Lab 1 - Unsupervised Learning - Yassine MKAOUAR

October 17, 2024

Name : Yassine MKAOUAR

E-mail : yassine.mkaouar@polytechnique.edu

1 MAP654I: Practical Introduction to Machine Learning

1.1 Practical Session 1: Unsupervised Learning

1.2 *Temperature dataset*

Importing libraries

[126]:
```
import numpy as np
import pylab as pl
import scipy as sp
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.metrics import adjusted_rand_score, rand_score
from sklearn.mixture import GaussianMixture
import seaborn as sns
from sklearn.manifold import TSNE
```

Loading data

[2]:
```
x1 = np.load("temper.npz") #load the data
```

Data exploration

[3]:
```
x1.files
```

[3]:
```
['villes', 'data', 'varname']
```

[4]:
```
# Load the data matrix, cities, and variable names from the dataset
data_matrix = x1['data']       # Contains the main data
villes = x1['villes']          # List of cities
varname = x1['varname']        # Names of the variables/columns

# Extract the temperature matrix (first 12 columns)
temperature_matrix = data_matrix[:, :12]

# Display the extracted temperature matrix
```

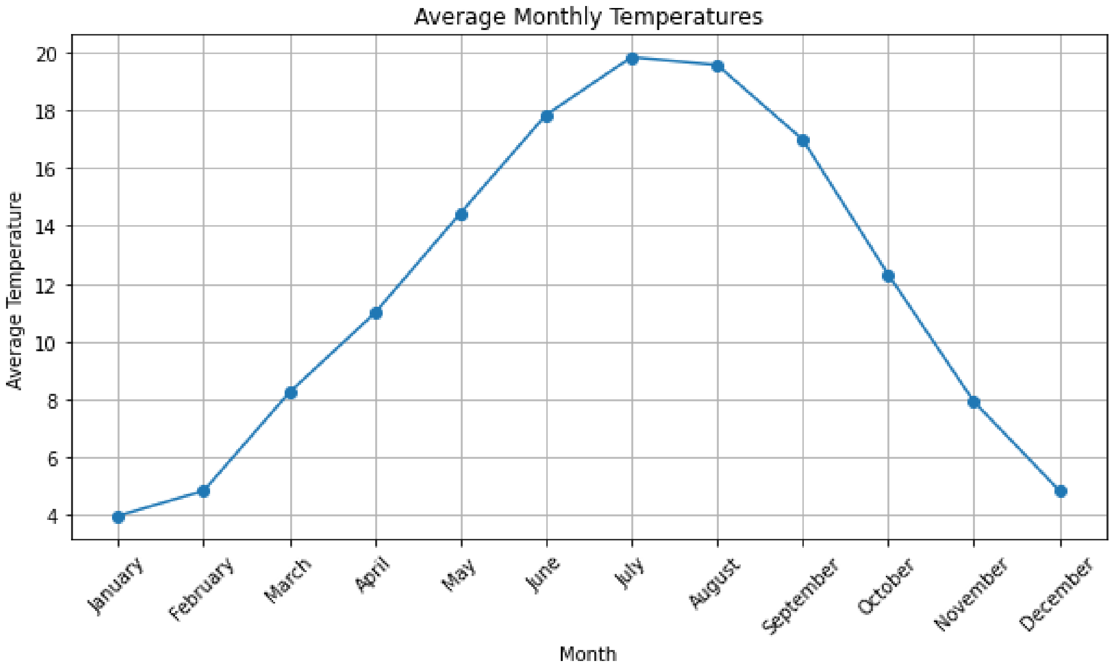
```
print("Matrice des températures (15x12) :")
print(temperature_matrix)

# Display the list of cities
print("\nListe des villes :")
print(villes)

# Display the names of the variables/columns
print("\nNoms des variables/colonnes :")
print(varname)
```

```
Matrice des températures (15x12) :
[[ 5.6  6.6 10.3 12.8 15.8 19.3 20.9 21.  18.6 13.8  9.1  6.2]
 [ 6.1  5.8  7.8  9.2 11.6 14.4 15.6 16.  14.7 12.   9.   7. ]
 [ 2.6  3.7  7.5 10.3 13.8 17.3 19.4 19.1 16.2 11.2  6.6  3.6]
 [ 1.5  3.2  7.7 10.6 14.5 17.8 20.1 19.5 16.7 11.4  6.5  2.3]
 [ 2.4  2.9  6.   8.9 12.4 15.3 17.1 17.1 14.7 10.4  6.1  3.5]
 [ 2.1  3.3  7.7 10.9 14.9 18.5 20.7 20.1 16.9 11.4  6.7  3.1]
 [ 5.5  6.6 10.  13.  16.8 20.8 23.3 22.8 19.9 15.  10.2  6.9]
 [ 5.6  6.7  9.9 12.8 16.2 20.1 22.7 22.3 19.3 14.6 10.   6.5]
 [ 5.   5.3  8.4 10.8 13.9 17.2 18.8 18.6 16.4 12.2  8.2  5.5]
 [ 7.5  8.5 10.8 13.3 16.7 20.1 22.7 22.5 20.3 16.  11.5  8.2]
 [ 3.4  4.1  7.6 10.7 14.3 17.5 19.1 18.7 16.  11.4  7.1  4.3]
 [ 4.8  5.3  7.9 10.1 13.1 16.2 17.9 17.8 15.7 11.6  7.8  5.4]
 [ 0.4  1.5  5.6  9.8 14.  17.2 19.  18.3 15.1  9.5  4.9  1.3]
 [ 4.7  5.6  9.2 11.6 14.9 18.7 20.9 20.9 18.3 13.3  8.6  5.5]
 [ 2.4  3.4  7.1  9.9 13.6 17.1 19.3 18.8 16.  11.   6.6  3.4]]

Liste des villes :
['Bordeaux' 'Brest' 'Clermont-Ferrand' 'Grenoble' 'Lille' 'Lyon'
 'Marseille' 'Montpellier' 'Nantes' 'Nice' 'Paris' 'Rennes' 'Strasbourg'
 'Toulouse' 'Vichy']

Noms des variables/colonnes :
['January' 'February' 'March' 'April' 'May' 'June' 'July' 'August'
 'September' 'October' 'November' 'December' 'Latitude' 'Longitude']
```

Monthly Average Temperature Calculation and Visualization

[5]:

```
# Calculate the averages for each month
mean_temperatures = np.mean(temperature_matrix, axis=0)

# Visualize the averages with plt.plot
plt.figure(figsize=(10, 5))
plt.plot(mean_temperatures, marker='o')
plt.title('Average Monthly Temperatures')
plt.xlabel('Month')
plt.ylabel('Average Temperature')
plt.xticks(ticks=np.arange(12), labels=x1['varname'][:12], rotation=45)
plt.grid(True)
plt.show()
```

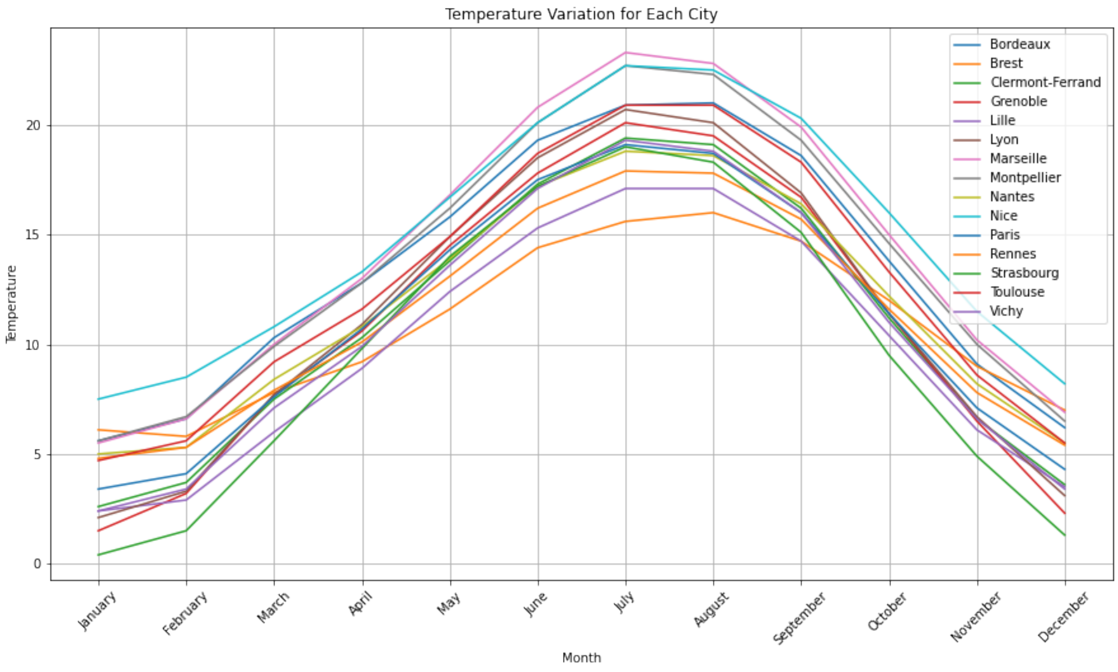


This code snippet calculates and visualizes the average monthly temperatures using data from the `temperature_matrix`. The plot shows that higher temperatures are observed in the summer months (June, July, and August) and lower temperatures in the winter months (December, January, and February).

Temperature Curves Plotting for Each City

[6]:

```
# Plot the temperature curves for each city
plt.figure(figsize=(15, 8))
for i in range(temperature_matrix.shape[0]):
    plt.plot(temperature_matrix[i], label=villes[i])

plt.title("Temperature Variation for Each City")
plt.xlabel("Month")
plt.ylabel("Temperature")
plt.xticks(ticks=np.arange(12), labels=x1['varname'][:12], rotation=45)
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```



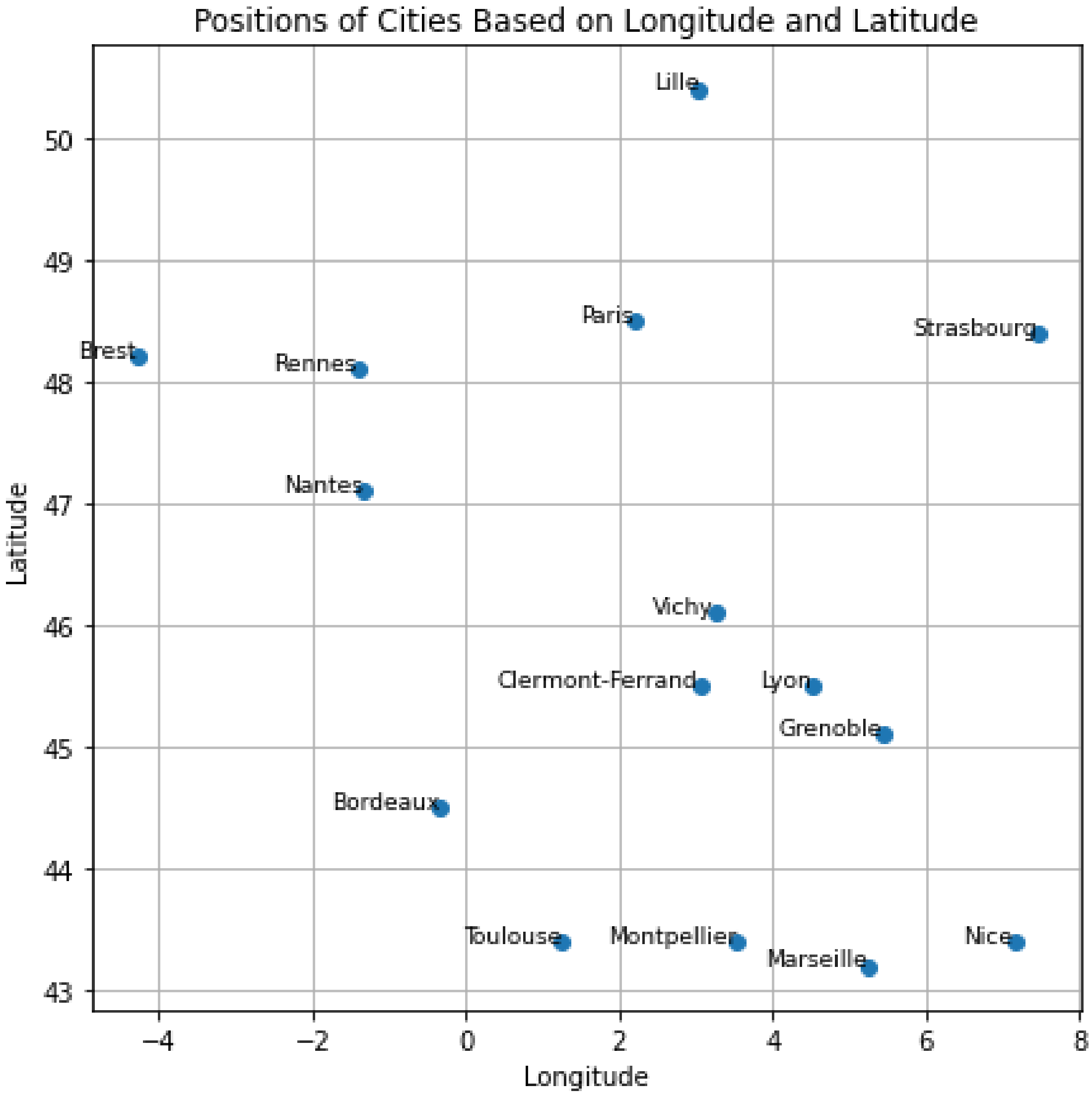
The graph displays the monthly temperature variations for different cities in France over the course of a year. Most cities show a similar trend, with temperatures increasing from January to July, peaking in the summer months, and then gradually decreasing towards December.

Overall, this graph provides a clear visual comparison of the temperature trends for different cities, highlighting both common patterns and unique characteristics of each location throughout the year.

Extracting and Plotting City Positions

[7]:

```
# Extract the latitude and longitude columns
latitude = x1['data'][:, 12]
longitude = x1['data'][:, 13]

# Extract the list of cities
cities = x1['villes']

# Plot the positions of the cities
plt.figure(figsize=(7, 7))
plt.scatter(longitude, latitude, marker='o')

# Annotate each point with the city name
for i, city in enumerate(cities):
    plt.text(longitude[i], latitude[i], city, fontsize=9, ha='right')

plt.title("Positions of Cities Based on Longitude and Latitude")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.grid(True)
plt.show()
```

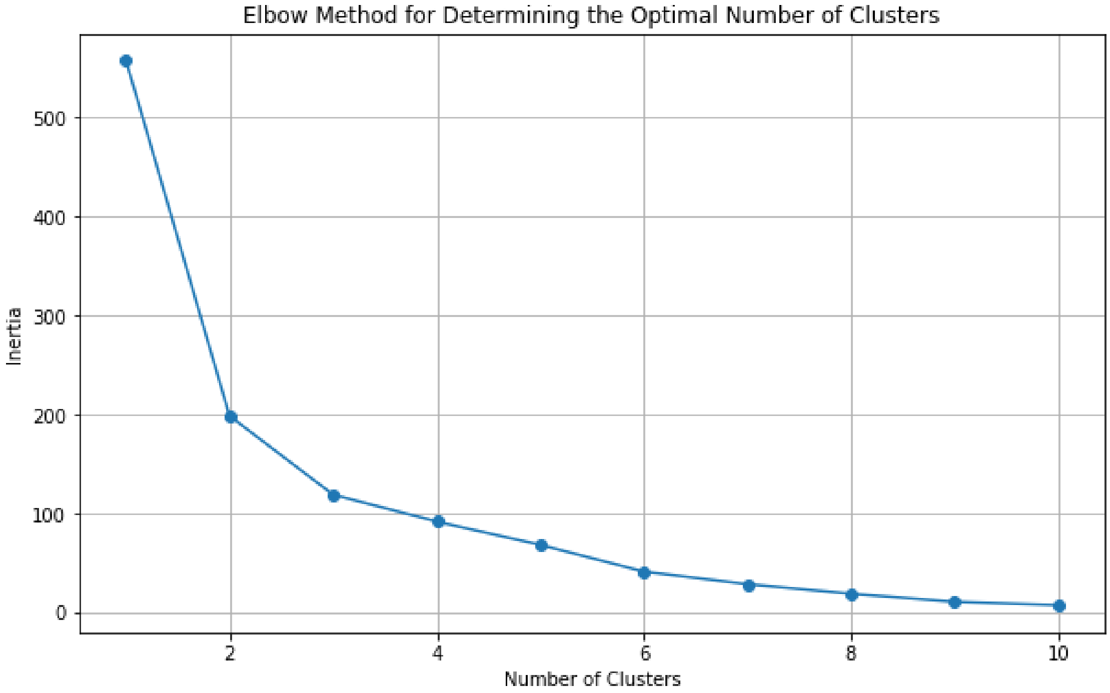


1.2.1 K-means clustering

Elbow method for determining the optimum number of clusters

[8]:
```
inertias = []
k_range = range(1, 11)

for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(temperature_matrix)
    inertias.append(kmeans.inertia_)
```

```
# Plot the inertia curve as a function of the number of clusters
plt.figure(figsize=(10, 6))
plt.plot(k_range, inertias, marker='o')
plt.title("Elbow Method for Determining the Optimal Number of Clusters")
plt.xlabel("Number of Clusters")
plt.ylabel("Inertia")
plt.grid(True)
plt.show()
```



We observe that the elbow occurs at (k = 2), where the inertia value begins to decline at a slower pace. However, at (k = 3), the inertia value continues to decrease at a notable rate. Consequently, both (k = 2) and (k = 3) can be considered for analysis and comparison of the results.

Training the K-means model

k_optimal = 2 Here, we will train the K-means model with (k = 2) and visualize the clusters. The algorithm is trained based on the temperature data for each city.

[40]:

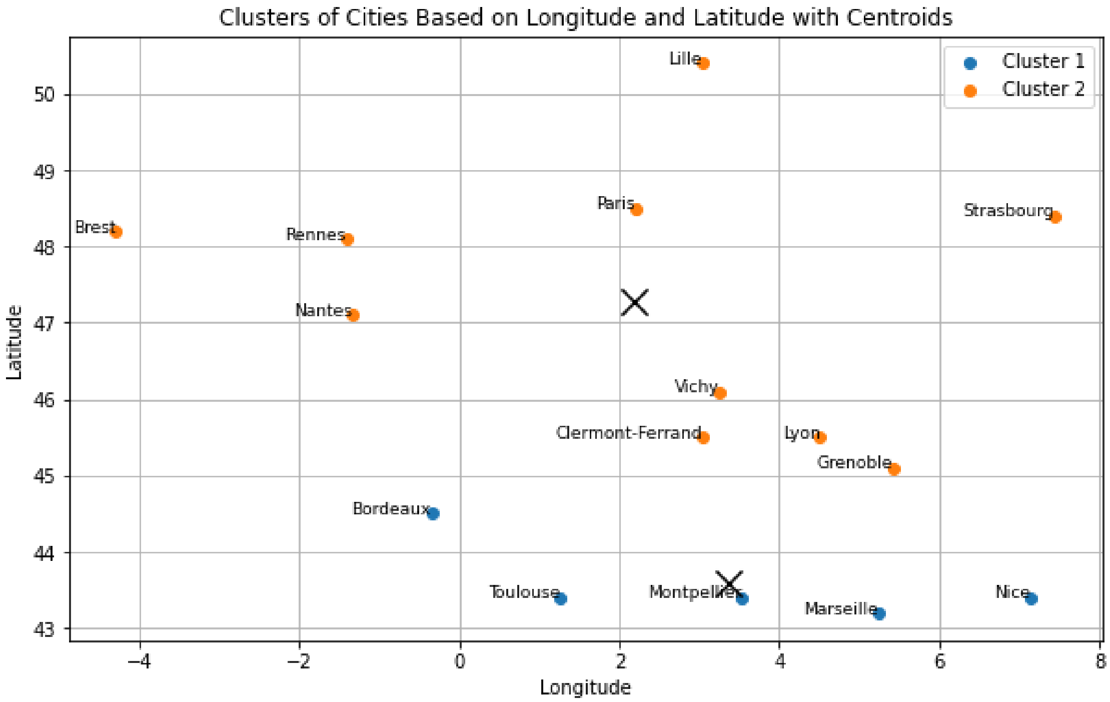
```
# Choose the number of clusters
k_optimal = 2

