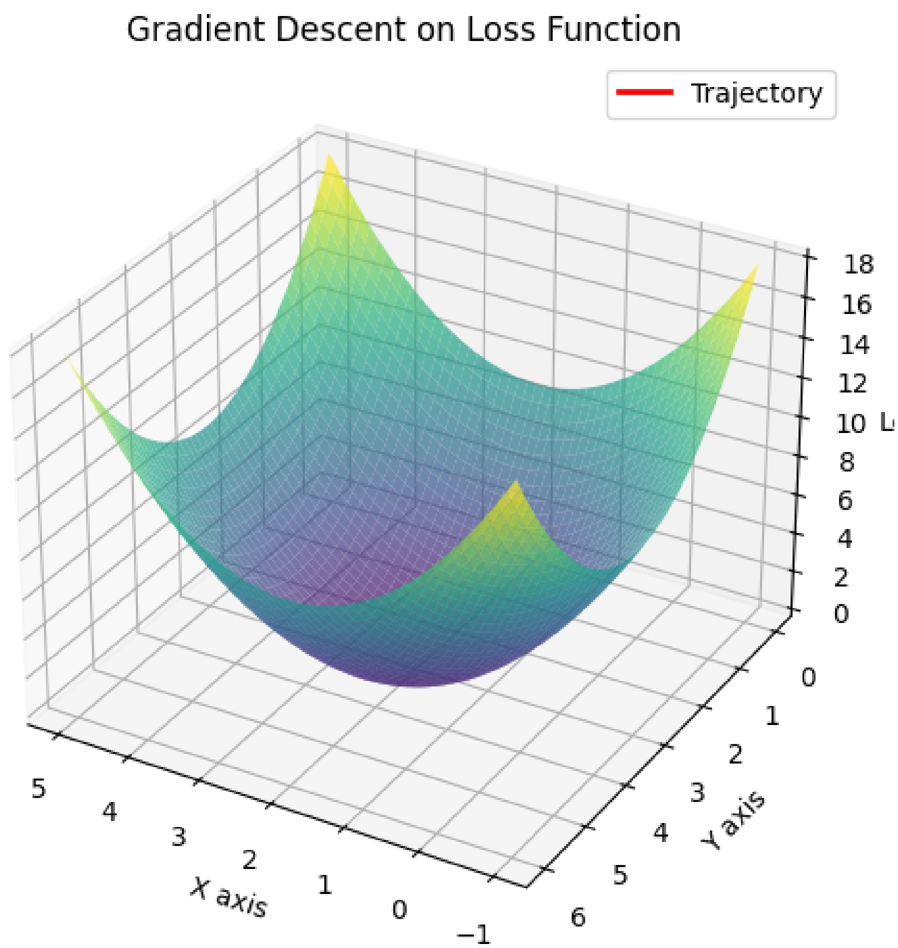
lab4

April 26, 2025

[1]:
```python
''''''''''''''''''''
'''          '''
''''''''''''''''''''

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

#
def loss_function(x, y):
    return (x - 2) ** 2 + (y - 3) ** 2

#
def gradient(x, y):
    partial_x = 2 * (x - 2)
    partial_y = 2 * (y - 3)
    return partial_x, partial_y

#
def gradient_descent(x, y, learning_rate=0.1, num_iterations=50):
    trajectory = [(x, y, loss_function(x, y))]
    for _ in range(num_iterations):
        grad_x, grad_y = gradient(x, y)
        x -= learning_rate * grad_x
        y -= learning_rate * grad_y
        trajectory.append((x, y, loss_function(x, y)))
    return np.array(trajectory)

#
x, y =  0, 0    #
lr = 0.1        #
num_iters = 50  #
trajectory = gradient_descent(x, y, learning_rate=lr, num_iterations=num_iters)

#
x_vals = np.linspace(-1, 5, 400) #
y_vals = np.linspace(0, 6, 400)
```

```python
X, Y = np.meshgrid(x_vals, y_vals)
Z = loss_function(X, Y)

#  3D
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection="3d")
ax.plot_surface(X, Y, Z, cmap="viridis", alpha=0.7)

#
ax.set_xlabel("X axis")
ax.set_ylabel("Y axis")
ax.set_zlabel("Loss")
ax.set_title("Gradient Descent on Loss Function")

#
ax.view_init(elev=30, azim=120)
#
(trajectory_line,) = ax.plot([], [], [], color="red", linewidth=2, label="Trajectory")
(point,) = ax.plot([], [], [], "ro")  #

#
def update(frame):
    trajectory_line.set_data(trajectory[: frame + 1, 0], trajectory[: frame + 1, 1])
    trajectory_line.set_3d_properties(trajectory[: frame + 1, 2])

    #
    point.set_data([trajectory[frame, 0]], [trajectory[frame, 1]])
    point.set_3d_properties([trajectory[frame, 2]])

    #
    ax.view_init(elev=30, azim=120 - frame * 2) #

    return trajectory_line, point

'''
trajectory_line.set_data(trajectory[:, 0], trajectory[:, 1])
trajectory_line.set_3d_properties(trajectory[:, 2])
point.set_data([trajectory[-1, 0]], [trajectory[-1, 1]])
point.set_3d_properties([trajectory[-1, 2]])
#
ax.view_init(elev=30, azim=120)
'''


#
```

```
ani = animation.FuncAnimation(fig, update, frames=len(trajectory),
    interval=200, blit=False)
ax.legend()
plt.show()
#           jupyter notebook
#  ani   warning     cell
```

