marker='o', s=40, edgecolor='k', alpha=0.7)
plt.title('Data Points Clustered by Gaussian Mixture Model')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')

# Plot the Gaussian ellipses
for i in range(gmm.n_components):
    mean = gmm.means_[i]
    cov = gmm.covariances_[i]
    eigenvalues, eigenvectors = np.linalg.eigh(cov)
    order = eigenvalues.argsort()[::-1]
    eigenvalues, eigenvectors = eigenvalues[order], eigenvectors[:, order]
    angle = np.degrees(np.arctan2(*eigenvectors[:, 0][::-1]))
    width, height = 2 * np.sqrt(eigenvalues)
    ellipse = plt.matplotlib.patches.Ellipse(xy=mean, width=width, height=height, angle=angle, edgecolor='red',
facecolor='none')
    plt.gca().add_patch(ellipse)

plt.show()
```

Output:


Data Points Clustered by Gaussian Mixture Model
Feature 2
10
5
0
−5
−10
−10
−8
−6
−4
−2
0
2
4
6
Feature 1

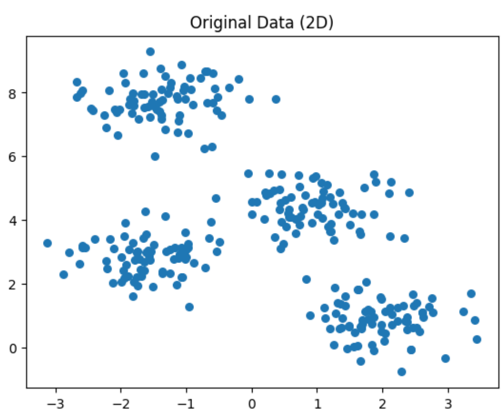
Experiment – 5

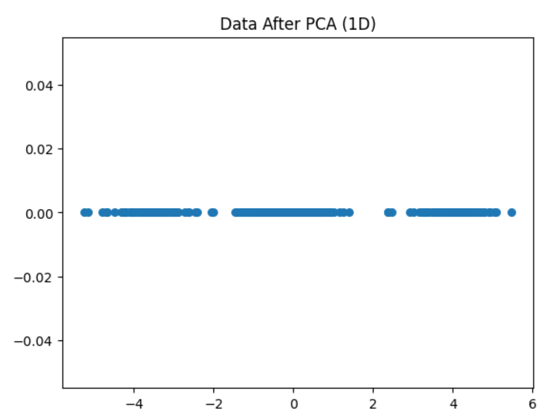
Aim : Implementation of the **PCA** algorithm.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.datasets import make_blobs

X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

plt.scatter(X[:, 0], X[:, 1], s=30, cmap='viridis')
plt.title("Original Data (2D)")
plt.show()

pca = PCA(n_components=1)  # Reducing to 1 component
X_pca = pca.fit_transform(X)

plt.scatter(X_pca, np.zeros_like(X_pca), s=30, cmap='viridis')
plt.title("Data After PCA (1D)")
plt.show()
```

Output:





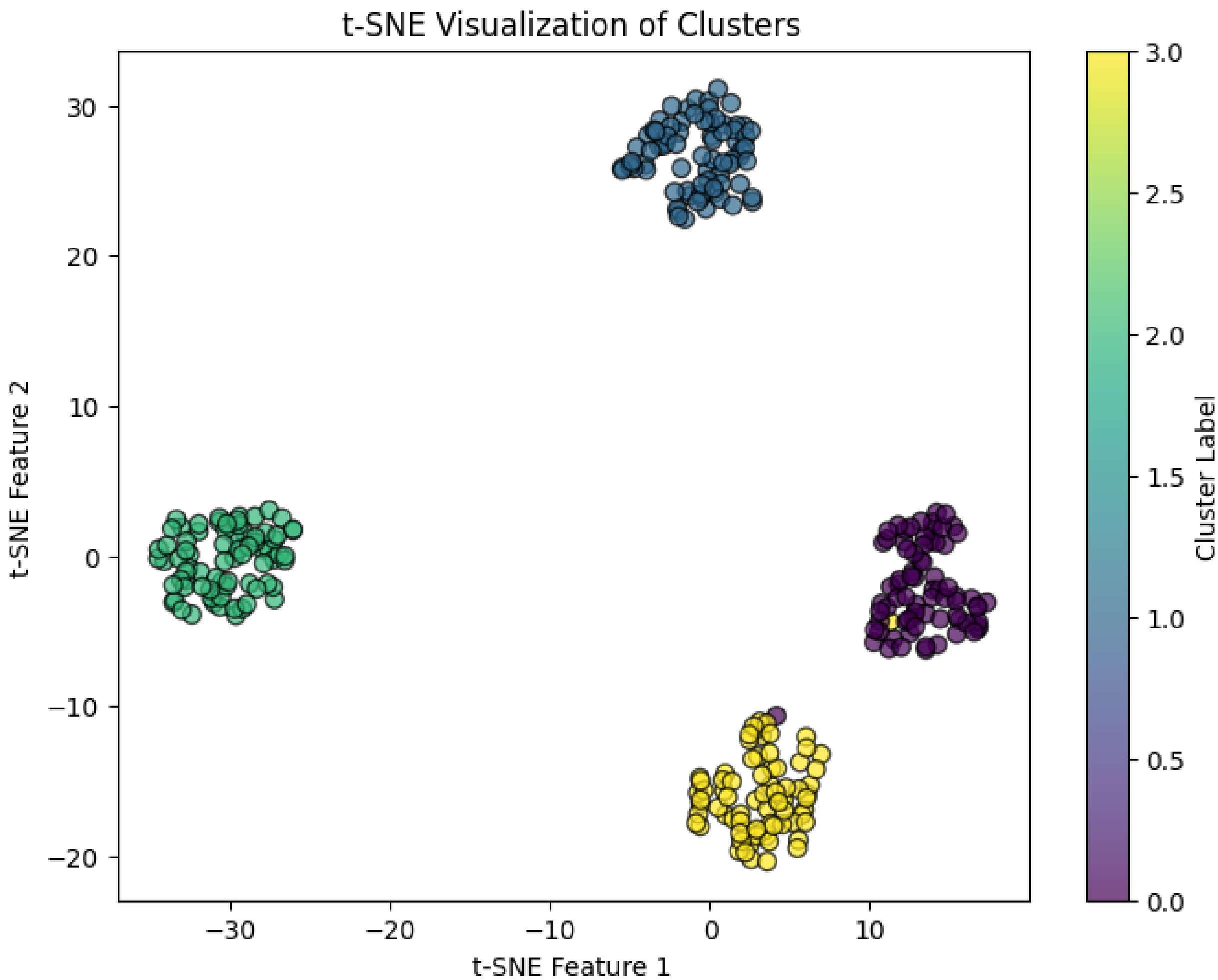
Experiment – 6

Aim : Implementation of the t- Stochastic Neighbor Embedding

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.manifold import TSNE

# Generate synthetic data with clusters
X, y = make_blobs(n_samples=300, centers=4, random_state=42, cluster_std=1.5)

# Apply t-SNE for dimensionality reduction
tsne = TSNE(n_components=2, random_state=42, perplexity=30, n_iter=1000)
X_tsne = tsne.fit_transform(X)

# Plot the t-SNE results
plt.figure(figsize=(8, 6))
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y, cmap='viridis', marker='o', s=50, edgecolor='k', alpha=0.7)
plt.title('t-SNE Visualization of Clusters')
plt.xlabel('t-SNE Feature 1')
plt.ylabel('t-SNE Feature 2')
plt.colorbar(label='Cluster Label')
plt.show()
```

Output:


t-SNE Visualization of Clusters
t-SNE Feature 1
t-SNE Feature 2
Cluster Label
−30
−20
−10
0
10
20
30
3.0
2.5
2.0
1.5
1.0
0.5
0.0

Experiment – 7

Aim : Implementation of the Markov Decision Process.

Code:

```
import numpy as np

grid_size = 4
states = grid_size * grid_size
rewards = np.zeros(states)
rewards[15] = 1
transition_matrix = np.zeros((states, states, 4))
actions = ['up', 'down', 'left', 'right']