# Apply the KMeans algorithm
```

```python
kmeans = KMeans(n_clusters=k_optimal, random_state=42) # random_state=42 for reproducibility
clusters = kmeans.fit_predict(temperature_matrix)
```

Cluster display by longitude and latitude

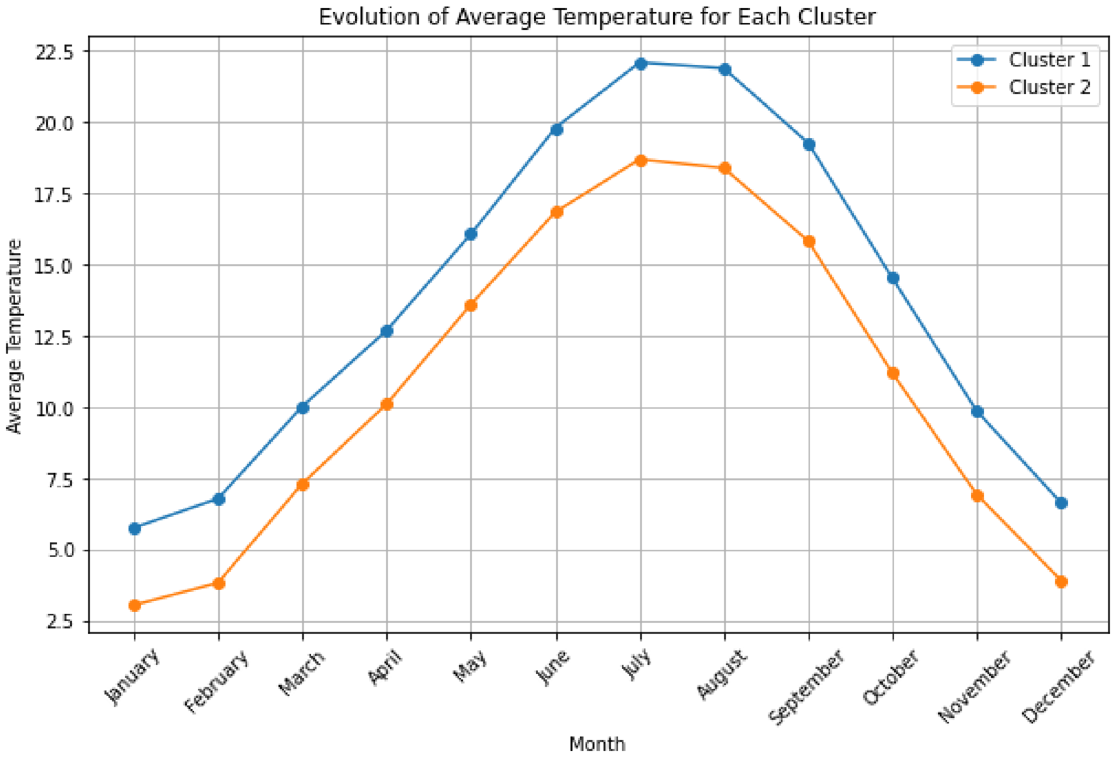
[41]:

```python
# Calculate the centroids using the average latitudes and longitudes of the clusters
centroids = np.zeros((k_optimal, 2))
for cluster in range(k_optimal):
    centroids[cluster, 0] = np.mean(longitude[clusters == cluster])
    centroids[cluster, 1] = np.mean(latitude[clusters == cluster])

# Visualize the clusters based on longitude and latitude
plt.figure(figsize=(10, 6))
for cluster in range(k_optimal):
    plt.scatter(longitude[clusters == cluster], latitude[clusters == cluster], label=f'Cluster {cluster + 1}', cmap='tab10', vmax=9)
    plt.scatter(centroids[cluster, 0], centroids[cluster, 1], marker='x', s=200, c='black')  # Display the centroids

# Annotate each point with the city name
for i, city in enumerate(cities):
    plt.text(longitude[i], latitude[i], city, fontsize=9, ha='right')

plt.title("Clusters of Cities Based on Longitude and Latitude with Centroids")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.legend()
plt.grid(True)
plt.show()
```



Display the average temperature curve for each cluster

[43]:
```
# Calculate the average temperature for each cluster
average_temperatures_per_cluster = np.zeros((k_optimal, temperature_matrix.
 ↪shape[1]))

for cluster in range(k_optimal):
    average_temperatures_per_cluster[cluster] = np.
 ↪mean(temperature_matrix[clusters == cluster], axis=0)

# Plot the average temperature evolution for each cluster
plt.figure(figsize=(10, 6))
for cluster in range(k_optimal):
    plt.plot(average_temperatures_per_cluster[cluster], marker='o',␣
 ↪label=f'Cluster {cluster + 1}')

plt.title("Evolution of Average Temperature for Each Cluster")
plt.xlabel("Month")
plt.ylabel("Average Temperature")
plt.xticks(ticks=np.arange(12), labels=varname[:12], rotation=45)
plt.legend()
plt.grid(True)
plt.show()
```



We can observe that the two clusters exhibit distinct temperature patterns throughout the year. Cluster 1 represents cities with higher average temperatures, while Cluster 2 includes cities with lower average temperatures. The temperature curves for each cluster provide a clear visual representation of the differences in temperature trends between the two groups of cities.

k_optimal = 3 Here, we will train the K-means model with (k = 3) and visualize the clusters. The algorithm is trained based on the temperature data for each city.

[44]:

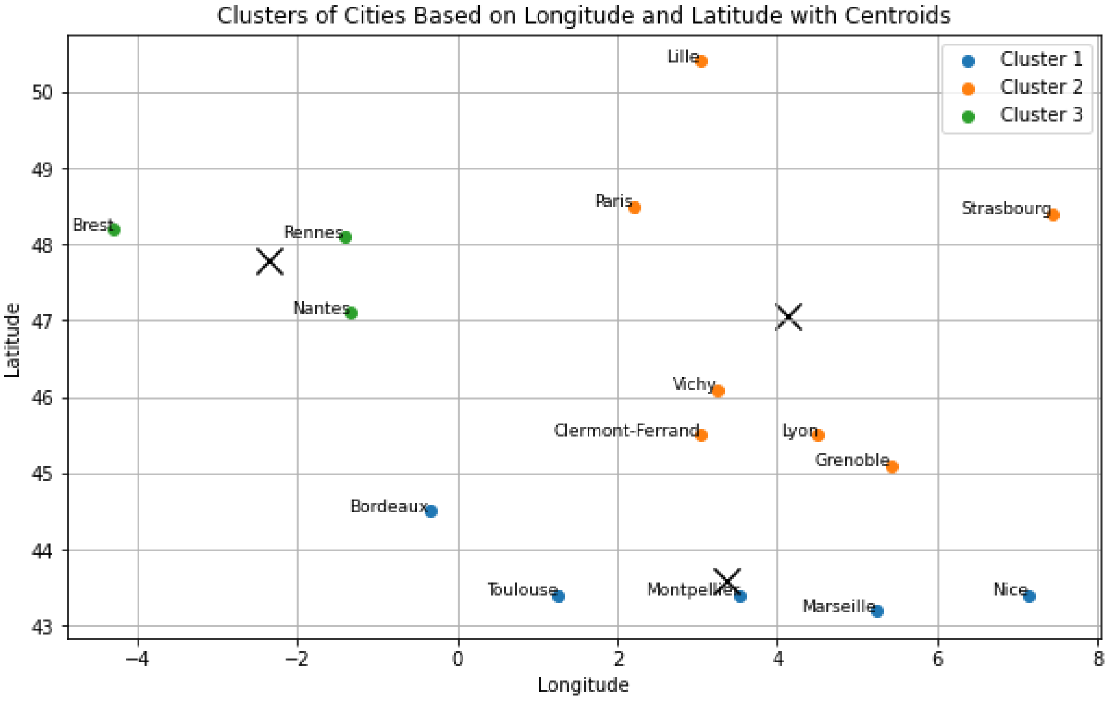
```
# Choix du nombre de clusters
k_optimal = 3

# Appliquer l'algorithme KMeans
kmeans = KMeans(n_clusters=k_optimal, random_state=42)
clusters = kmeans.fit_predict(temperature_matrix)
```

Cluster display by longitude and latitude

[45]:

```
# Calculate the centroids using the average latitudes and longitudes of the␣
 ↪clusters
centroids = np.zeros((k_optimal, 2))
for cluster in range(k_optimal):
    centroids[cluster, 0] = np.mean(longitude[clusters == cluster])
    centroids[cluster, 1] = np.mean(latitude[clusters == cluster])
```

```
# Visualize the clusters based on longitude and latitude
plt.figure(figsize=(10, 6))
for cluster in range(k_optimal):
    plt.scatter(longitude[clusters == cluster], latitude[clusters == cluster],␣
  ↪label=f'Cluster {cluster + 1}', cmap='tab10', vmax=9)
    plt.scatter(centroids[cluster, 0], centroids[cluster, 1], marker='x',␣
  ↪s=200, c='black')  # Display the centroids

# Annotate each point with the city name
for i, city in enumerate(cities):
    plt.text(longitude[i], latitude[i], city, fontsize=9, ha='right')

plt.title("Clusters of Cities Based on Longitude and Latitude with Centroids")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.legend()
plt.grid(True)
plt.show()
```

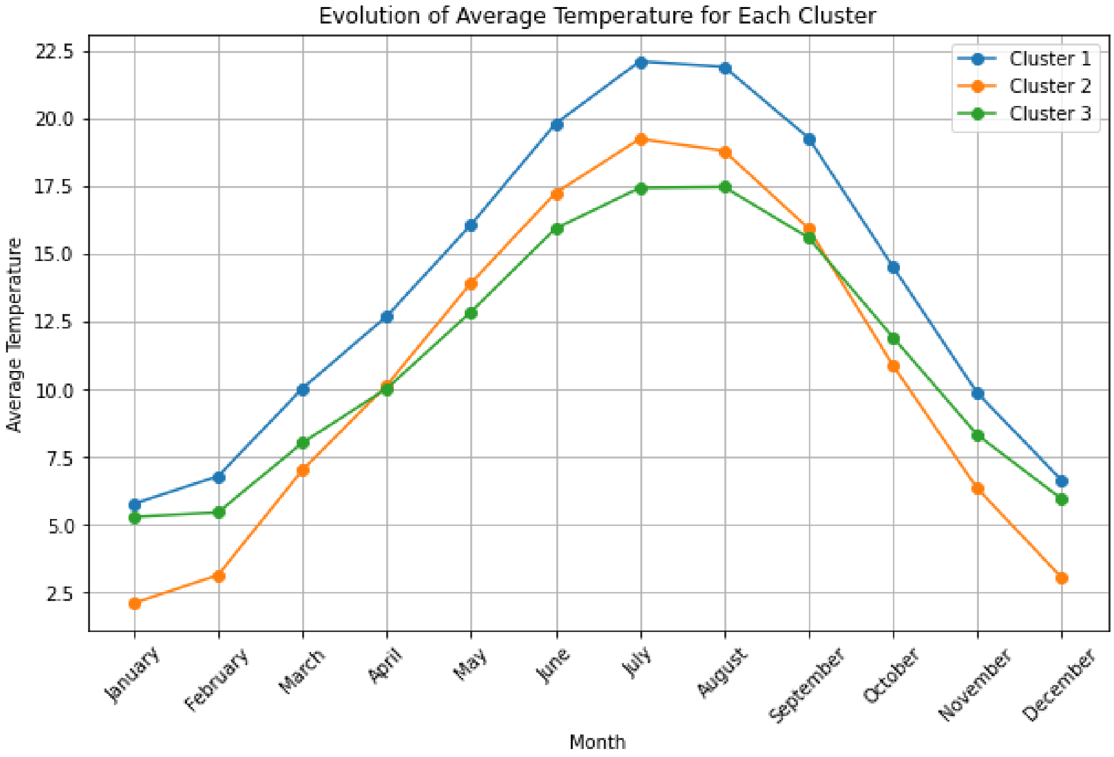


While training the K-means on the temperature data for each city, we observe that the samples are clustered by their geographical positions. So, the clustering recover a geographical similarity between the cities because each region has a specific temperature range and climate patterns are largely influenced by geographic location. For example, cities within the same region or with similar geographical characteristics (such as proximity to the coast, altitude, or latitude) tend to experience comparable temperature variations throughout the year.

Display the average temperature curve for each cluster

[14]:

```
# Calculate the average temperature for each cluster
average_temperatures_per_cluster = np.zeros((k_optimal, temperature_matrix.
 ↪shape[1]))

for cluster in range(k_optimal):
    average_temperatures_per_cluster[cluster] = np.
 ↪mean(temperature_matrix[clusters == cluster], axis=0)

# Plot the average temperature evolution for each cluster
plt.figure(figsize=(10, 6))
for cluster in range(k_optimal):
    plt.plot(average_temperatures_per_cluster[cluster], marker='o',␣
 ↪label=f'Cluster {cluster + 1}')

plt.title("Evolution of Average Temperature for Each Cluster")
plt.xlabel("Month")
plt.ylabel("Average Temperature")
plt.xticks(ticks=np.arange(12), labels=varname[:12], rotation=45)
plt.legend()
plt.grid(True)
plt.show()
```



Cluster 1 stands out as having the highest temperatures consistently throughout the year. When comparing **Cluster 2** and **Cluster 3**, distinct differences in temperature patterns emerge. During the summer, **Cluster 2** experiences higher temperatures than **Cluster 3**, with a more pronounced peak, indicating that cities in Cluster 2 generally have warmer summers. However, in the winter, **Cluster 2** has colder temperatures, with the curve dipping below that of **Cluster 3**, suggesting colder winters. This implies that cities in **Cluster 2** experience more extreme seasonal variations, with hot summers and cold winters. In contrast, **Cluster 3** maintains a more moderate temperature profile, with cooler summers and milder winters, showing less seasonal fluctuation overall.

1.2.2 Density Estimation

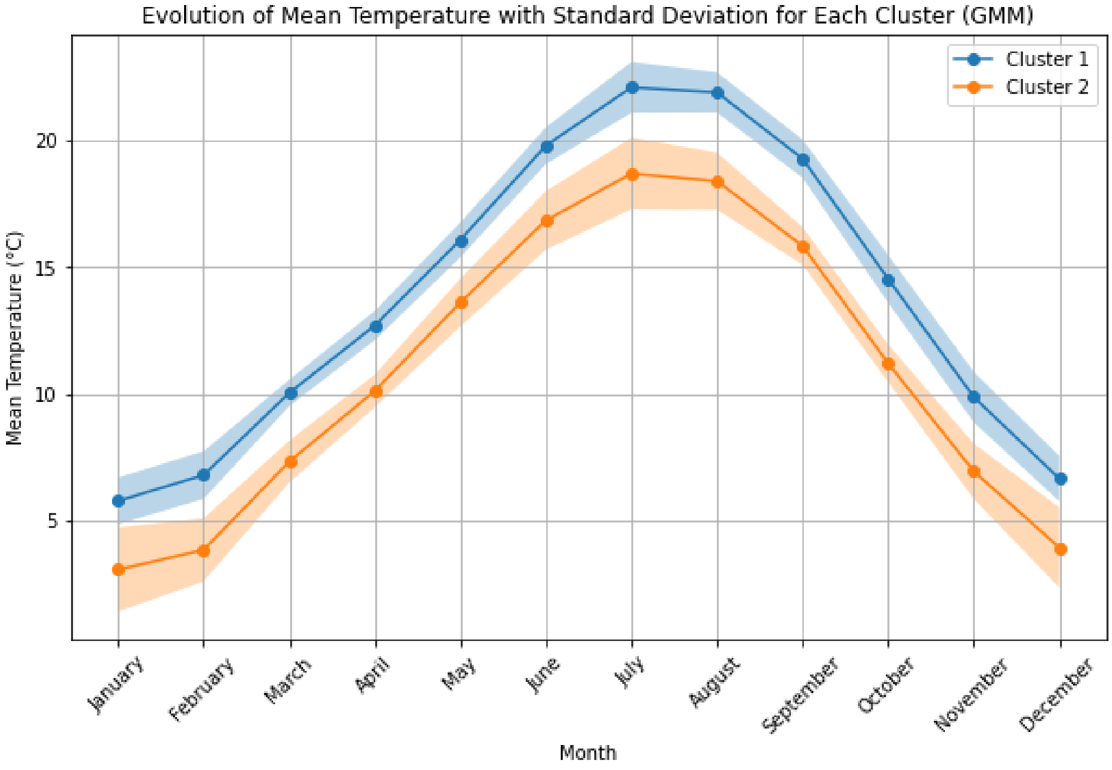
K = 2

[46]:
```
# Set up and fit the GMM
gmm = GaussianMixture(n_components=2, covariance_type='diag', random_state=42)
gmm.fit(temperature_matrix)

# Predict the labels (clusters) for each city
labels = gmm.predict(temperature_matrix)

# Get the GMM parameters
weights = gmm.weights_
means = gmm.means_
covariances = gmm.covariances_
std_devs = np.sqrt(covariances)

# Output the results
print("Villes:", villes)
print("Cluster labels:", labels)
print("Weights:", weights)
print("Means:", means)
print("Covariances:", covariances)
```

```
Villes: ['Bordeaux' 'Brest' 'Clermont-Ferrand' 'Grenoble' 'Lille' 'Lyon'
 'Marseille' 'Montpellier' 'Nantes' 'Nice' 'Paris' 'Rennes' 'Strasbourg'
 'Toulouse' 'Vichy']
Cluster labels: [0 1 1 1 1 1 0 0 1 0 1 1 1 0 1]
Weights: [0.33333329 0.66666671]
Means: [[ 5.78000015  6.80000017 10.04000012 12.70000016 16.08000017 19.80000016
  22.10000017 21.90000014 19.28000014 14.54000018  9.88000018  6.66000016]
 [ 3.07000012  3.85000012  7.33000013 10.1200001  13.61000009 16.85000013
  18.70000016 18.40000018 15.84000017 11.21000015  6.95000012  3.94000011]]
Covariances: [[0.85360096 0.88400092 0.27440094 0.33600088 0.47760087 0.5280009
  1.00800094 0.62800095 0.56960094 0.88640091 0.99760091 0.80240092]
 [2.76610099 1.54450111 0.6881012  0.41560113 0.89690105 1.33850115
 1.9880012  1.27000135 0.55240139 0.54890127 1.19450111 2.502401  ]]
```

[47]:

```python
# Plot the evolution of the mean temperature with standard deviation
plt.figure(figsize=(10, 6))

# For each cluster, plot the mean and fill the standard deviation band
for i in range(gmm.n_components):
    plt.plot(means[i], label=f'Cluster {i + 1}', marker='o')

    # Fill the area between mean ± std_dev
    plt.fill_between(np.arange(12),
                     means[i] - std_devs[i],
                     means[i] + std_devs[i],
                     alpha=0.3)

plt.title("Evolution of Mean Temperature with Standard Deviation for Each␣
 ↪Cluster (GMM)")
plt.xlabel("Month")
plt.ylabel("Mean Temperature (°C)")
plt.xticks(ticks=np.arange(12), labels=varname[:12], rotation=45)
plt.legend()
plt.grid(True)
plt.show()
```



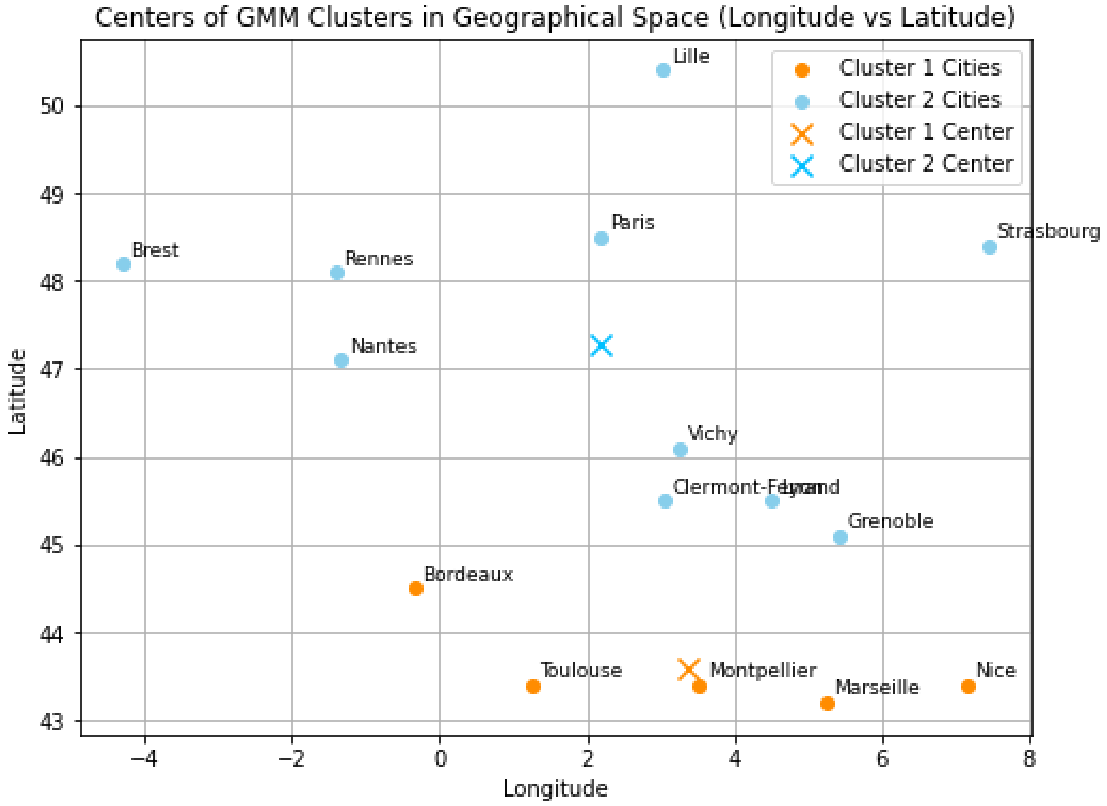
The plot shows the **mean temperature** evolution over the months for two distinct clusters identified by the Gaussian Mixture Model (GMM). Each cluster has associated error bands representing the **standard deviation** (the shaded regions around each line), indicating the variability in temperatures within that cluster.

1. **Cluster 1 (Blue)**:
   - This cluster represents cities that experience **higher mean temperatures** throughout the year.
   - The peak temperature is observed around the summer months (June to August), showing a noticeable increase in temperature during this period.
   - The error bands (standard deviation) indicate that there is some variability in temperatures within this cluster, but the variability is relatively moderate.
2. **Cluster 2 (Orange)**:
   - This cluster represents cities that experience **lower mean temperatures** throughout the year.
   - Similar to Cluster 1, the peak is also observed in the summer months, but the mean temperatures are consistently lower than those of Cluster 1.
   - The error bands in this cluster suggest slightly less variability compared to Cluster 1, but it still shows some spread in temperatures.

The shaded regions for both clusters show that while there are variations in temperature, each cluster maintains a consistent separation between the warmer (Cluster 1) and cooler (Cluster 2) cities across the year.

[51]:

```python
# Calculate the geographical centers (latitude and longitude) for each cluster
cluster_0_center = [latitude[labels == 0].mean(), longitude[labels == 0].mean()]
cluster_1_center = [latitude[labels == 1].mean(), longitude[labels == 1].mean()]

# Plot the geographical position of the cities with the cluster centers
plt.figure(figsize=(8, 6))

# Scatter plot for cities in Cluster 0 (e.g., red)
plt.scatter(longitude[labels == 0], latitude[labels == 0], color='darkorange', marker='o', label='Cluster 1 Cities', cmap='tab10', vmax=9)

# Scatter plot for cities in Cluster 1 (e.g., blue)
plt.scatter(longitude[labels == 1], latitude[labels == 1], color='skyblue', marker='o', label='Cluster 2 Cities', cmap='tab10', vmax=9)

# Plot the cluster centers
plt.scatter(cluster_0_center[1], cluster_0_center[0], color='darkorange', label='Cluster 1 Center', marker='x', s=100)
plt.scatter(cluster_1_center[1], cluster_1_center[0], color='deepskyblue', label='Cluster 2 Center', marker='x', s=100)

# Annotate the cities with their names
for i, ville in enumerate(villes):
    plt.text(longitude[i] + 0.1, latitude[i] + 0.1, ville, fontsize=9)

# Add plot titles and labels
plt.title('Centers of GMM Clusters in Geographical Space (Longitude vs␣
 ↪Latitude)')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.legend()
plt.grid(True)
plt.show()
```



The centers of the Gaussian distributions in the mixture model represent the mean temperatures for each cluster. This analysis helps to identify distinct temperature patterns and trends for different groups of cities based on their geographical locations and climate characteristics. We notice that for K = 2, the red cluster is the one with the highest temperatures represting the southern cities of France and the blue cluster is the one with the lowest temperatures representing the northern and eastern cities of France.

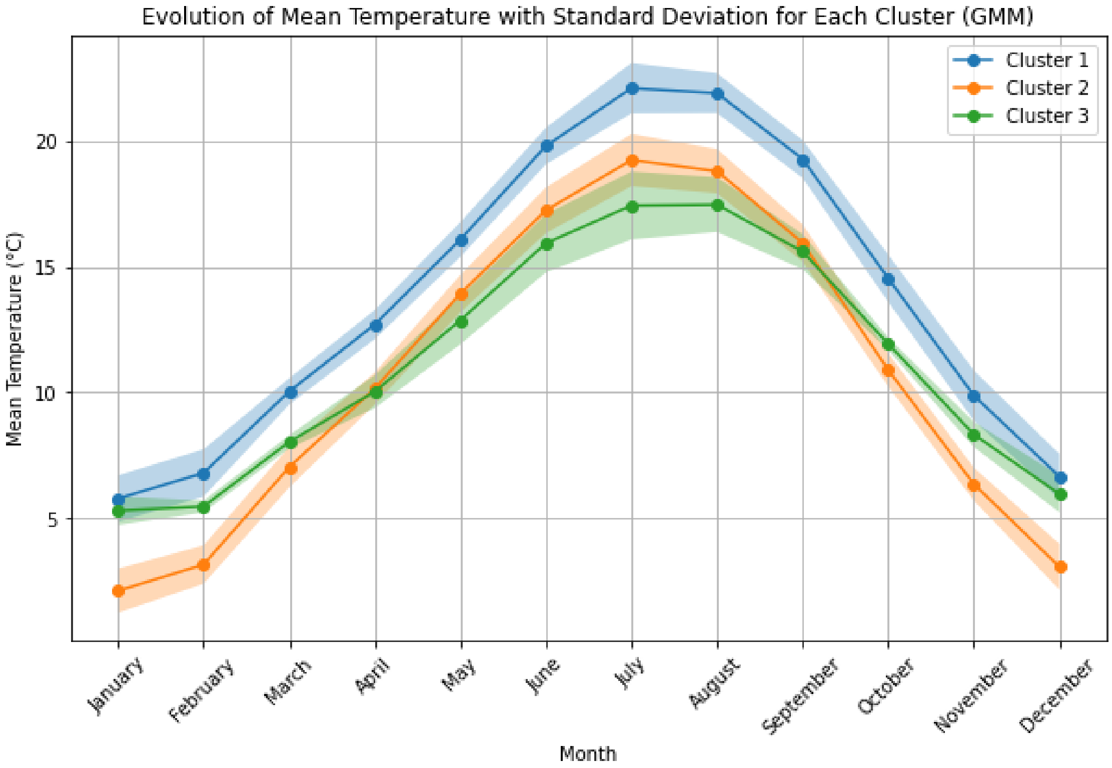
K = 3

[54]:
```
# Set up and fit the GMM
gmm = GaussianMixture(n_components=3, covariance_type='diag', random_state=42)
```

```
gmm.fit(temperature_matrix)

# Predict the labels (clusters) for each city
labels = gmm.predict(temperature_matrix)

# Get the GMM parameters
weights = gmm.weights_
means = gmm.means_
covariances = gmm.covariances_
std_devs = np.sqrt(covariances)

# Output the results
print("Villes:", villes)
print("Cluster labels:", labels)
print("Weights:", weights)
print("Means:", means)
print("Covariances:", covariances)
```

```
Villes: ['Bordeaux' 'Brest' 'Clermont-Ferrand' 'Grenoble' 'Lille' 'Lyon'
 'Marseille' 'Montpellier' 'Nantes' 'Nice' 'Paris' 'Rennes' 'Strasbourg'
 'Toulouse' 'Vichy']
Cluster labels: [0 2 1 1 1 1 0 0 2 0 1 2 1 0 1]
Weights: [0.33333333 0.46666667 0.2       ]
Means: [[ 5.78        6.8        10.04       12.7        16.08       19.8
   22.1        21.9        19.28       14.54        9.88        6.66      ]
 [ 2.11428573  3.15714287  7.02857143 10.15714286 13.92857143 17.24285714
   19.24285714 18.8        15.94285714 10.90000001  6.35714287  3.07142858]
 [ 5.3         5.46666667  8.03333333 10.03333333 12.86666666 15.93333332
   17.43333332 17.46666666 15.59999999 11.93333333  8.33333334  5.96666667]]
Covariances: [[0.853601   0.884001   0.274401   0.336001   0.477601   0.528001
  1.008001   0.628001   0.569601   0.886401   0.997601   0.802401  ]
 [0.76693981 0.58245    0.65061325 0.40530712 0.55347039 0.82244998
  1.07959284 0.77428671 0.54530712 0.43714386 0.42816428 0.83061327]
 [0.32666767 0.05555656 0.06888989 0.42888989 0.90888989 1.34222323
  1.81555656 1.18222323 0.48666767 0.06222322 0.24888989 0.53555656]]
```

[55]:
```
# Plot the evolution of the mean temperature with standard deviation
plt.figure(figsize=(10, 6))

# For each cluster, plot the mean and fill the standard deviation band
for i in range(gmm.n_components):
    plt.plot(means[i], label=f'Cluster {i + 1}', marker='o')

    # Fill the area between mean ± std_dev
    plt.fill_between(np.arange(12),
                     means[i] - std_devs[i],
                     means[i] + std_devs[i],
                    alpha=0.3)

plt.title("Evolution of Mean Temperature with Standard Deviation for Each␣
  ↪Cluster (GMM)")
plt.xlabel("Month")
plt.ylabel("Mean Temperature (°C)")
plt.xticks(ticks=np.arange(12), labels=varname[:12], rotation=45)
plt.legend()
plt.grid(True)
plt.show()
```

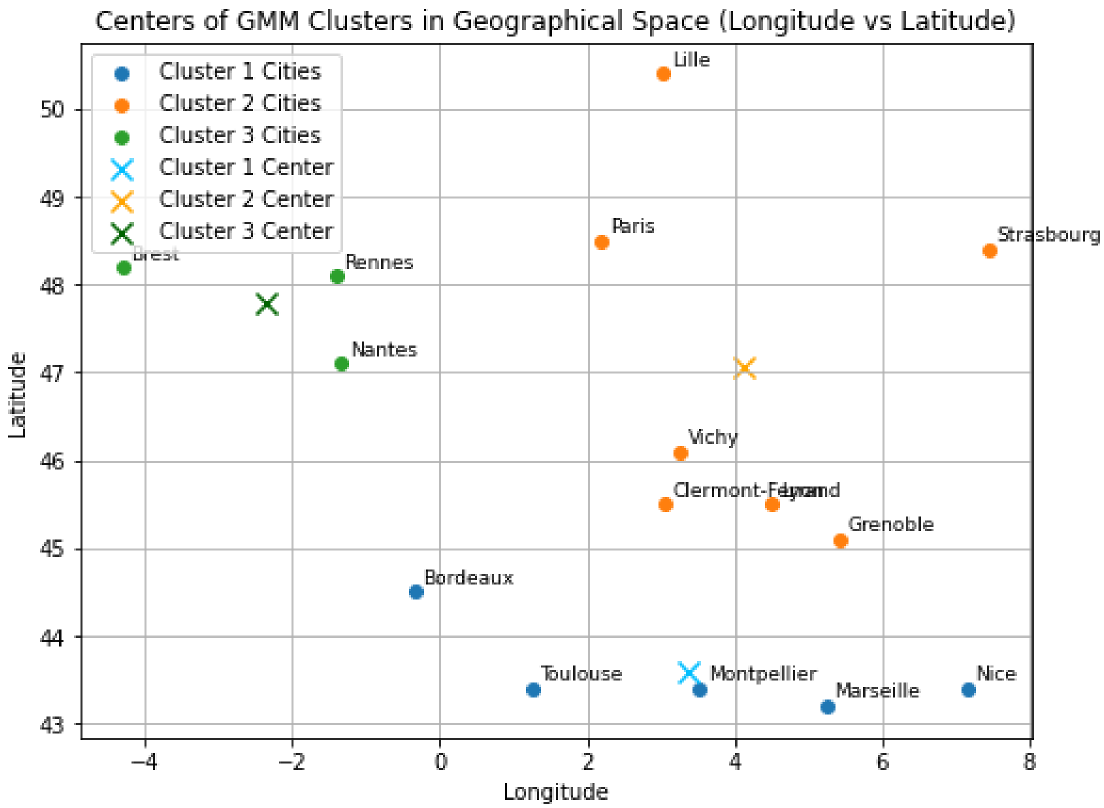


- **Cluster 1 (High Temperature)**: This cluster represents areas where temperatures are significantly higher, particularly in summer. The data shows a consistently high average temperature, with a relatively low standard deviation, indicating a certain stability of temperatures in this region.
- **Cluster 2**: In summer, the temperatures are higher than those of Cluster 3, while in winter, the temperatures are lower than those of Cluster 3.
- **Cluster 3**: In summer, this cluster exhibits lower temperatures compared to Cluster 2, indicating a cooler climate. In winter, the trend reverses, with higher temperatures compared to Cluster 2.

Variability and Standard Deviation: - Winter: During winter, the two clusters (Cluster 2 and Cluster 3) show a clear separation in terms of temperature, indicating that the areas corresponding to each cluster have distinct winter conditions. This suggests that climatic factors (such as altitude or proximity to the ocean) strongly influence these clusters, making their distinction more pronounced. - **Summer**: In summer, the variability of temperatures in the two clusters (Cluster 2 and Cluster 3) shows some overlap, with higher standard deviations. This may indicate greater climatic variability or local influences affecting the temperature, making the clusters less distinct compared to winter.

[61]:

```
# Calculate the geographical centers (latitude and longitude) for each cluster
cluster_0_center = [latitude[labels == 0].mean(), longitude[labels == 0].mean()]
cluster_1_center = [latitude[labels == 1].mean(), longitude[labels == 1].mean()]
cluster_2_center = [latitude[labels == 2].mean(), longitude[labels == 2].mean()]

# Plot the geographical position of the cities with the cluster centers
plt.figure(figsize=(8, 6))

# Scatter plot for cities in Cluster 0 (e.g., red)
plt.scatter(longitude[labels == 0], latitude[labels == 0],  marker='o',␣
 ↪label='Cluster 1 Cities', cmap='tab10', vmax=9)

# Scatter plot for cities in Cluster 1 (e.g., blue)
plt.scatter(longitude[labels == 1], latitude[labels == 1],  marker='o',␣
 ↪label='Cluster 2 Cities', cmap='tab10', vmax=9)

# Scatter plot for cities in Cluster 2 (e.g., green)
plt.scatter(longitude[labels == 2], latitude[labels == 2],  marker='o',␣
 ↪label='Cluster 3 Cities', cmap='tab10', vmax=9)

# Plot the cluster centers
plt.scatter(cluster_0_center[1], cluster_0_center[0], color='deepskyblue',␣
 ↪label='Cluster 1 Center', marker='x', s=100, cmap='tab10', vmax=9)
plt.scatter(cluster_1_center[1], cluster_1_center[0], color='orange',␣
 ↪label='Cluster 2 Center', marker='x', s=100, cmap='tab10', vmax=9)
plt.scatter(cluster_2_center[1], cluster_2_center[0], color='darkgreen',␣
 ↪label='Cluster 3 Center', marker='x', s=100, cmap='tab10', vmax=9)

# Annotate the cities with their names
for i, ville in enumerate(villes):
    plt.text(longitude[i] + 0.1, latitude[i] + 0.1, ville, fontsize=9)

# Add plot titles and labels
plt.title('Centers of GMM Clusters in Geographical Space (Longitude vs␣
 ↪Latitude)')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.legend()
plt.grid(True)
plt.show()
```



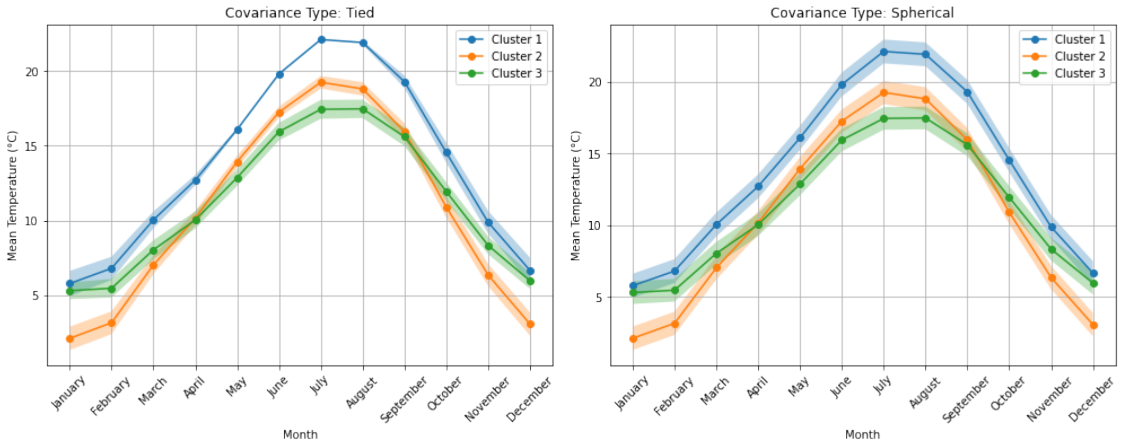
For k=3, The clusters are more clearly separated compared to k=2, where each cluster represents a distinct temperature profile based on the geographical characteristics of the cities. For example, Cluster 1 represents cities located in the northern and eastern regions of France, which experience lower temperatures throughout the year. Cluster 2 includes cities with moderate temperatures, while Cluster 3 consists of cities in the southern regions with higher temperatures. The distinct temperature patterns observed in each cluster highlight the influence of geographical location on climate and temperature variations.

[76]:

```
# Covariance types to test
covariance_types = ['tied','spherical']

# Initialize the figure
plt.figure(figsize=(14, 10))

# Loop over the different covariance types and fit GMM for each
for idx, cov_type in enumerate(covariance_types):
    # Create and fit the GMM model
    gmm = GaussianMixture(n_components=3, covariance_type=cov_type,␣
 ↪random_state=42)
    gmm.fit(temperature_matrix)
```

```python
    # Predict the labels (clusters) for each city
    labels = gmm.predict(temperature_matrix)

    # Get the GMM parameters
    means = gmm.means_

    covariances = gmm.covariances_

    std_devs = np.sqrt(covariances)

    # Create a subplot for each covariance type
    plt.subplot(2, 2, idx + 1)

    # Plot the evolution of the mean temperature with standard deviation
    for i in range(gmm.n_components):
        plt.plot(means[i], label=f'Cluster {i + 1}', marker='o')

        # Fill the area between mean ± std_dev
        plt.fill_between(np.arange(12),
                         means[i] - std_devs[i],
                         means[i] + std_devs[i],
                         alpha=0.3)

    plt.title(f"Covariance Type: {cov_type.capitalize()}")
    plt.xlabel("Month")
    plt.ylabel("Mean Temperature (°C)")
    plt.xticks(ticks=np.arange(12), labels=varname[:12], rotation=45)
    plt.legend()
    plt.grid(True)

# Adjust layout and show the figure
plt.tight_layout()
plt.show()
```

```
C:\Users\moham\AppData\Local\Temp\ipykernel_7620\1124486566.py:21:
RuntimeWarning: invalid value encountered in sqrt
  std_devs = np.sqrt(covariances)
```



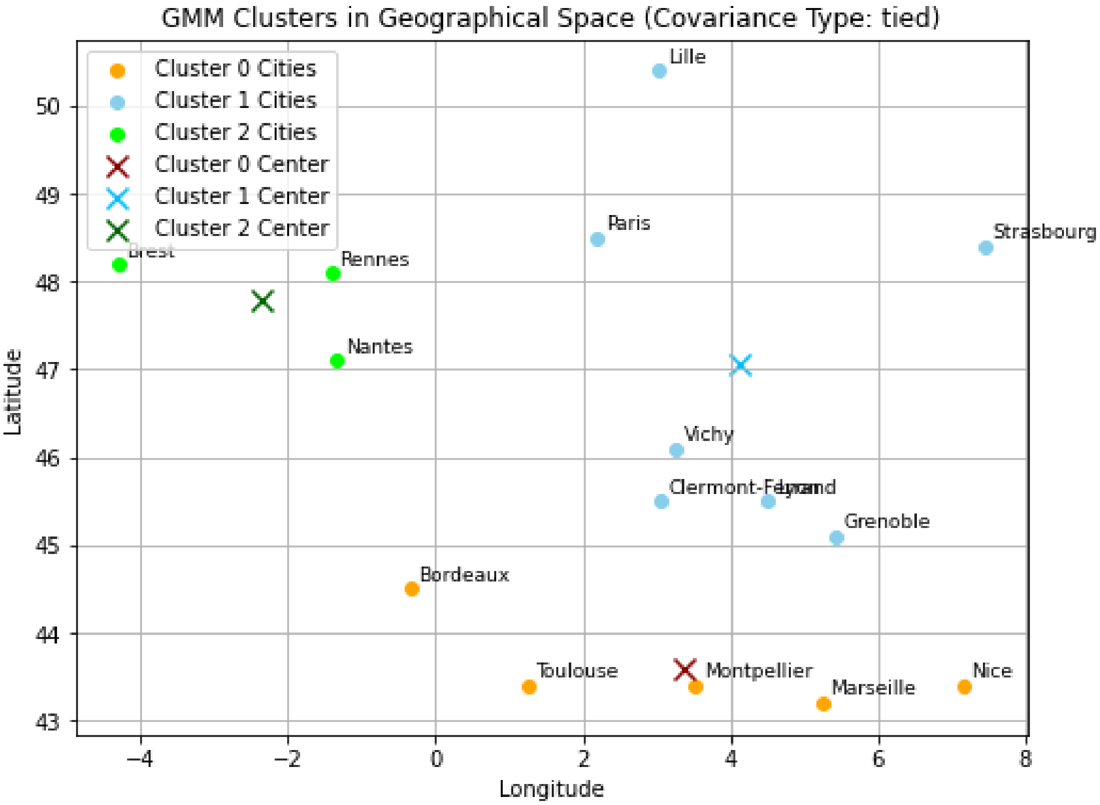
In Gaussian Mixture Models (GMM), four types of covariance are available:

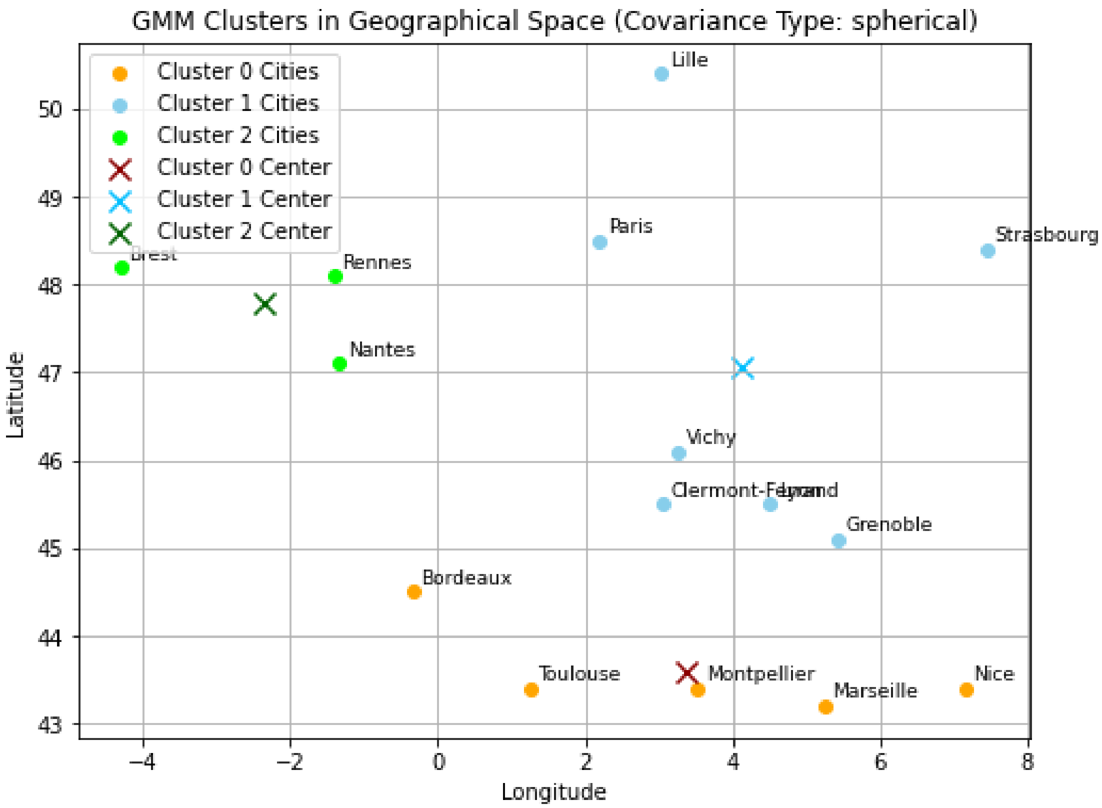
1. **Full**: Each component has its own full covariance matrix, allowing each component to have a unique shape, orientation, and size across all dimensions.
2. **Tied**: All components share the same general covariance matrix, which forces all components to have the same shape and orientation, encouraging a more spherical distribution.
3. **Diag**: Each component has its own diagonal covariance matrix, allowing components to have different variances along each dimension, but assuming no correlation between dimensions.
4. **Spherical**: Each component has its own unique variance, assuming that the shape of each component is spherical, with a single variance across all dimensions.

The two plots above represent the evolution of mean temperatures with 2 covariance types ('tied' and 'spherical') for three clusters over 12 months. The differences could be attributed to the different covariance types used in the Gaussian Mixture Model (GMM) for each plot, affecting how clusters are shaped and how much overlap there is between them but in general, the results are similar.

[77]:
```
for covariance_type in covariance_types:
    # Initialize the GMM model for each covariance type
    gmm = GaussianMixture(n_components=3, covariance_type=covariance_type,␣
 ↪random_state=42)