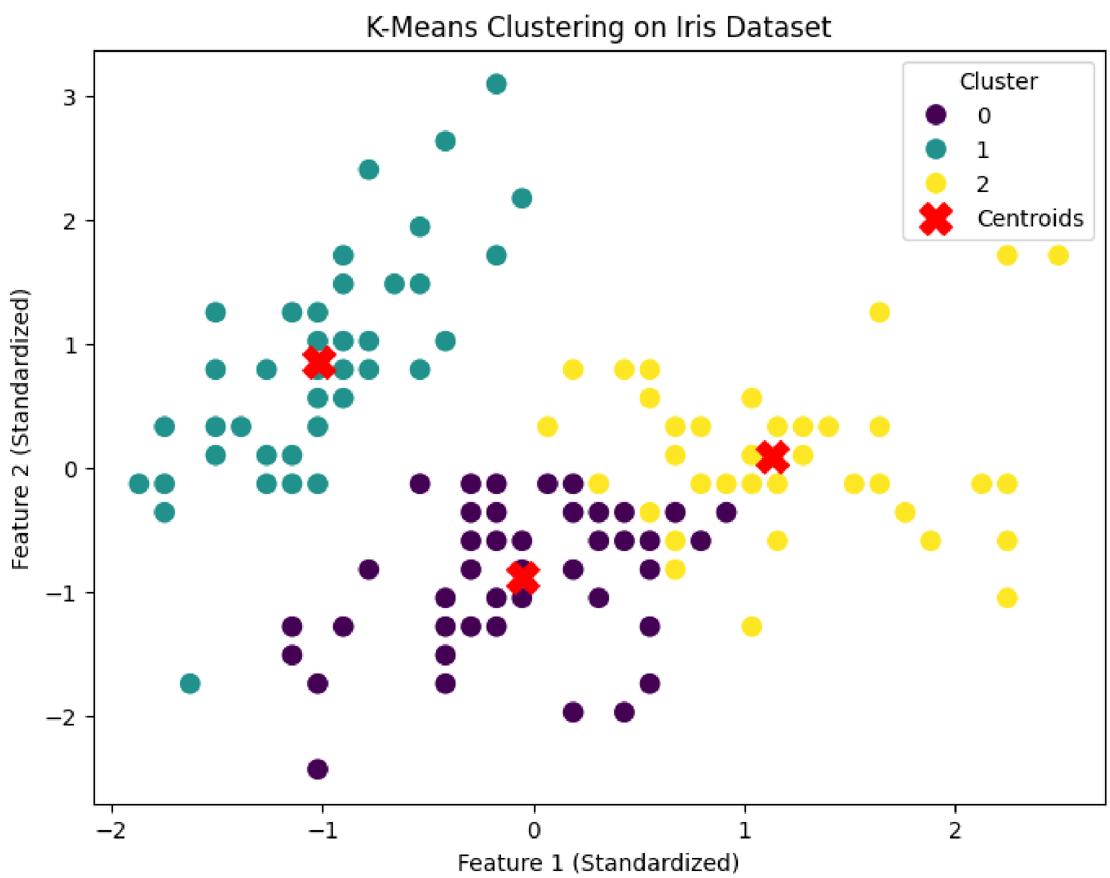


[2]:
```
''''''''''''''''''''
'''         '''
''''''''''''''''''''
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score
```

```
from scipy.stats import mode

#
iris = datasets.load_iris()
X = iris.data #
y = iris.target #

#
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

#    K-Means
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
kmeans.fit(X_scaled)

#
cluster_labels = kmeans.labels_

#
#  K-Means
mapped_labels = np.zeros_like(cluster_labels)
for i in range(3): #
    mask = cluster_labels == i
    mapped_labels[mask] = mode(y[mask], keepdims=True).mode[0]
    #

#
accuracy = accuracy_score(y, mapped_labels)
print(f"Accuracy of K-Means clustering: {accuracy:.2f}")

#
unique, counts = np.unique(cluster_labels, return_counts=True)
cluster_composition = dict(zip(unique, counts))
print("Cluster Composition:", cluster_composition)