# Defining a function to get the next state
def get_next_state(state, action):
    if action == 'up':
        return state - grid_size if state - grid_size >= 0 else state
    if action == 'down':
        return state + grid_size if state + grid_size < states else state
    if action == 'left':
        return state - 1 if state % grid_size != 0 else state
    if action == 'right':
        return state + 1 if state % grid_size != (grid_size - 1) else state
    return state

for state in range(states):
    for action in actions:
        next_state = get_next_state(state, action)
        transition_matrix[state, next_state, actions.index(action)] = 1

gamma = 0.9
V = np.zeros(states)

def value_iteration():
    threshold = 0.0001
    delta = float('inf')

    while delta > threshold:
        delta = 0
        for state in range(states):
            v = V[state]
            max_value = float('-inf')

            for action in actions:
                next_state = get_next_state(state, action)
```

```
            action_value = rewards[next_state] + gamma * V[next_state]
            max_value = max(max_value, action_value)

        V[state] = max_value
        delta = max(delta, abs(v - V[state]))

    return V

V = value_iteration()
print("Optimal Value Function:")
print(V.reshape(grid_size, grid_size))

def extract_policy():
    policy = np.zeros(states, dtype=int)
    for state in range(states):
        best_action_value = float('-inf')
        best_action = None

        for action in actions:
            next_state = get_next_state(state, action)
            action_value = rewards[next_state] + gamma * V[next_state]
            if action_value > best_action_value:
                best_action_value = action_value
                best_action = action

        policy[state] = actions.index(best_action)

    return policy

optimal_policy = extract_policy()

policy_grid = np.chararray((grid_size, grid_size), itemsize=5)
for i in range(grid_size):
    for j in range(grid_size):
        state = i * grid_size + j
        policy_grid[i, j] = actions[optimal_policy[state]]

print("Optimal Policy:")
print(policy_grid)
```

Output:

```
Optimal Value Function:
[[5.90405359 6.56015359 7.28915359 8.09915359]
 [6.56015359 7.28915359 8.09915359 8.99915359]
 [7.28915359 8.09915359 8.99915359 9.99915359]
 [8.09915359 8.99915359 9.99915359 9.99915359]]
Optimal Policy:
[[b'down' b'down' b'down' b'down']
 [b'down' b'down' b'down' b'down']
 [b'down' b'down' b'down' b'down']
 [b'right' b'right' b'right' b'down']]
```

Experiment – 8

Aim : Implementation of the Q-Learning.

Code:

```
import numpy as np
import gym

# Create the FrozenLake environment
env = gym.make("FrozenLake-v1", is_slippery=False)

# Initialize Q-table with zeros
state_size = env.observation_space.n
action_size = env.action_space.n
q_table = np.zeros((state_size, action_size))

# Set Q-learning parameters
learning_rate = 0.8
discount_rate = 0.95
episodes = 10000
max_steps = 100
epsilon = 1.0
max_epsilon = 1.0
min_epsilon = 0.01
decay_rate = 0.005

# Q-learning algorithm
for episode in range(episodes):
    state = env.reset()
    done = False
    for step in range(max_steps):
        # Exploration-exploitation trade-off
        if np.random.rand() < epsilon:
            action = env.action_space.sample()  # Explore
        else:
            action = np.argmax(q_table[state, :])  # Exploit

        # Take action and observe the outcome
        new_state, reward, done, info = env.step(action)

        # Update Q-table using the Q-learning formula
        q_table[state, action] = q_table[state, action] + learning_rate * (
            reward + discount_rate * np.max(q_table[new_state, :]) - q_table[state, action]
        )
```

```
        state = new_state  # Move to the new state

        if done:
            break

    # Decay epsilon to reduce exploration over time
    epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay_rate * episode)

print("Training completed.\n")
print("Q-table:")
print(q_table)

# Test the agent
state = env.reset()
done = False
print("\nTesting the learned policy:")


for step in range(max_steps):
    action = np.argmax(q_table[state, :])
    new_state, reward, done, info = env.step(action)
    state = new_state
    if done:
        if reward == 1:
            print("Goal reached!")
        else:
            print("Fell into a hole.")
        break
```

Output:

```
Training completed.

