CODE 1

To implement **K-Means clustering** or **Hierarchical Clustering** on the `sales_data_sample.csv` dataset and determine the optimal number of clusters using the **Elbow Method**, follow the steps below.

Steps for K-Means or Hierarchical Clustering:

1. **Import Libraries and Load the Dataset**:
   We will use `pandas` for data manipulation, `matplotlib` and `seaborn` for plotting, and `sklearn` for clustering algorithms.
2. **Preprocess the Data**:
   Clean the data, handle missing values, and select features to perform clustering.
3. **Determine the Optimal Number of Clusters** using the **Elbow Method**.
4. **Apply K-Means or Hierarchical Clustering**:
   We will apply K-Means clustering or Hierarchical clustering based on the determined number of clusters.
5. **Evaluate the Clusters**:
   Visualize the clusters and evaluate them based on the results.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from scipy.cluster.hierarchy import dendrogram, linkage

# Step 1: Load the dataset
url = "path_to_your_downloaded/sales_data_sample.csv"  # Update with correct path
df = pd.read_csv(url)

# Step 2: Explore the dataset
print(df.head())

# Step 3: Preprocess the data
# Check for missing values and drop rows or fill them
print(df.isnull().sum())
df = df.dropna()  # For simplicity, drop rows with missing values

# Select relevant features for clustering (e.g., sales and profit columns)
features = df[['Sales', 'Profit']]  # Modify based on available columns
```

```
# Normalize the data (important for K-Means)
scaler = StandardScaler()
features_scaled = scaler.fit_transform(features)

# Step 4: Determine the optimal number of clusters using the Elbow Method
inertia = []  # To store the sum of squared distances for different k values
k_range = range(1, 11)  # Check k values from 1 to 10

for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(features_scaled)
    inertia.append(kmeans.inertia_)

# Plot the Elbow Method graph
plt.figure(figsize=(8, 6))
plt.plot(k_range, inertia, marker='o', color='b')
plt.title('Elbow Method For Optimal k')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia (Sum of Squared Distances)')
plt.show()

# Step 5: Apply K-Means with the optimal number of clusters (based on the elbow plot)
# From the elbow plot, choose the value of k where the inertia starts to level off
optimal_k = 4  # Example, but you should choose based on the plot
kmeans = KMeans(n_clusters=optimal_k, random_state=42)
df['Cluster'] = kmeans.fit_predict(features_scaled)

# Step 6: Visualize the clusters
plt.figure(figsize=(8, 6))
sns.scatterplot(x='Sales', y='Profit', hue='Cluster', data=df, palette='Set2')
plt.title(f"K-Means Clustering (k={optimal_k})")
plt.show()

# Step 7: Evaluate the clusters
print(df.groupby('Cluster').mean())  # Cluster centroids for each feature
```

Explanation of the Steps:

1. **Dataset Loading and Preprocessing**:
    - We load the `sales_data_sample.csv` dataset using `pandas`.
    - We check for missing values using `df.isnull().sum()` and handle them. In this example, we drop rows with missing values for simplicity.

    - We select the features relevant for clustering, such as `Sales` and `Profit`.
2. **Scaling the Data**:
    - We normalize the selected features (`Sales` and `Profit`) using `StandardScaler`. This ensures that all features contribute equally to the distance computation in K-Means.
3. **Elbow Method**:
    - The **Elbow Method** is used to determine the optimal number of clusters. We plot the **inertia** (sum of squared distances of samples to their closest cluster center) for different values of `k` and look for the "elbow" point where the inertia starts decreasing at a slower rate.
4. **K-Means Clustering**:
    - We apply K-Means clustering with the optimal number of clusters determined from the elbow plot. We assign each data point to a cluster and store the cluster labels in a new column (`Cluster`).
5. **Visualization**:
    - We visualize the clusters on a scatter plot of `Sales` vs. `Profit`. Each cluster is assigned a different color.
6. **Cluster Evaluation**:
    - We print the mean values of `Sales` and `Profit` for each cluster to understand the characteristics of each cluster.

```
from scipy.cluster.hierarchy import dendrogram, linkage

# Step 1: Apply Hierarchical Clustering
Z = linkage(features_scaled, method='ward')  # Ward's method is typically used for minimizing variance

# Step 2: Plot the Dendrogram
plt.figure(figsize=(10, 7))
dendrogram(Z)
plt.title('Dendrogram for Hierarchical Clustering')
plt.xlabel('Samples')
plt.ylabel('Distance')
plt.show()

# Step 3: Apply Agglomerative Clustering based on Dendrogram
from sklearn.cluster import AgglomerativeClustering

# Determine the number of clusters based on the dendrogram (e.g., cut at 4 clusters)
hierarchical = AgglomerativeClustering(n_clusters=4, affinity='euclidean', linkage='ward')
df['Cluster_Hierarchical'] = hierarchical.fit_predict(features_scaled)

# Step 4: Visualize the clusters
```

```
plt.figure(figsize=(8, 6))
sns.scatterplot(x='Sales', y='Profit', hue='Cluster_Hierarchical', data=df, palette='Set2')
plt.title("Hierarchical Clustering (k=4)")
plt.show()

# Step 5: Evaluate the clusters (similar to K-Means)
print(df.groupby('Cluster_Hierarchical').mean())
```

Explanation of Hierarchical Clustering:

1. **Linkage Matrix**: We first compute the linkage matrix using `linkage()` with the chosen distance metric (e.g., `'ward'`). The `ward` method minimizes variance within clusters.
2. **Dendrogram**: The dendrogram plot shows how data points are merged into clusters. The point where the vertical line cuts the dendrogram indicates the number of clusters.
3. **Agglomerative Clustering**: We apply Agglomerative Clustering, which is a bottom-up approach for hierarchical clustering, and assign cluster labels.
4. **Cluster Visualization**: Similar to K-Means, we visualize the clusters and evaluate the mean values of the features within each cluster.

CODE 2

```
#!/usr/bin/env python
# coding: utf-8
# Input from user for number of queens
N = int(input("Enter the number of queens: "))
print(f"Entered number of queens: {N}\n")
# Chessboard initialization (NxN matrix with all elements set to 0)
board = [[0] * N for _ in range(N)]
# Function to check if a position (i, j) is under attack by any other queen
def is_attack(i, j):
# Check if there is a queen in the same row or column
for k in range(N):
if board[i][k] == 1 or board[k][j] == 1:
return True
# Check diagonals
for k in range(N):
for l in range(N):
if (k + l == i + j) or (k - l == i - j): # Checking if in diagonal
if board[k][l] == 1:
return True
return False
# Recursive function to solve the N-Queens problem
def N_queen(n):
```

```
# If n is 0, all queens are placed, return True (solution found)
if n == 0:
return True
# Try placing a queen in every position on the board
for i in range(N):
for j in range(N):
# Check if we can place a queen here
if not is_attack(i, j) and board[i][j] != 1:
board[i][j] = 1 # Place the queen
# Recursively try to place the remaining queens
if N_queen(n - 1):
return True # If a valid arrangement is found, return True
# If placing the queen here does not lead to a solution, backtrack
board[i][j] = 0
return False
# Solve the N-Queens problem
if N_queen(N):
# Output the solution
print(f"Solution for {N}-Queens Problem:")
for row in board:
print(" ".join(str(x) for x in row))
else:
print(f"No solution exists for {N}-Queens problem.")
```