    # Fit the GMM model to the temperature data
    gmm.fit(temperature_matrix)

    # Predict the cluster labels for each city
    labels = gmm.predict(temperature_matrix)

    # Calculate the geographical centers (latitude and longitude) for each␣
 ↪cluster
    cluster_0_center = [latitude[labels == 0].mean(), longitude[labels == 0].
 ↪mean()]
```

```python
    cluster_1_center = [latitude[labels == 1].mean(), longitude[labels == 1].mean()]
    cluster_2_center = [latitude[labels == 2].mean(), longitude[labels == 2].mean()]

    # Create a new figure for each covariance type
    plt.figure(figsize=(8, 6))

    # Scatter plot for cities in Cluster 0 (orange)
    plt.scatter(longitude[labels == 0], latitude[labels == 0], color='orange', marker='o', label='Cluster 0 Cities')

    # Scatter plot for cities in Cluster 1 (blue)
    plt.scatter(longitude[labels == 1], latitude[labels == 1], color='skyblue', marker='o', label='Cluster 1 Cities')

    # Scatter plot for cities in Cluster 2 (green)
    plt.scatter(longitude[labels == 2], latitude[labels == 2], color='lime', marker='o', label='Cluster 2 Cities')

    # Plot the cluster centers
    plt.scatter(cluster_0_center[1], cluster_0_center[0], color='darkred', label='Cluster 0 Center', marker='x', s=100)
    plt.scatter(cluster_1_center[1], cluster_1_center[0], color='deepskyblue', label='Cluster 1 Center', marker='x', s=100)
    plt.scatter(cluster_2_center[1], cluster_2_center[0], color='darkgreen', label='Cluster 2 Center', marker='x', s=100)

    # Annotate the cities with their names
    for i, ville in enumerate(villes):
        plt.text(longitude[i] + 0.1, latitude[i] + 0.1, ville, fontsize=9)

    # Add plot titles and labels
    plt.title(f'GMM Clusters in Geographical Space (Covariance Type: {covariance_type})')
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.legend()
    plt.grid(True)

    # Show the plot
    plt.show()
```


GMM Clusters in Geographical Space (Covariance Type: tied)
Cluster 0 Cities
Cluster 1 Cities
Cluster 2 Cities
Cluster 0 Center
Cluster 1 Center
Cluster 2 Center
Lille
Paris
Strasbourg
Brest
Rennes
Nantes
Vichy
Clermont-Ferrand
Lyon
Grenoble
Bordeaux
Toulouse
Montpellier
Marseille
Nice
Latitude
50
49
48
47
46
45
44
43
−4
−2
0
2
4
6
8
Longitude



The clusters remained consistent in terms of their temperature profiles and separation, indicating that the choice of covariance type did not have a substantial impact on the clustering results for this dataset.

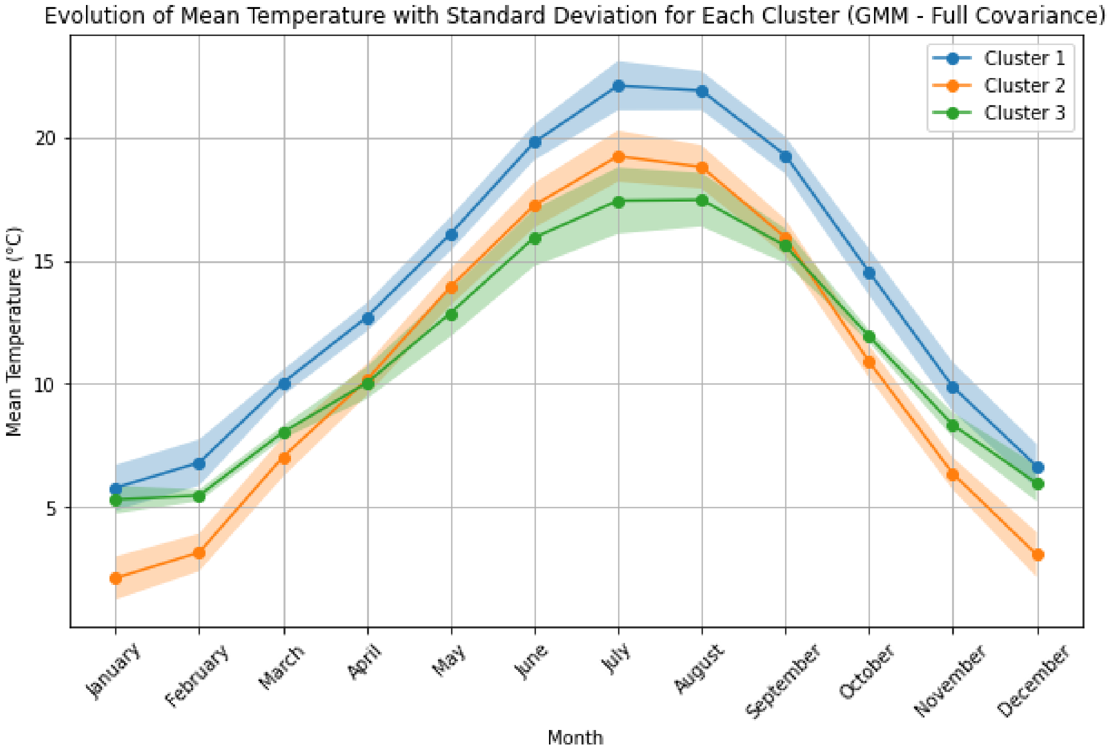
Let's try now to use the full covariance type to see if it will have an impact on the clustering results.

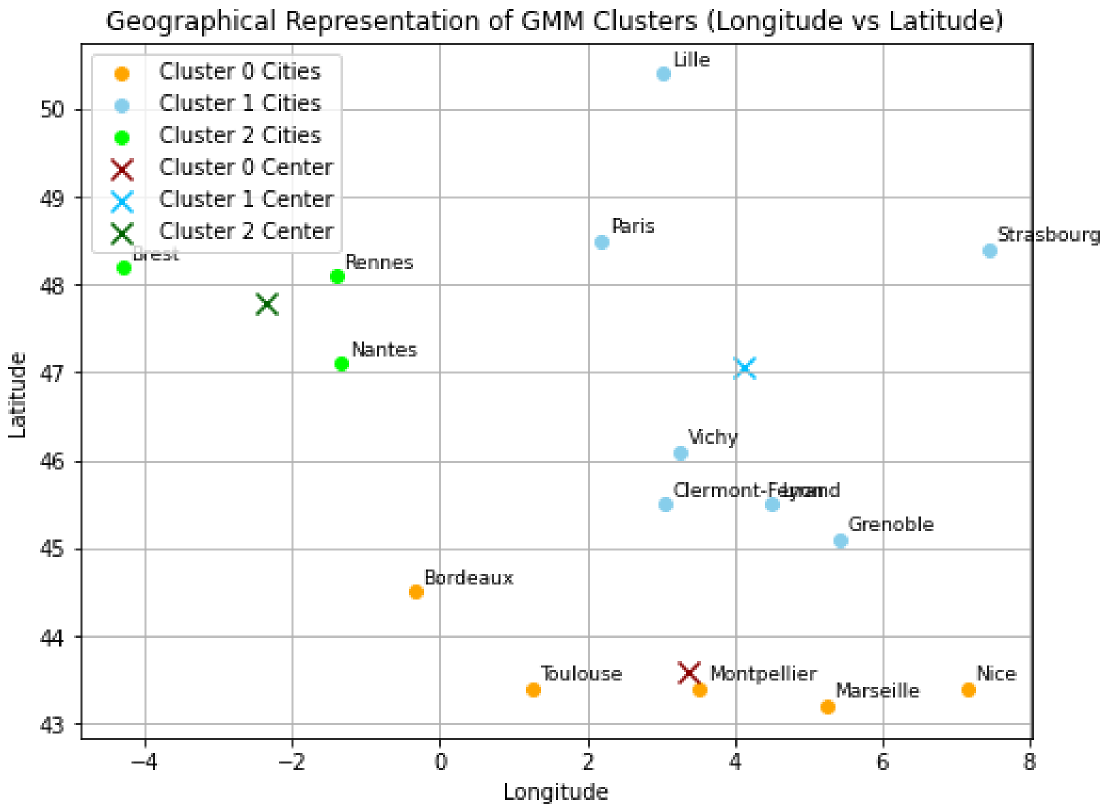
[78]:
```
# Initialize the GMM model with full covariance type
gmm = GaussianMixture(n_components=3, covariance_type='full', random_state=42)

# Fit the GMM model to the temperature data (temperature_matrix)
gmm.fit(temperature_matrix)

# Predict the labels (clusters) for each city based on the GMM model
labels = gmm.predict(temperature_matrix)

# Get the GMM parameters: means and covariances
means = gmm.means_
covariances = gmm.covariances_
std_devs = np.sqrt(covariances)  # Standard deviation for plotting

# Plot the evolution of the mean temperature for each cluster
plt.figure(figsize=(10, 6))
```

```
# For each cluster, plot the mean and fill the standard deviation band
for i in range(gmm.n_components):
    plt.plot(means[i], label=f'Cluster {i + 1}', marker='o')

    # Fill the area between mean ± std_dev (only works for diagonal elements)
    plt.fill_between(np.arange(12),
                     means[i] - np.diag(std_devs[i]),
                     means[i] + np.diag(std_devs[i]),
                     alpha=0.3)

plt.title("Evolution of Mean Temperature with Standard Deviation for Each␣
 ↪Cluster (GMM - Full Covariance)")
plt.xlabel("Month")
plt.ylabel("Mean Temperature (°C)")
plt.xticks(ticks=np.arange(12), labels= varname[:12], rotation=45)
plt.legend()
plt.grid(True)
plt.show()

# Now, plot the geographical representation of the clusters
plt.figure(figsize=(8, 6))

# Scatter plot for cities in Cluster 0
plt.scatter(longitude[labels == 0], latitude[labels == 0], color='orange',␣
 ↪marker='o', label='Cluster 0 Cities')

# Scatter plot for cities in Cluster 1
plt.scatter(longitude[labels == 1], latitude[labels == 1], color='skyblue',␣
 ↪marker='o', label='Cluster 1 Cities')

# Scatter plot for cities in Cluster 2
plt.scatter(longitude[labels == 2], latitude[labels == 2], color='lime',␣
 ↪marker='o', label='Cluster 2 Cities')

# Plot the cluster centers
cluster_0_center = [latitude[labels == 0].mean(), longitude[labels == 0].mean()]
cluster_1_center = [latitude[labels == 1].mean(), longitude[labels == 1].mean()]
cluster_2_center = [latitude[labels == 2].mean(), longitude[labels == 2].mean()]

plt.scatter(cluster_0_center[1], cluster_0_center[0], color='darkred',␣
 ↪label='Cluster 0 Center', marker='x', s=100)
plt.scatter(cluster_1_center[1], cluster_1_center[0], color='deepskyblue',␣
 ↪label='Cluster 1 Center', marker='x', s=100)
plt.scatter(cluster_2_center[1], cluster_2_center[0], color='darkgreen',␣
 ↪label='Cluster 2 Center', marker='x', s=100)
```

```
# Annotate the cities with their names
for i, ville in enumerate(villes):
    plt.text(longitude[i] + 0.1, latitude[i] + 0.1, ville, fontsize=9)

# Add plot titles and labels
plt.title('Geographical Representation of GMM Clusters (Longitude vs Latitude)')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.legend()
plt.grid(True)
plt.show()
```

```
C:\Users\moham\AppData\Local\Temp\ipykernel_7620\1951933018.py:13:
RuntimeWarning: invalid value encountered in sqrt
  std_devs = np.sqrt(covariances)  # Standard deviation for plotting
```





While using the 'full' covariance type, we get as well similar results to the previous ones. The clusters are consistent in terms of their temperature profiles and separation, indicating that the choice of covariance type did not have a substantial impact on the clustering results for this dataset.

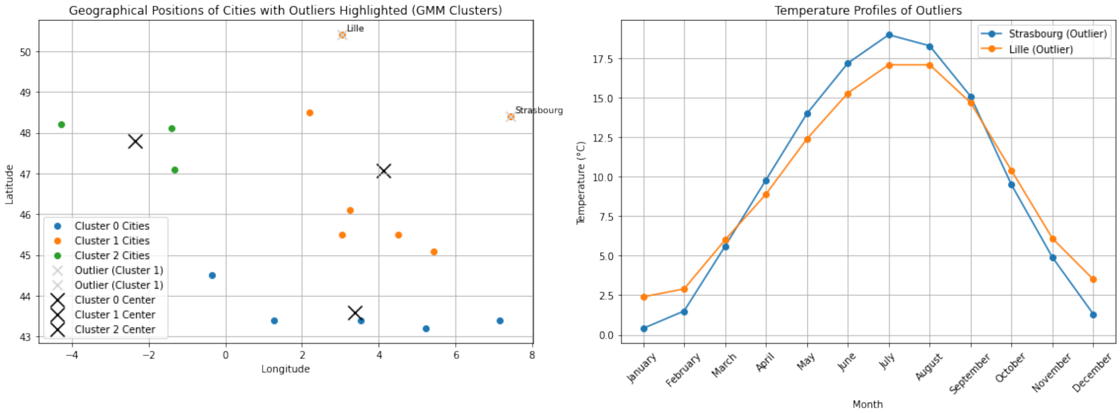
Log Prpobability

[89]:
```python
# Set up and fit the GMM
gmm = GaussianMixture(n_components=3, covariance_type='diag', random_state=42)
gmm.fit(temperature_matrix)

# Compute the log probability (log likelihood) for each sample (city's␣
  ↪temperature over 12 months)
log_probs = gmm.score_samples(temperature_matrix)

# Sort the samples by log probability to find the ones with the smallest scores␣
  ↪(potential outliers)
outlier_indices = np.argsort(log_probs)[:2]  # Take the 2 samples with the␣
  ↪lowest log probability

# Recover the outlier cities (their temperature data and corresponding␣
  ↪geographical info)
outlier_temperatures = temperature_matrix[outlier_indices]
```

```
outlier_latitudes = latitude[outlier_indices]
outlier_longitudes = longitude[outlier_indices]
outlier_villes = [villes[i] for i in outlier_indices]

# Calculate the geographical centers (latitude and longitude) for each cluster
cluster_centers = []
for i in range(gmm.n_components):
    cluster_center = [latitude[labels == i].mean(), longitude[labels == i].
 ↪mean()]
    cluster_centers.append(cluster_center)

# Create a figure for side-by-side plots
fig, axs = plt.subplots(1, 2, figsize=(16, 6))

# --------- Geographical Positions Plot ---------
# Scatter plot for cities in Cluster 0
axs[0].scatter(longitude[labels == 0], latitude[labels == 0],  marker='o',␣
 ↪label='Cluster 0 Cities', cmap='tab10', vmax=9)

# Scatter plot for cities in Cluster 1
axs[0].scatter(longitude[labels == 1], latitude[labels == 1],  marker='o',␣
 ↪label='Cluster 1 Cities', cmap='tab10', vmax=9)

# Scatter plot for cities in Cluster 2
axs[0].scatter(longitude[labels == 2], latitude[labels == 2],  marker='o',␣
 ↪label='Cluster 2 Cities', cmap='tab10', vmax=9)

# Highlight outliers in the corresponding cluster colors but with 'x' marker␣
 ↪and larger size
for i in outlier_indices:
    if labels[i] == 0:
        axs[0].scatter(longitude[i], latitude[i], color='lightgray',␣
 ↪marker='x', s=100, label='Outlier (Cluster 0)')
    elif labels[i] == 1:
        axs[0].scatter(longitude[i], latitude[i], color='lightgray',␣
 ↪marker='x', s=100, label='Outlier (Cluster 1)')
    elif labels[i] == 2:
        axs[0].scatter(longitude[i], latitude[i], color='lightgray',␣
 ↪marker='x', s=100, label='Outlier (Cluster 2)')

# Plot the cluster centers
for i, center in enumerate(cluster_centers):
    axs[0].scatter(center[1], center[0], color='black', label=f'Cluster {i}␣
 ↪Center', marker='x', s=200, edgecolor='white')

# Annotate the outliers with their city names
```

```
for i, ville in enumerate(outlier_villes):
    axs[0].text(outlier_longitudes[i] + 0.1, outlier_latitudes[i] + 0.1, ville, fontsize=9)

# Add plot titles and labels for the first subplot
axs[0].set_title('Geographical Positions of Cities with Outliers Highlighted (GMM Clusters)')
axs[0].set_xlabel('Longitude')
axs[0].set_ylabel('Latitude')
axs[0].legend()
axs[0].grid(True)

# --------- Temperature Profiles Plot ---------
# Plot the temperature profiles of the outliers
for i, idx in enumerate(outlier_indices):
    axs[1].plot(temperature_matrix[idx], label=f'{villes[idx]} (Outlier)', marker='o')

# Add plot titles and labels for the second subplot
axs[1].set_title('Temperature Profiles of Outliers')
axs[1].set_xlabel('Month')
axs[1].set_ylabel('Temperature (°C)')
axs[1].set_xticks(np.arange(12))
axs[1].set_xticklabels(varname[:12], rotation=45)
axs[1].legend()
axs[1].grid(True)