Q-table:
[[0.73509189 0.77378094 0.6983373  0.73509189]
 [0.73509189 0.         0.66341964 0.69829237]
 [0.69833728 0.42141696 0.         0.63678493]
 [0.53048964 0.         0.         0.        ]
 [0.77378094 0.81450625 0.         0.73509189]
 [0.         0.         0.         0.        ]
 [0.         0.89971597 0.         0.37882205]
 [0.         0.         0.         0.        ]
 [0.81450625 0.         0.857375   0.77378094]
 [0.81450625 0.9025     0.9025     0.        ]
 [0.85737474 0.95       0.         0.82298546]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.9025     0.95       0.857375  ]
 [0.9025     0.95       1.         0.9025    ]
 [0.         0.         0.         0.        ]]

Testing the learned policy:
Goal reached!
```

Experiment – 9

Aim : Implementation of the Policy Gradient Method.

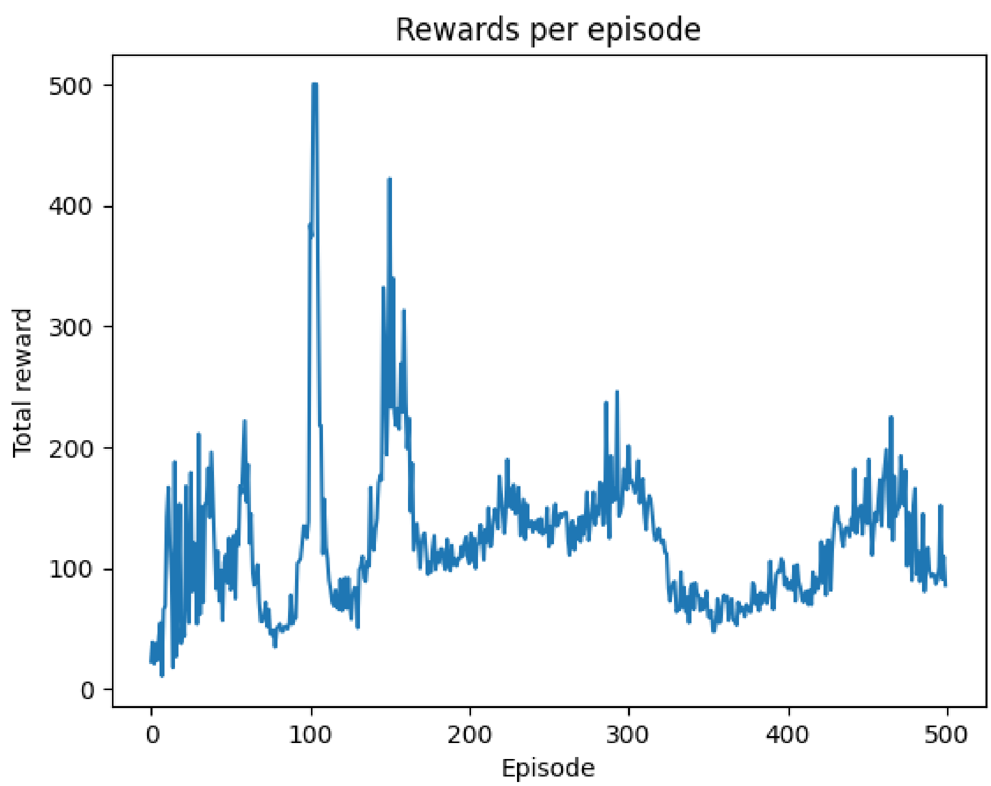
Code:

```
import gym
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

class PolicyNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return self.softmax(x)


env = gym.make('CartPole-v1')
input_size = env.observation_space.shape[0]
hidden_size = 128
output_size = env.action_space.n
policy = PolicyNetwork(input_size, hidden_size, output_size)

optimizer = optim.Adam(policy.parameters(), lr=0.01)
gamma = 0.99

def compute_discounted_rewards(rewards, gamma):
    discounted_rewards = np.zeros_like(rewards, dtype=np.float32)
    running_add = 0
    for t in reversed(range(0, len(rewards))):
        running_add = running_add * gamma + rewards[t]
        discounted_rewards[t] = running_add
    return discounted_rewards

def train_policy_gradient(epochs=500):
    episode_rewards = []
    all_rewards = []
```