#
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_scaled[:, 0], y=X_scaled[:, 1], hue=cluster_labels,␣
 ↪palette="viridis", s=100)

plt.scatter(
    kmeans.cluster_centers_[:, 0],
    kmeans.cluster_centers_[:, 1],
    c="red",
    marker="X",
    s=200,
    label="Centroids"
```

```
)
plt.title("K-Means Clustering on Iris Dataset")
plt.xlabel("Feature 1 (Standardized)")
plt.ylabel("Feature 2 (Standardized)")
plt.legend(title="Cluster")
plt.show()

#
sample_data = np.array([[5.1, 3.5, 1.4, 0.2]])
 #
sample_scaled = scaler.transform(sample_data)
predicted_cluster = kmeans.predict(sample_scaled)

print(f"The sample data {sample_data} is predicted to belong to cluster␣
 ↪{predicted_cluster[0]}")
```

```
Accuracy of K-Means clustering: 0.83
Cluster Composition: {np.int32(0): np.int64(53), np.int32(1): np.int64(50),
np.int32(2): np.int64(47)}
```



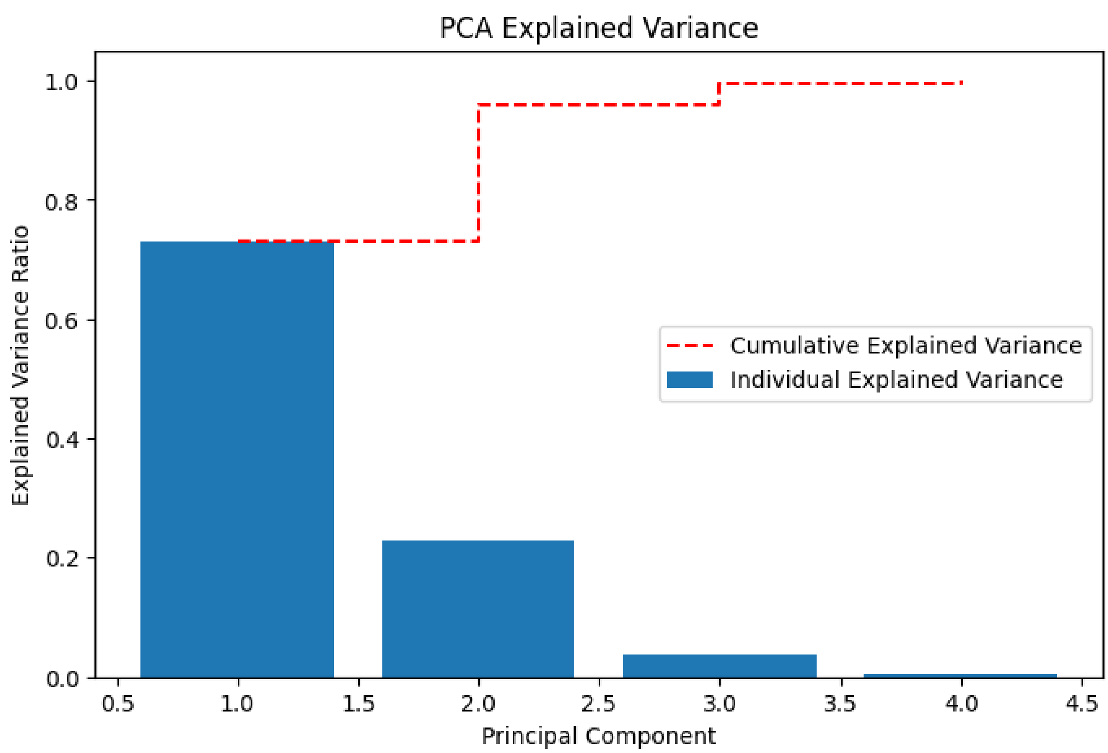
```
The sample data [[5.1 3.5 1.4 0.2]] is predicted to belong to cluster 1
```