# Show the combined figure with both plots
plt.tight_layout()
plt.show()
```

```
C:\Users\moham\AppData\Local\Temp\ipykernel_7620\2292965173.py:47: UserWarning:
You passed a edgecolor/edgecolors ('white') for an unfilled marker ('x').
Matplotlib is ignoring the edgecolor in favor of the facecolor.  This behavior
may change in the future.
  axs[0].scatter(center[1], center[0], color='black', label=f'Cluster {i}
Center', marker='x', s=200, edgecolor='white')
```

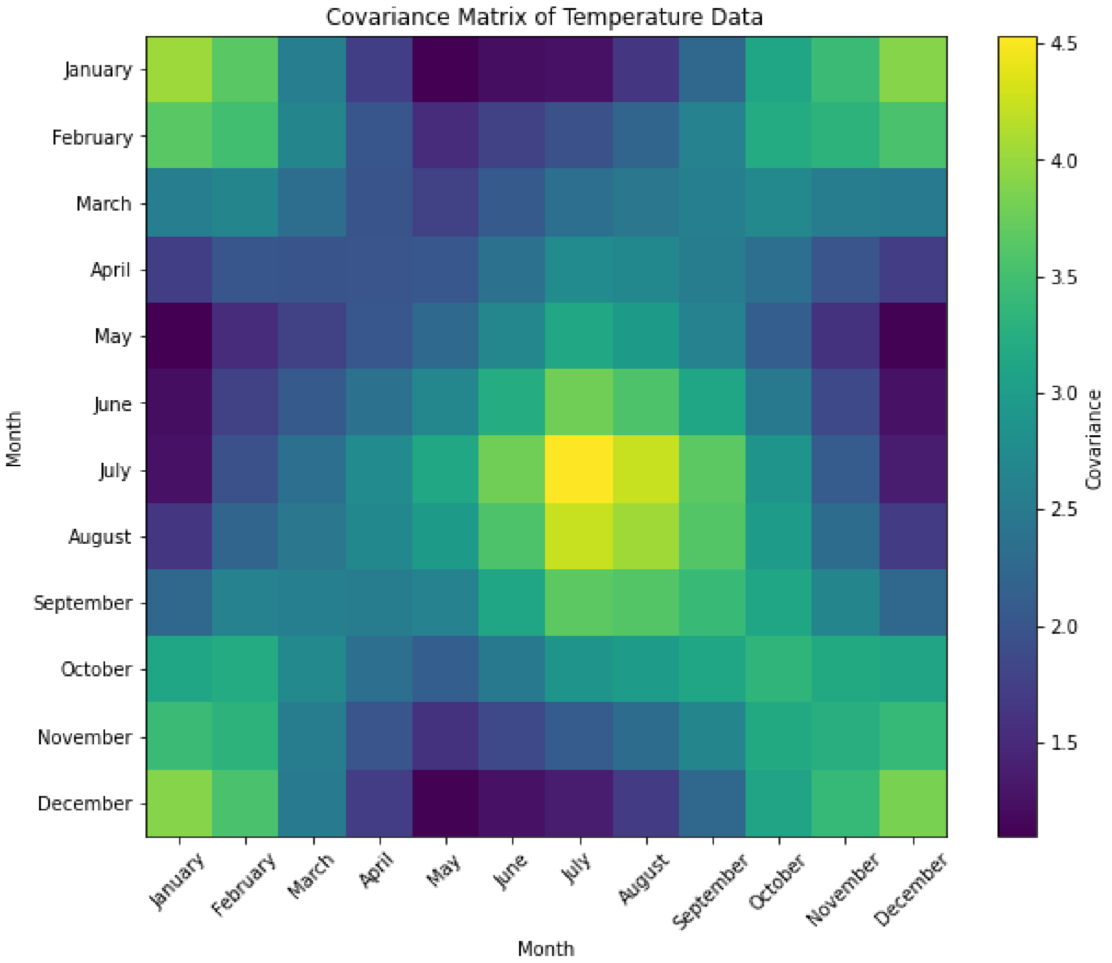


Strasbourg and Lille are the cities with the smallest log probability, which means that they are the cities that are the least likely to be in the clusters that they are in. This is mainly because they are far from the clusters centers and this could be due to the fact that they are cities that are located in the north of France and that they have a climate that is different from the other cities in the same cluster.

1.2.3 Dimensionality Reduction

[131]:

```
# Compute the covariance matrix of the temperature data
cov_matrix = np.cov(temperature_matrix.T)

# Plot the covariance matrix as an image
plt.figure(figsize=(10, 8))
plt.imshow(cov_matrix, cmap='viridis', interpolation='none')
plt.colorbar(label='Covariance')
plt.title('Covariance Matrix of Temperature Data')
plt.xlabel('Month')
plt.ylabel('Month')
plt.xticks(ticks=np.arange(12), labels=varname[:12], rotation=45)
plt.yticks(ticks=np.arange(12), labels=varname[:12])
plt.grid(False)
plt.show()
```

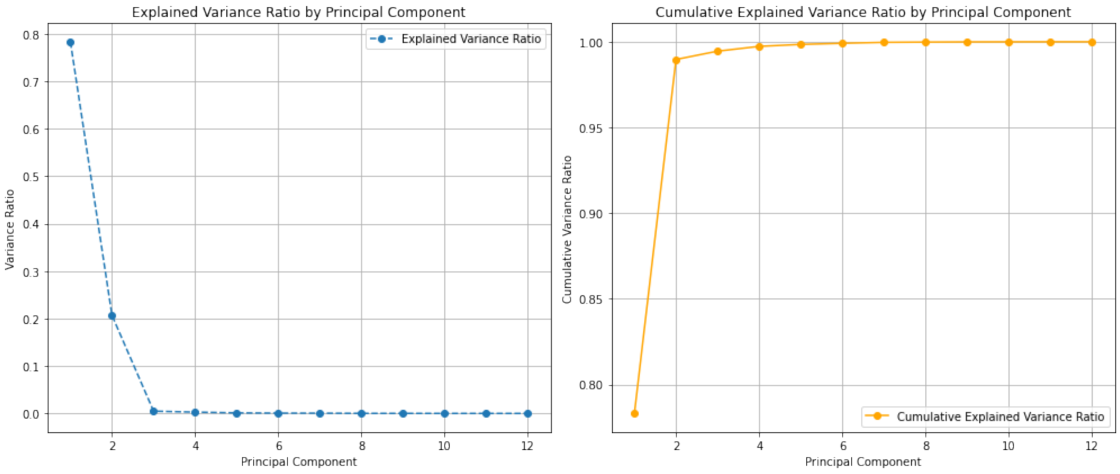


- **Diagonal Line**:
  - The bright yellow diagonal from top-left to bottom-right represents the covariance of each month with itself. Since a variable perfectly correlates with itself, these values are the highest (the variance for each month).
- **Off-Diagonal Covariances**:
  - The off-diagonal squares represent covariances between different months. For example, if a square between December and January (or any pair of months) is bright, it indicates that temperatures in these two months tend to move together (high positive covariance).
  - If a square is darker (towards purple), it indicates that temperatures between those two months have less in common or move independently (low covariance).
- **Seasonal Patterns**:
  - You may notice blocks of similar color, especially for **adjacent months**. This is because temperatures in neighboring months (for example, June-July or October-November) tend to be similar due to gradual seasonal transitions.
  - However, months far apart (e.g., January and July) might show less covariance (darker colors) because temperatures in winter and summer differ significantly.

Principal Components Analysis (PCA)

[147]:

```
# Perform PCA on the temperature data
pca = PCA(n_components=None)
pca.fit(temperature_matrix)

# Recover the explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_

# Calculate the cumulative explained variance ratio
cumulative_explained_variance_ratio = np.cumsum(explained_variance_ratio)

# Create subplots
fig, axs = plt.subplots(1, 2, figsize=(14, 6))  # 1 row, 2 columns

# Plot the explained variance ratio
axs[0].plot(np.arange(1, len(explained_variance_ratio) + 1),␣
 ↪explained_variance_ratio,
            marker='o', linestyle='--', label='Explained Variance Ratio')
axs[0].set_title('Explained Variance Ratio by Principal Component')
axs[0].set_xlabel('Principal Component')
axs[0].set_ylabel('Variance Ratio')
axs[0].legend()
axs[0].grid(True)

# Plot the cumulative explained variance ratio
axs[1].plot(np.arange(1, len(cumulative_explained_variance_ratio) + 1),
            cumulative_explained_variance_ratio, marker='o', linestyle='-',
            label='Cumulative Explained Variance Ratio', color='orange')
axs[1].set_title('Cumulative Explained Variance Ratio by Principal Component')
axs[1].set_xlabel('Principal Component')
axs[1].set_ylabel('Cumulative Variance Ratio')
axs[1].legend()
axs[1].grid(True)

# Adjust layout
plt.tight_layout()

# Show the plot
plt.show()
```

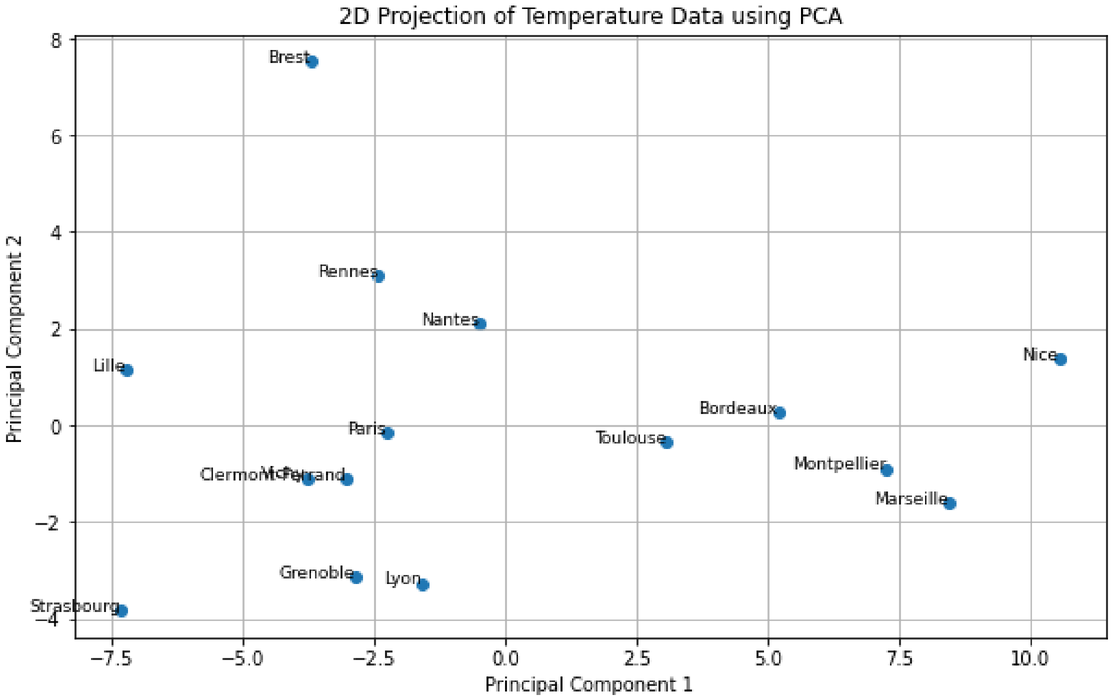


1. **Explained Variance Ratio (Blue Dots and Dashed Line)**:
   - The blue dashed line represents the **explained variance ratio** for each principal component (PC). This value shows the proportion of the total dataset's variance captured by each component.
   - The sharp decrease after the first two components indicates that **most of the variance** is captured by the first few components, with subsequent components capturing much less.
2. **Cumulative Explained Variance Ratio (Orange Line)**:
   - The orange line shows the **cumulative explained variance ratio**, which represents the total variance captured by the first p components.
   - As you can see, the curve plateaus after the second component, indicating that almost all the variance is captured by the first two principal components.

- The cumulative explained variance ratio reaches **close to 100% after the second component** (around 95%-100%), meaning that when projecting the data onto 2 dimensions (the first 2 principal components), you retain **most of the original information**.
- This suggests that the first two components are highly informative, and projecting onto just two dimensions would capture **almost all of the variability** in the original high-dimensional data.

[138]:

```
# Perform PCA to reduce the dimensionality to 2
pca = PCA(n_components=2)
projected_data = pca.fit_transform(temperature_matrix)

# Plot the projected samples
plt.figure(figsize=(10, 6))
plt.scatter(projected_data[:, 0], projected_data[:, 1], marker='o')

# Annotate each point with the city name
for i, city in enumerate(cities):
    plt.text(projected_data[i, 0], projected_data[i, 1], city, fontsize=9,␣
 ↪ha='right')

plt.title("2D Projection of Temperature Data using PCA")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.grid(True)
plt.show()
```

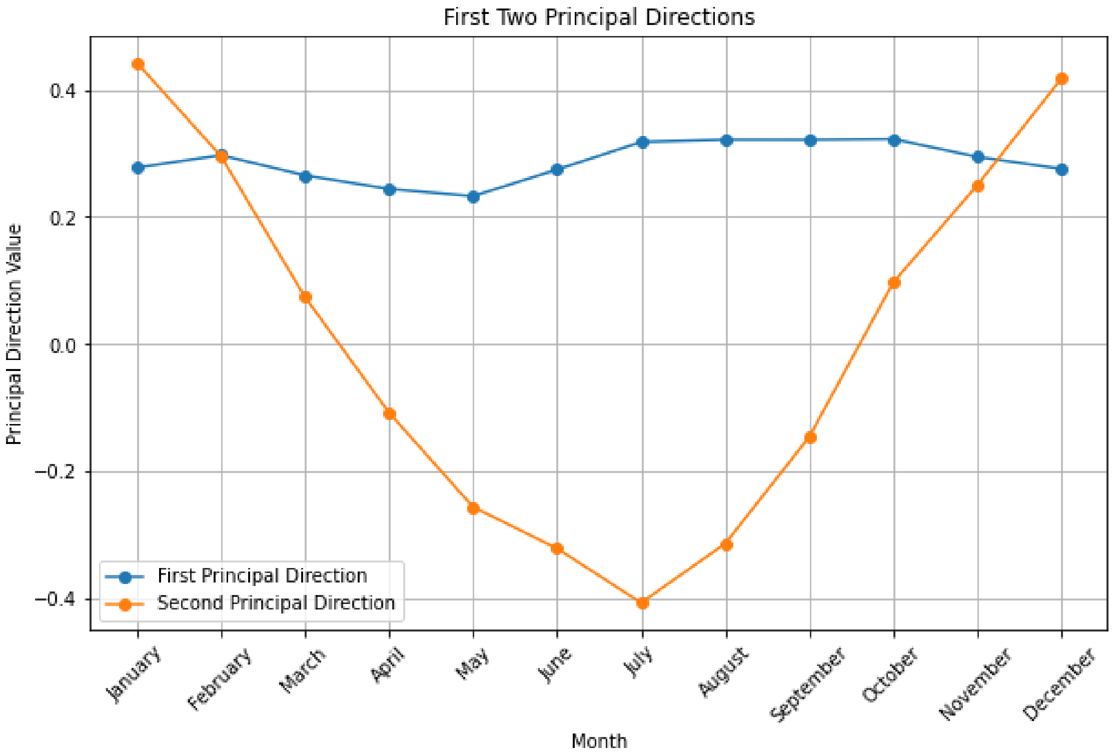


The PCA plot shows French cities projected onto two dimensions based on their temperature data. Cities that are close together, like **Nice and Marseille** or **Lyon and Grenoble**, likely have similar climates, which aligns with their geographic regions. Outliers, such as **Brest** and **Nice**.

While the plot loosely correlates with geographic locations, it primarily captures **climatic similarities**. PCA successfully reduces the data while preserving major climate-related differences, even if geographic proximity isn't perfectly reflected.

[141]:
```
# Perform PCA to reduce the dimensionality to 2
pca = PCA(n_components=2)
pca.fit(temperature_matrix)

# Get the first two principal directions
principal_directions = pca.components_

# Plot the principal directions
```

```python
plt.figure(figsize=(10, 6))
plt.plot(principal_directions[0], marker='o', label='First Principal Direction')
plt.plot(principal_directions[1], marker='o', label='Second Principal Direction')
plt.title('First Two Principal Directions')
plt.xlabel('Month')
plt.ylabel('Principal Direction Value')
plt.xticks(ticks=np.arange(12), labels=varname[:12], rotation=45)
plt.legend()
plt.grid(True)
plt.show()

# Interpretation
print("First Principal Direction:", principal_directions[0])
print("Second Principal Direction:", principal_directions[1])
```



```
First Principal Direction: [0.27765046 0.29726177 0.26549892 0.24426184
0.23276055 0.27461466
 0.31796069 0.32160147 0.32149157 0.32257588 0.2946594  0.27575128]
Second Principal Direction: [ 0.44271144  0.29596995  0.07437784 -0.10733591
-0.25565325 -0.3209949
 -0.40635887 -0.31355378 -0.1457905   0.09718023  0.25000485  0.41750952]
```

In the plot, the first and second principal directions (components) represent the directions in the original space along which the variance is maximized.

First Principal Direction (Blue Line):

- The first principal component shows a fairly stable trend with small fluctuations.
- Moving along this direction in the original space would capture the most dominant variation across the features. Since this component is fairly flat, it suggests that no single feature (or month) dominates the overall variance. It could indicate consistent seasonal variation across months or cities.

Second Principal Direction (Orange Line):

- The second principal component displays a more pronounced "V" shape, suggesting a sharp dip in the middle features and rising values on the extremes.
- Moving along this direction in the original space means we are moving along a pattern that contrasts early and late months with the middle ones. It could indicate a seasonal effect, where the variance between the middle months (e.g., summer) and the extremes (e.g., winter) is significant.

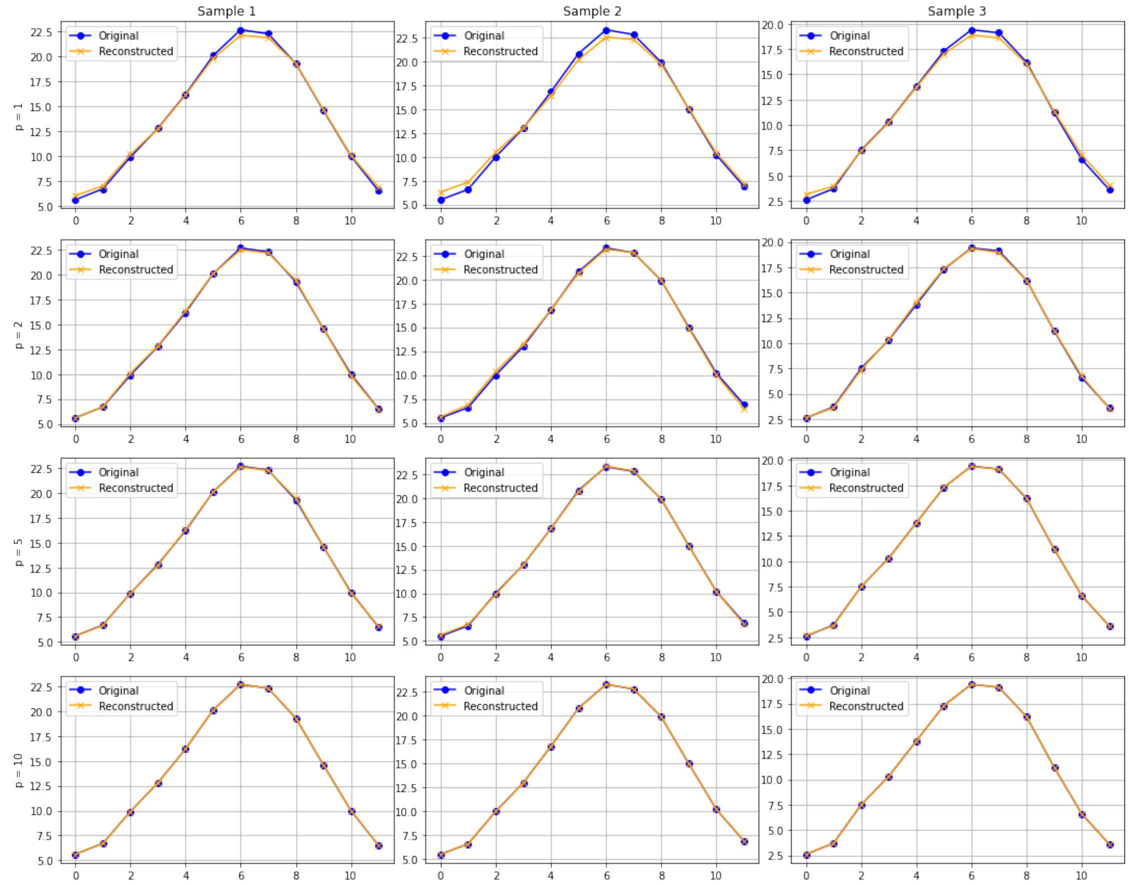
Impact:

- **Along the First Principal Direction**: It captures the primary trend across all features. Moving in this direction emphasizes the overall variance spread in the dataset, which could be linked to a broad pattern like yearly trends in temperature.

- **Along the Second Principal Direction**: It captures secondary trends or specific deviations, such as seasonal peaks or unusual months (like those at the extremes). It reveals how certain months differ from the overall trend established by the first component.

[149]:

```
# Define the number of samples to visualize
num_samples_to_plot = 3

# Select random indices for the samples to visualize
random_indices = np.random.choice(range(temperature_matrix.shape[0]),␣
 ↪num_samples_to_plot, replace=False)

# Define the subspace dimensions to test
subspace_dimensions = [1, 2, 5, 10]

# Set up the figure for plotting
plt.figure(figsize=(15, len(subspace_dimensions) * 3))

# Loop over the subspace dimensions
for i, p in enumerate(subspace_dimensions):
    # Perform PCA with p components
    pca = PCA(n_components=p)
    projected_data = pca.fit_transform(temperature_matrix)
    reconstructed_data = pca.inverse_transform(projected_data)
    # Plot the original and reconstructed samples together
    for j, idx in enumerate(random_indices):
        plt.subplot(len(subspace_dimensions), num_samples_to_plot, i *␣
 ↪num_samples_to_plot + j + 1)

        # Original sample in blue
        plt.plot(temperature_matrix[idx], marker='o', label='Original',␣
 ↪color='blue')

        # Reconstructed sample in orange
        plt.plot(reconstructed_data[idx], marker='x', label='Reconstructed',␣
 ↪color='orange')