```
  for epoch in range(epochs):
    state = env.reset()
    state = torch.FloatTensor(state)
    done = False
    log_probs = []
    rewards = []
    actions = []

    while not done:
      action_probs = policy(state)
      dist = torch.distributions.Categorical(action_probs)
      action = dist.sample()

      next_state, reward, done, _ = env.step(action.item())
      next_state = torch.FloatTensor(next_state)

      log_probs.append(dist.log_prob(action))
      rewards.append(reward)
      actions.append(action.item())

      state = next_state

    discounted_rewards = compute_discounted_rewards(rewards, gamma)
    discounted_rewards = torch.tensor(discounted_rewards)

    loss = 0
    for log_prob, reward in zip(log_probs, discounted_rewards):
      loss += -log_prob * reward

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    episode_rewards.append(np.sum(rewards))
    all_rewards.append(np.sum(rewards))

    if epoch % 10 == 0:
      print(f"Epoch {epoch}/{epochs}, Total Reward: {np.sum(rewards)}")

  return episode_rewards, all_rewards

episode_rewards, all_rewards = train_policy_gradient(epochs=500)
```

```
plt.plot(episode_rewards)
plt.title('Rewards per episode')
plt.xlabel('Episode')
plt.ylabel('Total reward')
plt.show()

def visualize_agent():
    state = env.reset()
    state = torch.FloatTensor(state)
    done = False
    while not done:
        action_probs = policy(state)
        dist = torch.distributions.Categorical(action_probs)
        action = dist.sample()
        next_state, reward, done, _ = env.step(action.item())
        env.render()
        state = torch.FloatTensor(next_state)

    env.close()

visualize_agent()
```

Output:



Experiment – 10

Aim : Implementation of the Actor-Critic architecture.

Code:

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

env = gym.make('CartPole-v1')

actor = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(env.action_space.n, activation='softmax')
])

critic = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(1)
])

actor_optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
critic_optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

# Main training loop
num_episodes = 1000
gamma = 0.99

for episode in range(num_episodes):
    state = env.reset()
    episode_reward = 0

    with tf.GradientTape(persistent=True) as tape:
        for t in range(1, 10000):  # Limit the number of time steps
            # Choose an action using the actor
            action_probs = actor(np.array([state]))
            action = np.random.choice(env.action_space.n, p=action_probs.numpy()[0])

            # Take the chosen action and observe the next state and reward
            next_state, reward, done, _ = env.step(action)

            # Compute the advantage
            state_value = critic(np.array([state]))[0, 0]
            next_state_value = critic(np.array([next_state]))[0, 0]
            advantage = reward + gamma * next_state_value - state_value
```

```
        # Compute actor and critic losses
        actor_loss = -tf.math.log(action_probs[0, action]) * advantage
        critic_loss = tf.square(advantage)

        episode_reward += reward

        # Update actor and critic
        actor_gradients = tape.gradient(actor_loss, actor.trainable_variables)
        critic_gradients = tape.gradient(critic_loss, critic.trainable_variables)
        actor_optimizer.apply_gradients(zip(actor_gradients, actor.trainable_variables))
        critic_optimizer.apply_gradients(zip(critic_gradients, critic.trainable_variables))

        if done:
            break

    if episode % 10 == 0:
        print(f"Episode {episode}, Reward: {episode_reward}")

env.close()
```

Output:

```
Episode 0, Reward: 29.0
Episode 10, Reward: 14.0
Episode 20, Reward: 15.0
Episode 30, Reward: 15.0
Episode 40, Reward: 31.0
Episode 50, Reward: 20.0
Episode 60, Reward: 22.0
Episode 70, Reward: 8.0
Episode 80, Reward: 51.0
Episode 90, Reward: 14.0
Episode 100, Reward: 11.0
Episode 110, Reward: 25.0
Episode 120, Reward: 16.0
....
```