[4]:

```
''''''''''''''''''''
'''         '''
''''''''''''''''''''

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

#
iris = datasets.load_iris()
X = iris.data #
y = iris.target #

#       PCA
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

#    PCA
pca = PCA(n_components=4) #   4
X_pca = pca.fit_transform(X_scaled)

#
explained_variance = pca.explained_variance_ratio_  #
cumulative_variance = np.cumsum(explained_variance)  #
#
for i, variance in enumerate(explained_variance):
    print(f"Principal Component {i+1}: {variance:.2%} variance explained")

#
plt.figure(figsize=(8, 5))
plt.bar(
    range(1, len(explained_variance) + 1),
    explained_variance,
    align="center",
    label="Individual Explained Variance",
)
plt.step(
    range(1, len(cumulative_variance) + 1),
    cumulative_variance,
    where="post",
    linestyle="--",
    color="red",
```

```
    label="Cumulative Explained Variance",
)
plt.xlabel("Principal Component")
plt.ylabel("Explained Variance Ratio")
plt.title("PCA Explained Variance")
plt.legend()
plt.show()
```

```
Principal Component 1: 72.96% variance explained
Principal Component 2: 22.85% variance explained
Principal Component 3: 3.67% variance explained
Principal Component 4: 0.52% variance explained
```



[ ]:
```
#
def box_packing(boxes, capacity):
    """


        boxes:
        capacity:



    """
```

```python
    boxes.sort(reverse=True)  #
    bins = []
    for box in boxes:
        placed = False
        for b in bins:
            if sum(b) + box <= capacity:
                b.append(box)
                placed = True
                break
        if not placed:
            bins.append([box])
    return len(bins)

#
boxes = [4, 8, 1, 4, 2, 1]
capacity = 10
print(f"     : {box_packing(boxes, capacity)}")
```

```
     : 2
```

[ ]:
```python
#
def fibonacci_dp(n):
    """



        n:   n


          n
    """
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]

#
n = 10
print(f"     {n}   : {fibonacci_dp(n)}")
```

```
    10   : 55
```

[ ]:
```python
#
from collections import deque

def eight_puzzle(start, goal):
    """
```

```
        start:
        goal:


    """
    def neighbors(state):
        idx = state.index(0)
        x, y = divmod(idx, 3)
        moves = []
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                nidx = nx * 3 + ny
                new_state = list(state)
                new_state[idx], new_state[nidx] = new_state[nidx],␣
↪new_state[idx]
                moves.append(tuple(new_state))
        return moves

    queue = deque([(start, 0)])
    visited = set()
    visited.add(start)

    while queue:
        state, steps = queue.popleft()
        if state == goal:
            return steps
        for neighbor in neighbors(state):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, steps + 1))

#
start = (1, 2, 3, 4, 5, 6, 7, 8, 0)
goal = (1, 2, 3, 4, 5, 6, 0, 7, 8)
print(f"     : {eight_puzzle(start, goal)}")
```

    : 2

[ ]:
```
#     n
def solve_n_queens(n):
    """
        n

        n:
```

```python
    """
    def is_valid(board, row, col):
        for i in range(row):
            if board[i] == col or abs(board[i] - col) == abs(i - row):
                return False
        return True

    def backtrack(row):
        if row == n:
            solutions.append(board[:])
            return
        for col in range(n):
            if is_valid(board, row, col):
                board[row] = col
                backtrack(row + 1)
                board[row] = -1

    solutions = []
    board = [-1] * n
    backtrack(0)
    return solutions

#
n = 8
solutions = solve_n_queens(n)
print(f"{n}     {len(solutions)}  ")
```

```
8      92
```