        # Add labels and titles
        if j == 0:
            plt.ylabel(f'p = {p}')
        if i == 0:
            plt.title(f'Sample {j + 1}')

        plt.legend()
        plt.grid(True)

# Adjust layout and display the plot
plt.tight_layout()
plt.show()
```



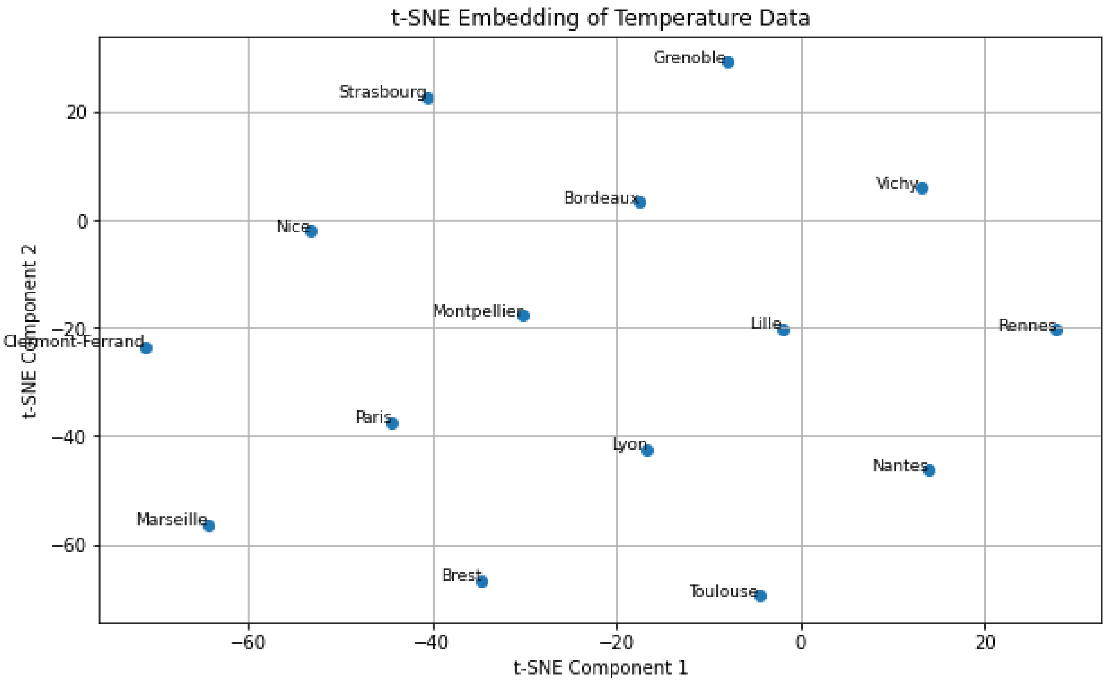
It's clear that from p=2, the reconstructed data is very close to the original data, indicating that the first two principal components capture most of the variance in the dataset. This suggests that the temperature data for French cities can be effectively represented in a lower-dimensional space while retaining the essential information about temperature patterns and variations.

1.2.4 Manifold Learning : TSNE

[151]:
```
from sklearn.manifold import TSNE

# Compute t-SNE embedding for the temperature dataset
tsne_temp = TSNE(n_components=2, random_state=42)
temperature_embedding = tsne_temp.fit_transform(temperature_matrix)

# Plot the t-SNE embedding for the temperature dataset
plt.figure(figsize=(10, 6))
plt.scatter(temperature_embedding[:, 0], temperature_embedding[:, 1],␣
  ↪marker='o')

# Annotate each point with the city name
```

```
for i, city in enumerate(cities):
    plt.text(temperature_embedding[i, 0], temperature_embedding[i, 1], city,␣
 ↪fontsize=9, ha='right')

plt.title("t-SNE Embedding of Temperature Data")
plt.xlabel("t-SNE Component 1")
plt.ylabel("t-SNE Component 2")
plt.grid(True)
plt.show()
```

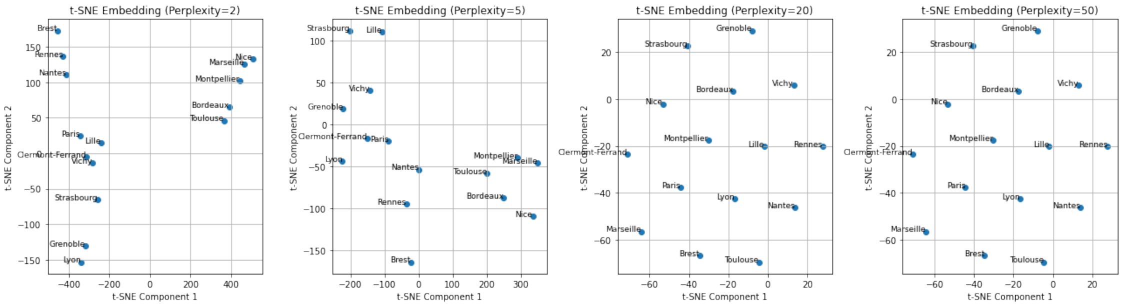
```
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:795: FutureWarning: The default
initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:805: FutureWarning: The default learning
rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
```



[155]:
```
from sklearn.manifold import TSNE

# Define different values of perplexity to test
perplexities = [2, 5, 20,50]

# Set up the figure for plotting
```

```
plt.figure(figsize=(18, 5))

# Loop over the different values of perplexity
for i, perplexity in enumerate(perplexities):
    # Compute t-SNE embedding for the temperature dataset with the current perplexity
    tsne_temp = TSNE(n_components=2, perplexity=perplexity, random_state=42)
    temperature_embedding = tsne_temp.fit_transform(temperature_matrix)

    # Create a subplot for each perplexity value
    plt.subplot(1, len(perplexities), i + 1)

    # Plot the t-SNE embedding for the temperature dataset
    plt.scatter(temperature_embedding[:, 0], temperature_embedding[:, 1], marker='o')

    # Annotate each point with the city name
    for j, city in enumerate(cities):
        plt.text(temperature_embedding[j, 0], temperature_embedding[j, 1], city, fontsize=9, ha='right')

    # Add plot titles and labels
    plt.title(f"t-SNE Embedding (Perplexity={perplexity})")
    plt.xlabel("t-SNE Component 1")
    plt.ylabel("t-SNE Component 2")
    plt.grid(True)

# Adjust layout and show the plot
plt.tight_layout()
plt.show()
```

```
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:795: FutureWarning: The default
initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:805: FutureWarning: The default learning
rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:795: FutureWarning: The default
initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:805: FutureWarning: The default learning
rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:795: FutureWarning: The default
initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:805: FutureWarning: The default learning
rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:795: FutureWarning: The default
initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:805: FutureWarning: The default learning
rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
```



Using t-SNE with **perplexity = 2**, three distinct clusters emerged, representing regions with similar temperature profiles. This aligns with findings from k-means and Gaussian Mixture Models (GMM), indicating that the temperature data has identifiable structures.

However, as the perplexity increases, the clusters become less interpretable. In fact, since perplexity is a measure of the number of nearest neighbors used in the algorithm, higher perplexity values can lead to more global structures being captured, potentially obscuring local relationships.

1.3 *Digits dataset*

Loading data

[25]:
```
# Load digits data
digits = np.load("digits.npz")
```

Data exploration

[26]:
```
digits.files
```

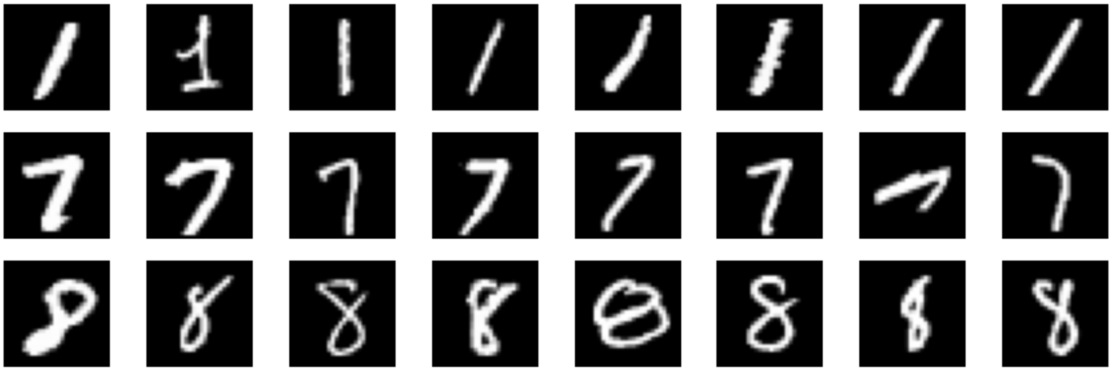
[26]:
```
['xt', 'yt', 'y', 'x']
```

[27]:

```
# Load xt, yt, y and x from the dataset
xt = digits['xt']
yt = digits['yt']
x2 = digits['x']
y2 = digits['y']


# Perform one simple pre-processing that scales the values between [0, 1]
x2 = x2 / 255
```

[28]:

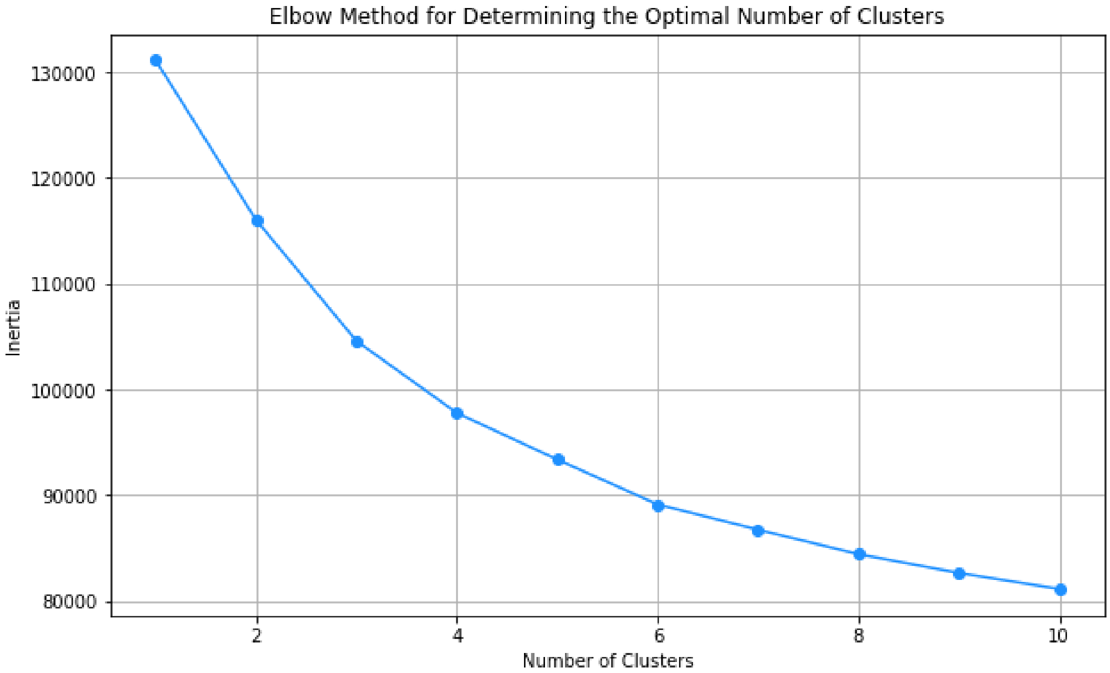
```
# Set up the figure
plt.figure(figsize=(15, 5))

# Define the number of samples
samples_per_digit = 8

# Get the indices for each digit (1, 7, and 8) from y2
indices_1 = np.where(y2 == 1)[0] # np.where returns a tuple with the first␣
 ↪element being the indices
indices_7 = np.where(y2 == 7)[0]
indices_8 = np.where(y2 == 8)[0]

# Randomly choose samples from each digit
random_indices_1 = np.random.choice(indices_1, samples_per_digit, replace=False)
random_indices_7 = np.random.choice(indices_7, samples_per_digit, replace=False)
random_indices_8 = np.random.choice(indices_8, samples_per_digit, replace=False)

# Plot the samples, horizontally for each digit
for i in range(samples_per_digit):
    # Plot digit 1 in the first row
    plt.subplot(3, samples_per_digit, i + 1)
    plt.imshow(x2[random_indices_1[i]].reshape(28, 28), cmap='gray')
    plt.axis('off')

    # Plot digit 7 in the second row
    plt.subplot(3, samples_per_digit, samples_per_digit + i + 1)
    plt.imshow(x2[random_indices_7[i]].reshape(28, 28), cmap='gray')
    plt.axis('off')

    # Plot digit 8 in the third row
    plt.subplot(3, samples_per_digit, 2 * samples_per_digit + i + 1)
    plt.imshow(x2[random_indices_8[i]].reshape(28, 28), cmap='gray')
    plt.axis('off')

# Display the plot
plt.show()
```



K-means clustering Since we are dealing with 3 labels, so we will set k=3 for the K-means clustering algorithm. But I wanted to show you the elbow method to see how it works.

Elbow method for determining the optimum number of clusters

[29]:
```
inertias = []
k_range = range(1, 11)

for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(x2)
    inertias.append(kmeans.inertia_)

# Plot the inertia curve as a function of the number of clusters
plt.figure(figsize=(10, 6))
plt.plot(k_range, inertias, marker='o', color='dodgerblue')
plt.title("Elbow Method for Determining the Optimal Number of Clusters")
plt.xlabel("Number of Clusters")
plt.ylabel("Inertia")
plt.grid(True)
plt.show()
```

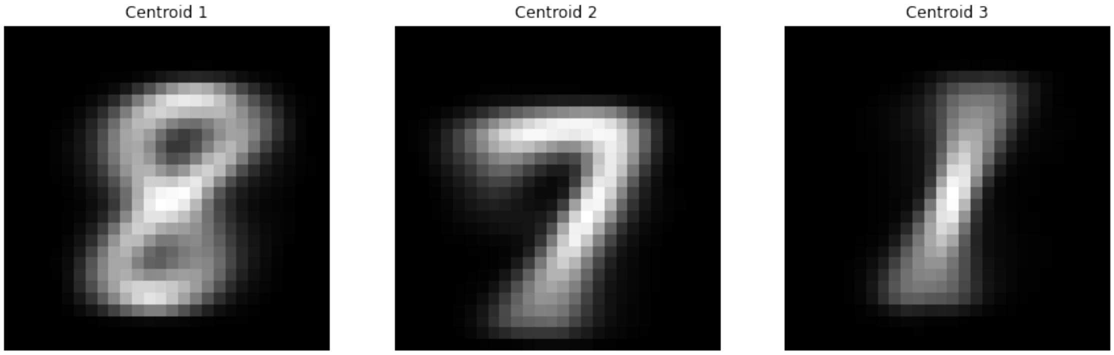


We can't see a clear elbow in the plot, but we can observe that the inertia value decreases more rapidly when the number of clusters increases from 2 to 3. This suggests that the optimal number of clusters may be 3, as it leads to a significant reduction in inertia compared to the case of 2 clusters.

k_optimal = 3

[30]:
```
# Choose the number of clusters
k_optimal = 3

# Apply the KMeans algorithm
kmeans = KMeans(n_clusters=k_optimal, random_state=42) # random_state=42 for␣
 ↪reproducibility
clusters = kmeans.fit_predict(x2)
```

[31]:
```
# Store the clusters centroids
centroids = kmeans.cluster_centers_

# Plot the centroids images corresponding to each cluster
plt.figure(figsize=(15, 5))
for i, centroid in enumerate(centroids):
    plt.subplot(1, k_optimal, i + 1)
    plt.imshow(centroid.reshape(28, 28), cmap='gray')
    plt.axis('off')
    plt.title(f'Centroid {i + 1}')
```



We can observe that the K-means algorithm successfully clusters the digits into 3 distinct groups and samples are clustered by class on digits. The algorithm is trained based on the pixel values of the images and showed a good performance in separating the digits into different clusters.

Clusters centroids doesn't represent any digit because they are the average of the pixel values of the images in each cluster. So, while plotting the centroids, we can recognize the shape of the digits, but they doesn't represent any true image.

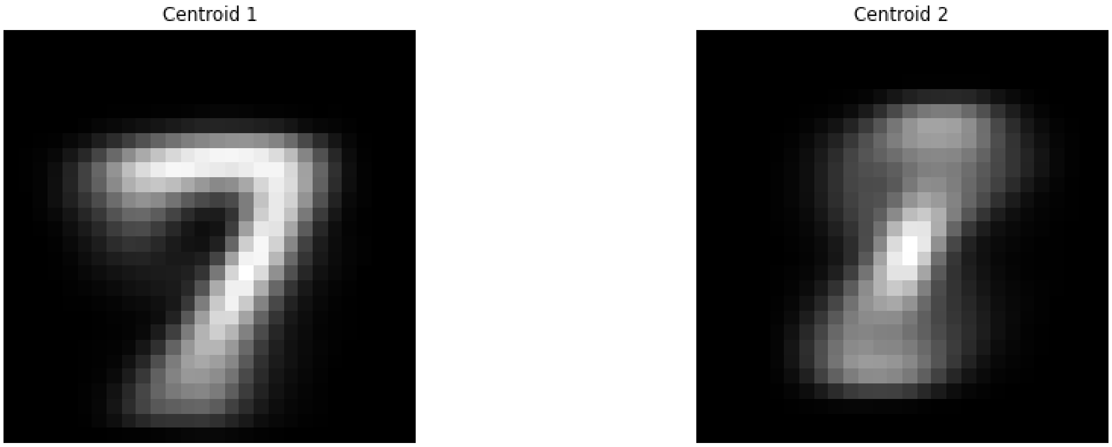
k_optimal = 2

[32]:

```
# Choose the number of clusters
k_optimal = 2

# Apply the KMeans algorithm
kmeans = KMeans(n_clusters=k_optimal, random_state=42) # random_state=42 for␣
  ↪reproducibility
clusters = kmeans.fit_predict(x2)
```

[33]:

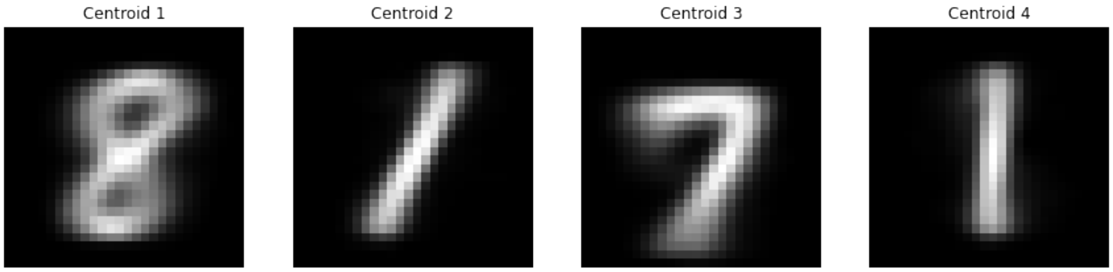
```
# Store the clusters centroids
centroids = kmeans.cluster_centers_

# Plot the centroids images corresponding to each cluster
plt.figure(figsize=(15, 5))
for i, centroid in enumerate(centroids):
    plt.subplot(1, k_optimal, i + 1)
    plt.imshow(centroid.reshape(28, 28), cmap='gray')
    plt.axis('off')
    plt.title(f'Centroid {i + 1}')
```



Here, the k-means algorithm is trained with k=2 and supposed that digits corresponding to 1 and 8 are in the same cluster.

k_optimal = 4

[34]:
```
# Choose the number of clusters
k_optimal = 4

# Apply the KMeans algorithm
kmeans = KMeans(n_clusters=k_optimal, random_state=42) # random_state=42 for reproducibility
clusters = kmeans.fit_predict(x2)

# Store the clusters centroids
centroids = kmeans.cluster_centers_

# Plot the centroids images corresponding to each cluster
plt.figure(figsize=(15, 5))
for i, centroid in enumerate(centroids):
    plt.subplot(1, k_optimal, i + 1)
    plt.imshow(centroid.reshape(28, 28), cmap='gray')
    plt.axis('off')
    plt.title(f'Centroid {i + 1}')
```

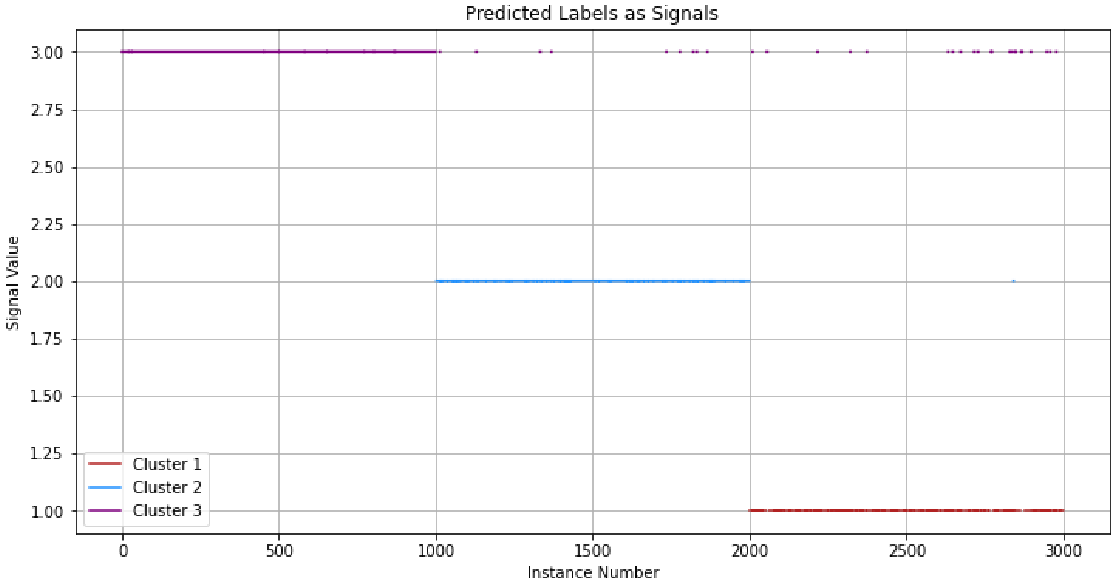


Here, the k-means algorithm is trained with k=4 and supposed that there are 2 clusters for digits corresponding to 1. So, it's obvious that the best number of clusters is 3 where each cluster corresponds to a digit class.

---

[35]:

```
# Train K-means model (assuming 3 clusters for digits 1, 7, and 8)
kmeans = KMeans(n_clusters=3, random_state=42)
predicted_labels = kmeans.fit_predict(x2)

# Create a signal array
signal_length = len(predicted_labels)
signal = np.zeros(signal_length)

# Assign values to the signal based on cluster membership
for i in range(signal_length):
    signal[i] = predicted_labels[i] + 1  # Add 1 to match cluster index to␣
 ↪signal value

# Set colors for each cluster
colors = ['firebrick', 'dodgerblue', 'purple']  # One color for each cluster

# Plot the signal
plt.figure(figsize=(12, 6))
for j in range(3):  # Iterate through each cluster
    plt.plot(np.where(predicted_labels == j, signal, np.nan), color=colors[j],␣
 ↪label=f'Cluster {j + 1}')

plt.title('Predicted Labels as Signals')
plt.xlabel('Instance Number')
plt.ylabel('Signal Value')
plt.legend()
plt.grid(True)
plt.show()
```



Since the samples are ordered by class in the dataset, a clustering respecting the true class is piecewise constant. So we can conclude that the k-means performs well on this dataset and is able to separate the digits into distinct clusters.

Remark **:** We can notice that digits corresponding to 1 are somtimes assigned to other clusters. This is due to the fact that the digit 1 can be written in different ways and the algorithm can't distinguish between them. So, the k-means algorithm is sensitive to the way the data is represented and the features used for clustering.

---

[36]:
```
# Compute the quality of the clustering using the ground truth labels

# Reshape the predicted labels
y2 = y2.reshape(-1)

# Compute the rand score
rand_index = rand_score(y2, predicted_labels)
print(f"Rand Score: {rand_index}")

# Compute the Adjusted Rand score
ari_rand_index = adjusted_rand_score(y2, predicted_labels)
print(f"Adjusted Rand Index: {ari_rand_index}")
```

```
Rand Score: 0.8808471712793153
Adjusted Rand Index: 0.7337556340018966
```

The **Rand Index** is a measure of the similarity between two data clusterings. It considers all pairs of samples and counts the number of pairs that are assigned to the same or different clusters in the predicted and true clusterings. The rand index ranges from 0 to 1, where 1 indicates perfect agreement between the predicted and true clusterings. It's calculated as follows:

$$RI = \frac{(number\_of\_agreeing\_pairs)}{(number\_of\_pairs)}.$$

In this case, the rand index is close to 1, indicating that the K-means algorithm performs well in clustering the digits dataset.

The **Adjusted Rand Index (ARI)** is a measure of the similarity between two data clusterings that considers all pairs of samples and counts the number of pairs that are assigned to the same or different clusters in the predicted and true clusterings. The ARI ranges from -1 to 1, where 1 indicates perfect agreement between the predicted and true clusterings, 0 indicates random clustering, and negative values indicate disagreement between the clusterings. It's calculated as follows:

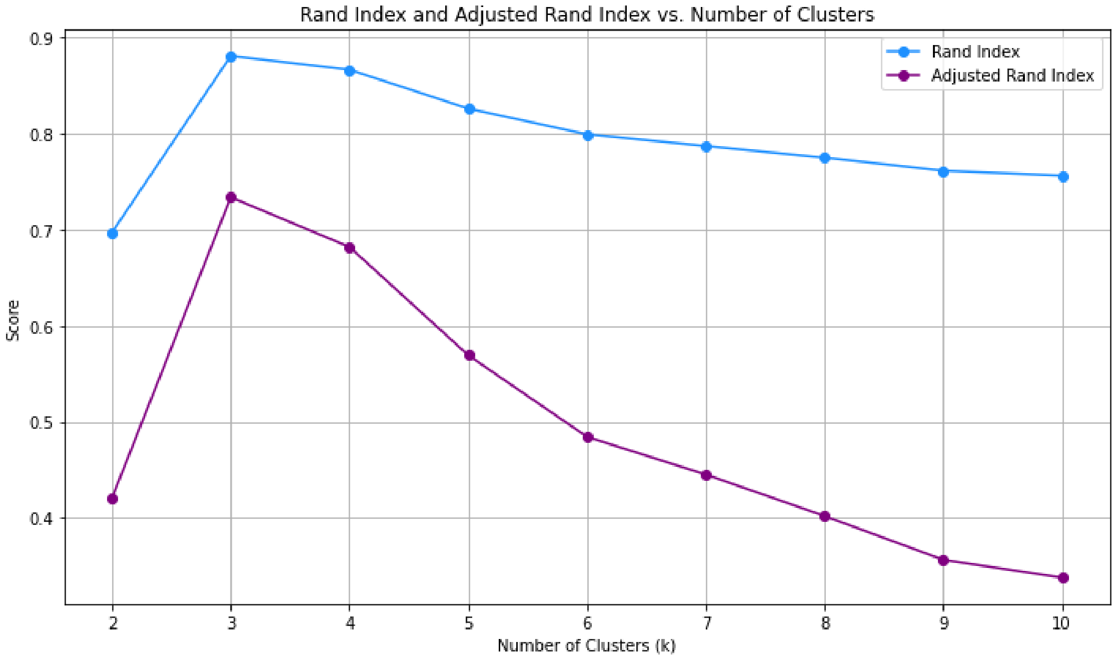
$$ARI = \frac{RI - Expected\_RI}{max(RI) - Expected\_RI}.$$

Next, we will adjust the value of K and repeat the previous two steps to identify the optimal values that yield the best results based on the Rand Index and Adjusted Rand Index.

[37]:

```
# Define a range of k values to test
k_values = range(2, 11)

# Initialize lists to store the Rand Index and Adjusted Rand Index for each k
rand_indices = []
adjusted_rand_indices = []

# Loop over the range of k values
for k in k_values:
    # Apply the KMeans algorithm
    kmeans = KMeans(n_clusters=k, random_state=42)
    clusters = kmeans.fit_predict(x2)

    # Compute the Rand Index
    rand_index = rand_score(y2, clusters)
    rand_indices.append(rand_index)

    # Compute the Adjusted Rand Index
    adjusted_rand_index = adjusted_rand_score(y2, clusters)
    adjusted_rand_indices.append(adjusted_rand_index)

# Plot both the Rand Index and Adjusted Rand Index for each k in the same figure
plt.figure(figsize=(10, 6))

plt.plot(k_values, rand_indices, marker='o', color='dodgerblue', label="Rand␣
  ↪Index")
```

```
plt.plot(k_values, adjusted_rand_indices, marker='o', color='purple', label="Adjusted Rand Index")

plt.title("Rand Index and Adjusted Rand Index vs. Number of Clusters")
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Score")
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()
```



Interpretation :

- **Rand Index (Blue Curve)**:
  - The Rand Index increases rapidly at K=3, indicating a good match when the number of clusters aligns with the three classes of digits (1, 7, and 8). After K=3, the performance plateaus or declines slightly, suggesting that adding more clusters doesn't improve clustering quality.
- **Adjusted Rand Index (Purple Curve)**:
  - The Adjusted Rand Index also peaks at K=3, suggesting this is the optimal number of clusters for separating the digits. Beyond K=3, it decreases, indicating that increasing the number of clusters leads to incorrect splits that don't correspond well to the actual data structure.

1.3.1 Density Estimation

K = 3

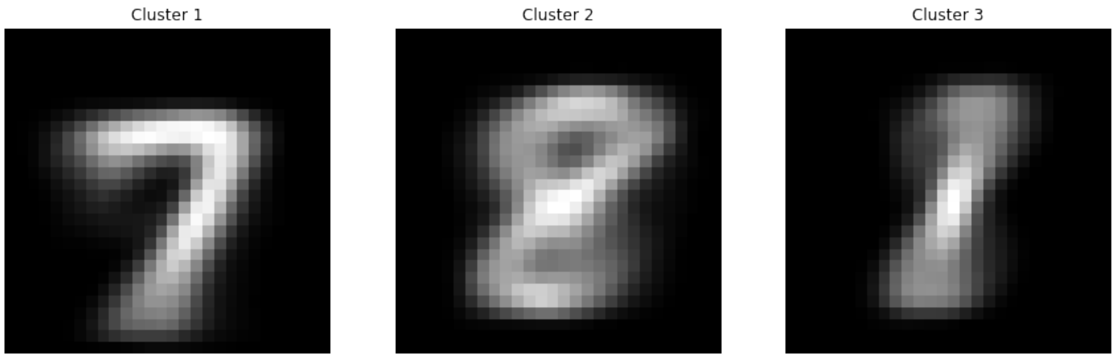
[99]:

```
# Initialize the GMM model with 3 components and diagonal covariance type
gmm = GaussianMixture(n_components=3, covariance_type='diag', random_state=42)

# Fit the GMM model to the digits data
gmm.fit(x2)

# Predict the cluster labels for each sample
gmm_labels = gmm.predict(x2)

# Print the means and covariances of the GMM components
print("Means of GMM components:\n", gmm.means_)
print("\nCovariances of GMM components:\n", gmm.covariances_)
```

```
Means of GMM components:
 [[0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]]

Covariances of GMM components:
 [[1.e-06 1.e-06 1.e-06 … 1.e-06 1.e-06 1.e-06]
 [1.e-06 1.e-06 1.e-06 … 1.e-06 1.e-06 1.e-06]
 [1.e-06 1.e-06 1.e-06 … 1.e-06 1.e-06 1.e-06]]
```

[100]:

```
# Plot the centers of the Gaussian distributions in the mixture
plt.figure(figsize=(15, 5))

# Iterate through each component (cluster) in the GMM
for i, mean in enumerate(gmm.means_):
    plt.subplot(1, gmm.n_components, i + 1)
    plt.imshow(mean.reshape(28, 28), cmap='gray')
    plt.axis('off')
    plt.title(f'Cluster {i + 1}')

plt.show()
```

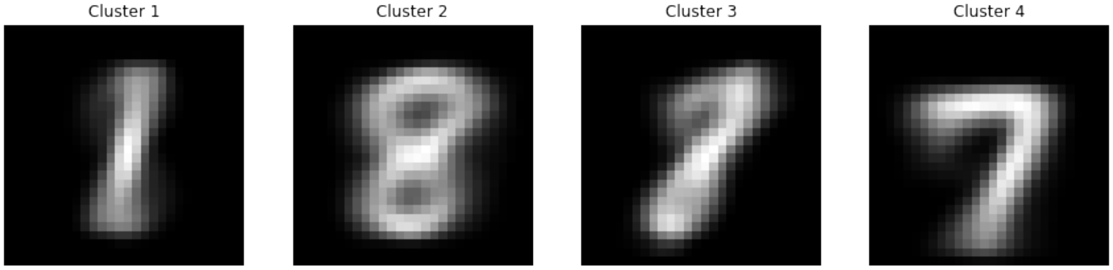


The centers of the Gaussian distributions in the mixture model represent the mean pixel values for each cluster. It's clear that the clusters are well separated and represent distinct groups of digits (1, 7, and 8) even though the centroids don't correspond to any true digit image.

K = 4

[102]:
```
# Initialize the GMM model with 3 components and diagonal covariance type
gmm = GaussianMixture(n_components=4, covariance_type='diag', random_state=42)

# Fit the GMM model to the digits data
gmm.fit(x2)

# Predict the cluster labels for each sample
gmm_labels = gmm.predict(x2)

# Print the means and covariances of the GMM components
print("Means of GMM components:\n", gmm.means_)
print("\nCovariances of GMM components:\n", gmm.covariances_)

# Plot the centers of the Gaussian distributions in the mixture
plt.figure(figsize=(15, 5))

# Iterate through each component (cluster) in the GMM
for i, mean in enumerate(gmm.means_):
    plt.subplot(1, gmm.n_components, i + 1)
    plt.imshow(mean.reshape(28, 28), cmap='gray')
    plt.axis('off')
    plt.title(f'Cluster {i + 1}')

plt.show()
```

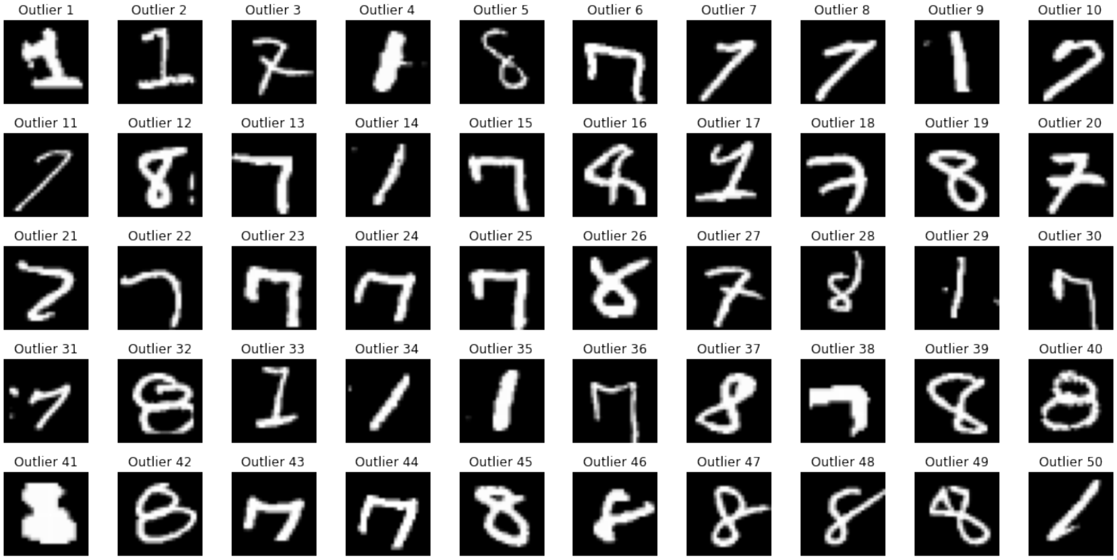
```
Means of GMM components:
 [[0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]]

Covariances of GMM components:
 [[1.e-06 1.e-06 1.e-06 … 1.e-06 1.e-06 1.e-06]
 [1.e-06 1.e-06 1.e-06 … 1.e-06 1.e-06 1.e-06]
 [1.e-06 1.e-06 1.e-06 … 1.e-06 1.e-06 1.e-06]
 [1.e-06 1.e-06 1.e-06 … 1.e-06 1.e-06 1.e-06]]
```



When changing the number of clusters to 4, we can see that the model creates two clusters for the digit 1. This is because the digit 1 can be written in different ways, leading to variations in the pixel values. So, it's clear that the best number of clusters is 3 where each cluster corresponds to a digit class.

[108]:

```
# Initialize the GMM model with 3 components and diagonal covariance type
gmm = GaussianMixture(n_components=3, covariance_type='diag', random_state=42)

# Fit the GMM model to the digits data
gmm.fit(x2)

# Compute the log probability (log likelihood) for each sample
log_probs = gmm.score_samples(x2)

# Sort the samples by log probability to find the ones with the smallest scores␣
 ↪(potential outliers)
outlier_indices = np.argsort(log_probs)[:50]  # Take the 50 samples with the␣
 ↪lowest log probability

# Recover the outlier samples
outlier_samples = x2[outlier_indices]

# Plot the outlier samples
n_outliers = outlier_samples.shape[0]
n_cols = 10  # Number of columns in the plot
n_rows = n_outliers // n_cols + (n_outliers % n_cols > 0)  # Calculate number␣
 ↪of rows needed

plt.figure(figsize=(15, n_rows * 1.5))

for i, sample in enumerate(outlier_samples):
    plt.subplot(n_rows, n_cols, i + 1)
    plt.imshow(sample.reshape(28, 28), cmap='gray')
    plt.axis('off')
    plt.title(f'Outlier {i + 1}')
plt.tight_layout()  # Adjust layout to prevent overlap
plt.show()
```



This method works well to identify outliers in the dataset, as it can detect samples that are not well written or that deviate significantly from the average pixel values of the digits. These outliers can be removed in order to improve the clustering performance and obtain more accurate results after training the model.

[111]:
```
# Initialize the GMM model with 3 components and diagonal covariance type
gmm = GaussianMixture(n_components=3, covariance_type='diag', random_state=42)

# Fit the GMM model to the data
gmm.fit(x2)

# Estimate the labels of the samples
predicted_labels = gmm.predict(x2)

# Compute the clustering Rand score
# Note: true_labels should be the ground truth labels of your dataset
rand_score_value = rand_score(y2, predicted_labels)

print(f'Rand Score: {rand_score_value}')
```

```
Rand Score: 0.7615936423252195
```

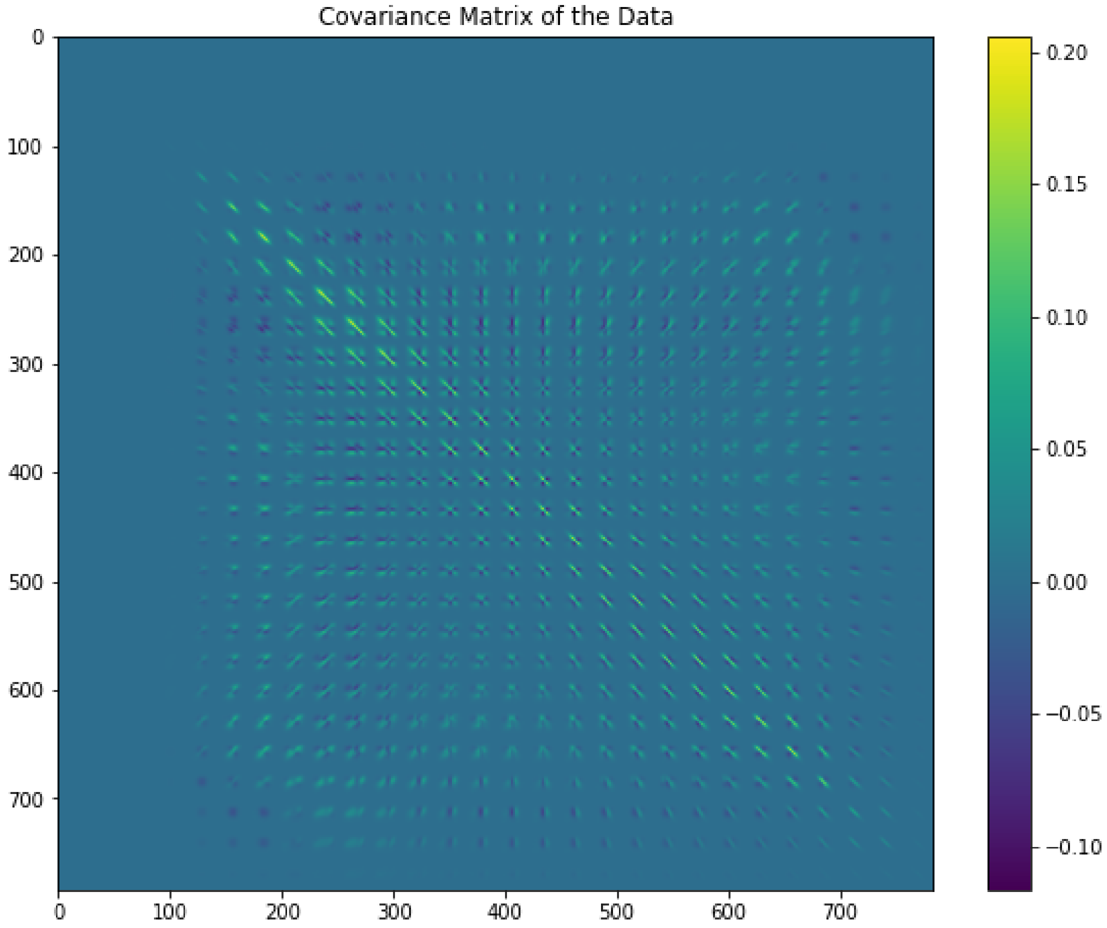
The Rand score is a measure of similarity between two data clusterings. It quantifies how well the predicted clusters match the true clusters of the data.

A Rand score of 0.7616 (approximately 0.76) indicates a relatively high level of agreement between the predicted clustering and the true labels of our data. However, K-means clustering gives a better rand score than the Gaussian Mixture Model (GMM) with a score of 0.88.

1.3.2 Dimensionality Reduction

[116]:

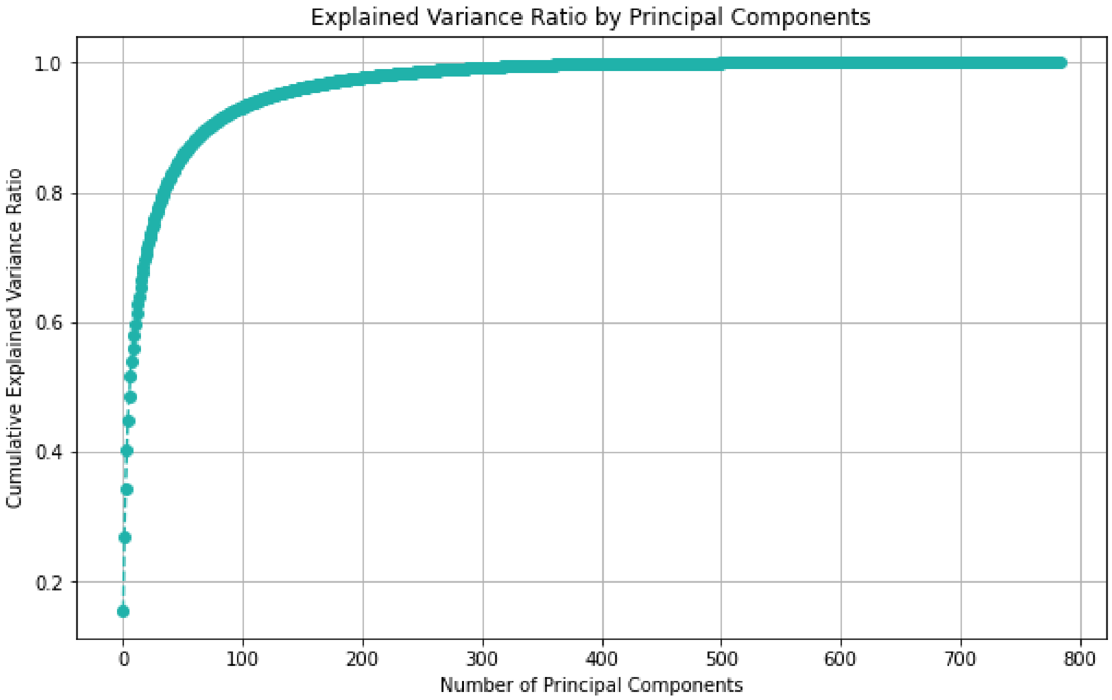
```
# Compute the covariance matrix of the data
cov_matrix = np.cov(x2.T)

# Plot the covariance matrix as an image
plt.figure(figsize=(10, 8))
plt.imshow(cov_matrix, cmap='viridis', aspect='auto')
plt.colorbar()
plt.title('Covariance Matrix of the Data')
plt.show()
```



High covariance values are predominantly concentrated in the center of the covariance matrix, especially along the diagonal. This suggests that pixels in the central regions of the images exhibit significant variability within the dataset. In contrast, the low covariance values near the matrix edges indicate that pixels in the peripheral regions remain relatively stable, as they mostly correspond to the background, which shows minimal variation.

PCA

[119]:
```
# Initialize PCA with n_components=None to keep all dimensions
pca = PCA(n_components=None)

# Fit PCA on the data
pca.fit(x2)

# Recover the explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_

# Plot the explained variance ratio
plt.figure(figsize=(10, 6))
plt.plot(np.cumsum(explained_variance_ratio), marker='o', linestyle='--',␣
 ↪color='lightseagreen')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance Ratio')
plt.title('Explained Variance Ratio by Principal Components')
plt.grid(True)
plt.show()

# Discuss the quantity of information preserved when projecting on p = 2␣
 ↪dimensions
explained_variance_p2 = np.sum(explained_variance_ratio[:2])
print(f"Explained Variance with 2 Principal Components: {explained_variance_p2:.
 ↪2f}")
```

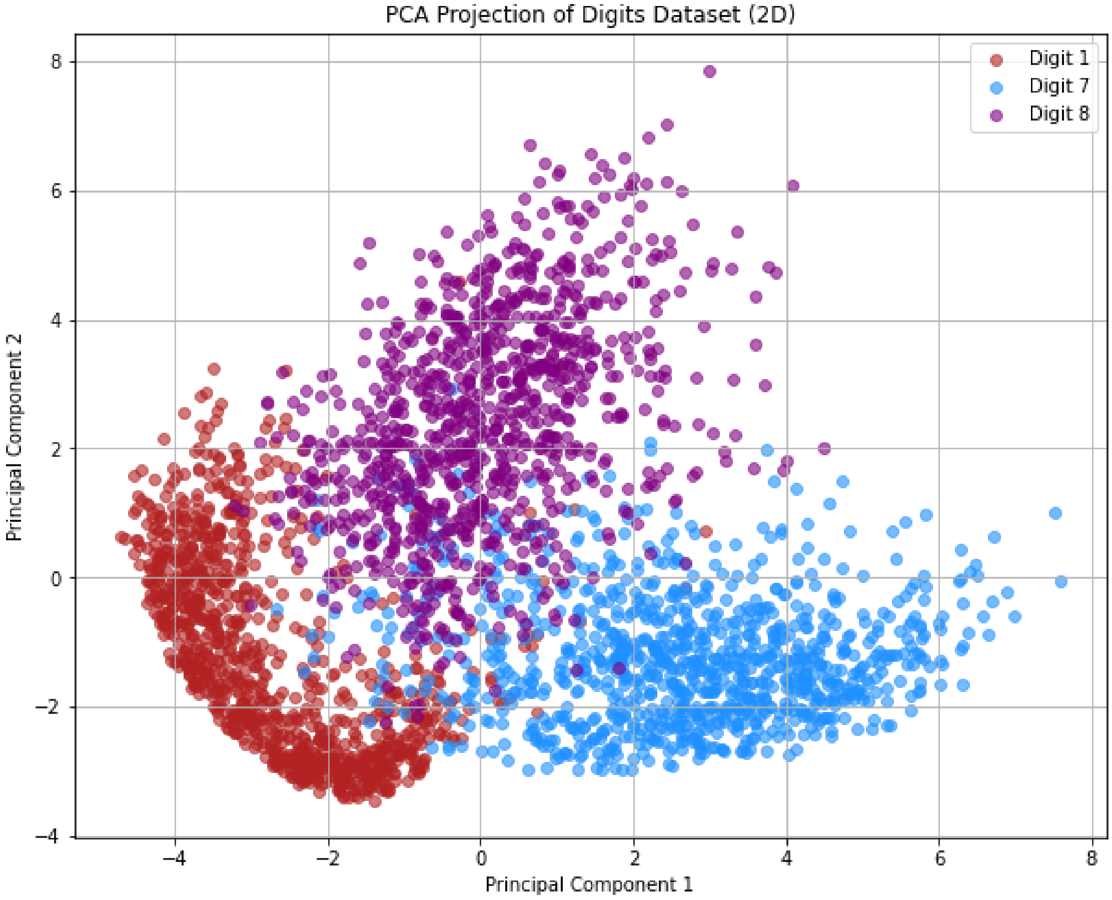


```
Explained Variance with 2 Principal Components: 0.27
```

The explained Variance with 2 Principal Components is 0.27, which means that the two components explain 27% of the variance in the data. This indicates that the two principal components capture a significant portion of the data's variability but are not sufficient to represent the entire dataset. And it would be better to use more components to capture more variance such as 100 components.

[120]:

```
# Initialize PCA with n_components=2 to reduce the data to 2 dimensions
pca_2d = PCA(n_components=2)

# Fit PCA on the digits data and transform the data to 2 dimensions
x2_pca = pca_2d.fit_transform(x2)

# Plot the projected samples in 2D space, colored by their true class labels
plt.figure(figsize=(10, 8))

# Define a color map for the classes
colors = ['firebrick', 'dodgerblue', 'purple']

# Scatter plot of the projected samples
for i, digit in enumerate(np.unique(y2)):
    plt.scatter(x2_pca[y2 == digit, 0], x2_pca[y2 == digit, 1],
                color=colors[i], label=f'Digit {digit}', alpha=0.6)

plt.title('PCA Projection of Digits Dataset (2D)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.grid(True)
plt.show()
```



Information Preserved:

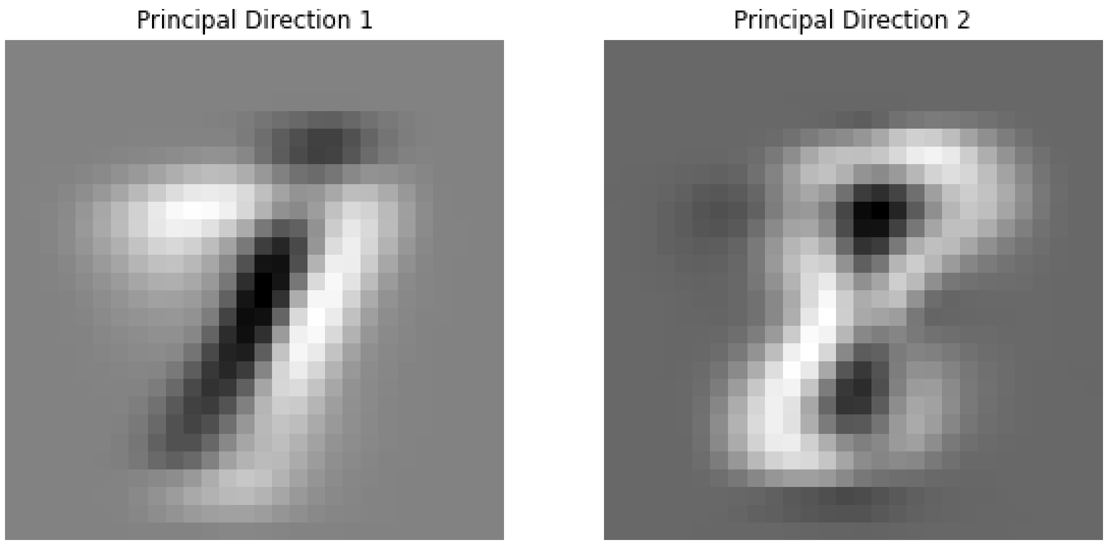
1. **Variance in the Data:** PCA attempts to capture the directions (principal components) in which the data varies the most. The first principal component explains the largest amount of variance, while the second principal component captures the second-largest variance, orthogonal to the first.
   - In this 2D projection, much of the variance in the dataset is preserved, but naturally, some information is lost due to reducing high-dimensional data to just two dimensions.
2. **Structure and Patterns:** Although the dimensionality has been reduced, the relative positioning of the clusters in the plot suggests that some structure has been preserved. In particular:
   - **Clusters**: The three distinct clusters in the plot likely represent the different digit classes (Digits 1, 7, and 8 as per the legend). Even though this is a 2D projection, the

separation indicates that PCA was able to distinguish between the different digit shapes and patterns effectively.

- **Separation of Classes**: The clusters suggest that the different digits still maintain a separable structure in this lower-dimensional space, reflecting their differences in the original high-dimensional space.

Correlation to True Class: True Class Labels (Digit Categories): - The plot shows a clear separation between different digits (1, 7, 8), with each class forming distinct clusters. This suggests that the PCA components are indeed correlated to the true digit classes, despite the dimensionality reduction. - The fact that digit 7 (blue) is somewhat mixed with digit 8 (purple) might reflect similarities in their shapes (e.g., both having similar upper strokes). However, digit 1 (red) is well-separated, which makes sense given its distinct vertical shape. - The **relative position** of the clusters may reflect certain underlying geometric or shape similarities between the digits. For instance, digits with similar overall shapes or strokes may appear closer together. - The **spread** of each cluster could also be influenced by the variability within each digit's class. Digits that are written with more variability in form (e.g., different handwriting styles for the same digit) may have wider or less compact clusters.

[121]:
```python
# Get the first two principal directions (components)
first_two_components = pca.components_[:2]

# Plot the digits corresponding to the first two principal directions
plt.figure(figsize=(10, 5))

for i, component in enumerate(first_two_components):
    plt.subplot(1, 2, i + 1)
    plt.imshow(component.reshape(28, 28), cmap='gray')
    plt.axis('off')
    plt.title(f'Principal Direction {i + 1}')

plt.show()
```

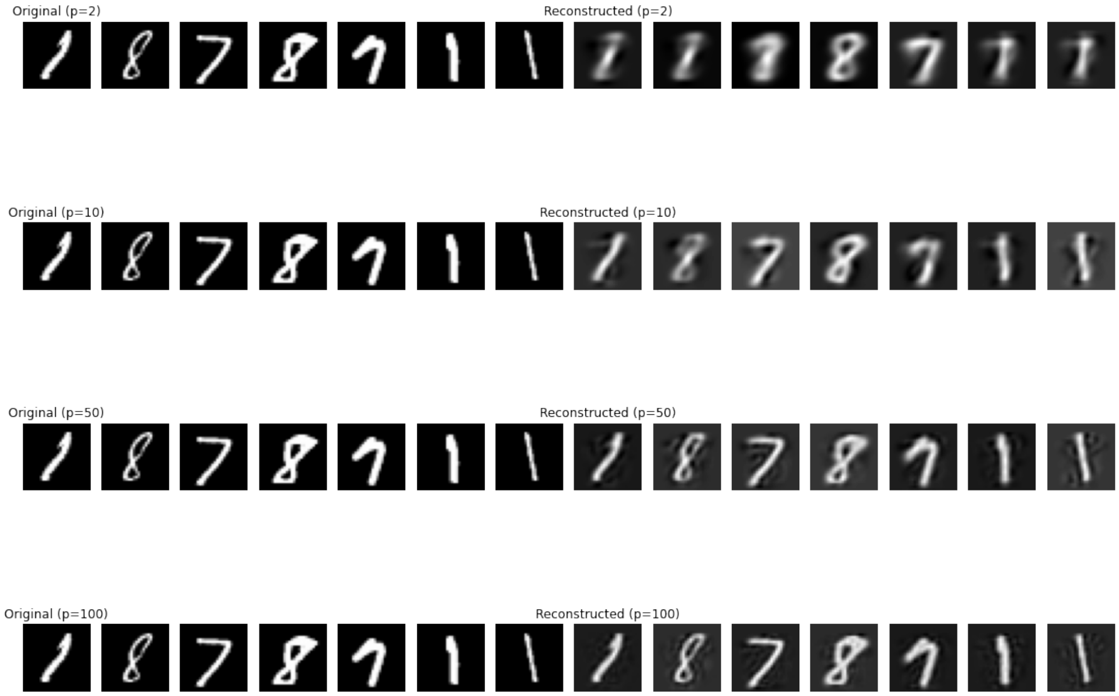


1. **First Image: Resembles the Shape of a “1” or “7”**
   - This image corresponds to the **first principal component**, which captures the **first-largest variation** in the dataset.
   - Since it resembles the shape of a “1” or “7”, this indicates that the first principal direction captures the variation in digits with simpler, more linear shapes (such as “1”, “7”, or similar).
   - **Interpretation**:
     - The **second principal component** might focus on **straight-line features** and how they differ between digits. For example, “1” and “7” are distinguished by their simplicity, and the second component could be capturing the **presence or absence of diagonal or horizontal strokes**, which distinguish these digits from others.
     - Moving along this direction could impact the **thickness or curvature of the straight lines**, emphasizing or diminishing characteristics like the top stroke of a “7” or the straightness of a “1”.

2. **Second Image: Resembles the Shape of a “8”**
   - Since it resembles the shape of an “8”, this suggests that **many samples in the dataset share common features with the digit “8”** or similar shapes. The digit “8” has a relatively complex structure with two loops (top and bottom), which contributes to significant variability in the dataset.

[125]:

```python
# Define the subspace dimensions to test
subspace_dimensions = [2, 10, 50, 100]

# Select a few samples to plot
sample_indices = np.random.choice(len(x2), 7, replace=False)
original_samples = x2[sample_indices]

# Plot the original and reconstructed samples
plt.figure(figsize=(15, len(subspace_dimensions) * 3))

for i, p in enumerate(subspace_dimensions):
    # Initialize PCA with n_components=p
    pca = PCA(n_components=p)

    # Fit PCA on the data and transform the data to p dimensions
    x2_pca = pca.fit_transform(x2)

    # Reconstruct the data from the projected samples
    x2_reconstructed = pca.inverse_transform(x2_pca)

    # Get the reconstructed samples
    reconstructed_samples = x2_reconstructed[sample_indices]
    for j, (original, reconstructed) in enumerate(zip(original_samples, reconstructed_samples)):
        # Plot the original sample
        plt.subplot(len(subspace_dimensions), 2 * len(sample_indices), i * 2 * len(sample_indices) + j + 1)
        plt.imshow(original.reshape(28, 28), cmap='gray')
        plt.axis('off')
        if j == 0:
            plt.title(f'Original (p={p})')

        # Plot the reconstructed sample
        plt.subplot(len(subspace_dimensions), 2 * len(sample_indices), i * 2 * len(sample_indices) + len(sample_indices) + j + 1)
        plt.imshow(reconstructed.reshape(28, 28), cmap='gray')
        plt.axis('off')
        if j == 0:
            plt.title(f'Reconstructed (p={p})')

plt.tight_layout()
plt.show()
```



It's clear that when p=100, the reconstruction error is very low which means that the model is able to reconstruct the data with a very high accuracy but we can still can recognize the digits in the reconstructed images for p=10.

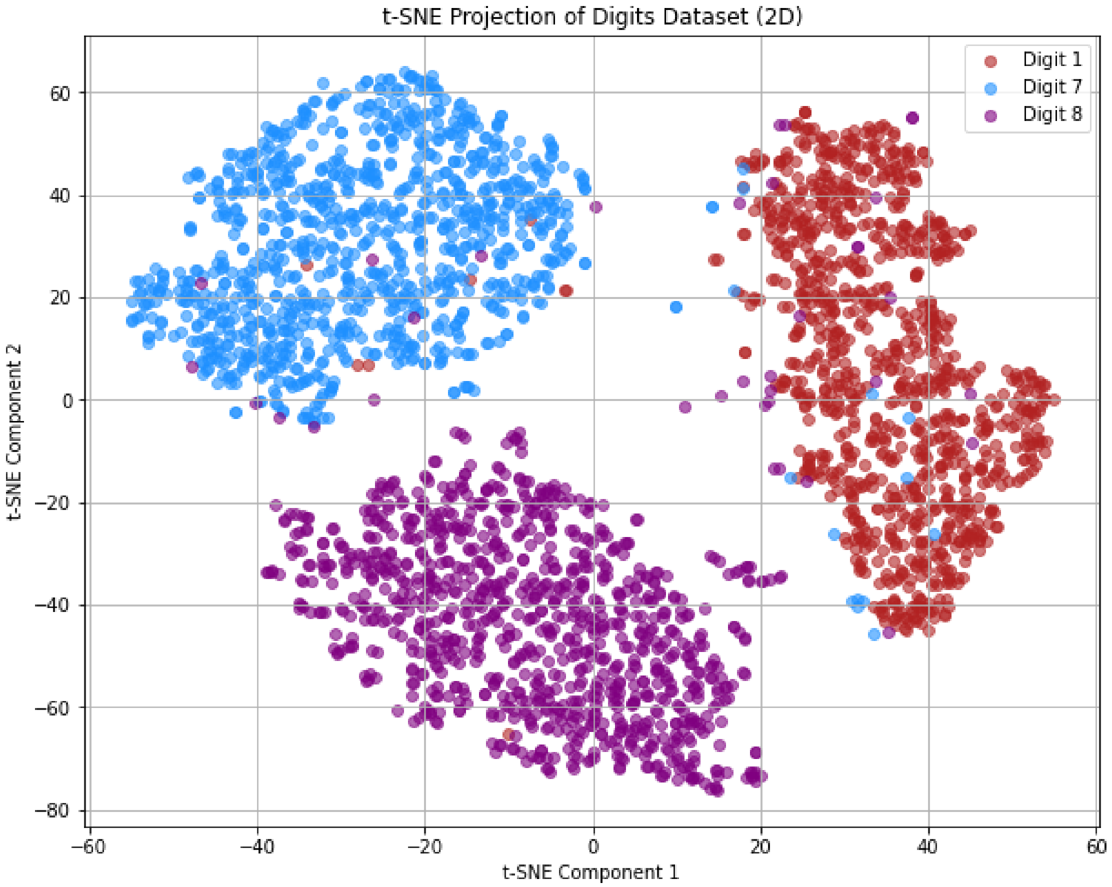
The impact of p on the quality of reconstruction is evident in the images. As p increases, the quality of the reconstructed images improves, with more details and clearer shapes visible in the digits. This is because higher values of p allow the model to capture more information and variability in the data, resulting in more accurate reconstructions.

1.3.3 Manifold Learning : TSNE

[127]:

```
# Initialize t-SNE with n_components=2 to reduce the data to 2 dimensions
tsne = TSNE(n_components=2, random_state=42) # perplexity=30 by default

# Fit t-SNE on the digits data and transform the data to 2 dimensions
x2_tsne = tsne.fit_transform(x2)

# Plot the projected samples in 2D space, colored by their true class labels
plt.figure(figsize=(10, 8))

# Define a color map for the classes
colors = ['firebrick', 'dodgerblue', 'purple']

# Scatter plot of the projected samples
for i, digit in enumerate(np.unique(y2)):
    plt.scatter(x2_tsne[y2 == digit, 0], x2_tsne[y2 == digit, 1],
                color=colors[i], label=f'Digit {digit}', alpha=0.6)

plt.title('t-SNE Projection of Digits Dataset (2D)')
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.legend()
plt.grid(True)
plt.show()
```

```
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:795: FutureWarning: The default
initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:805: FutureWarning: The default learning
rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
```



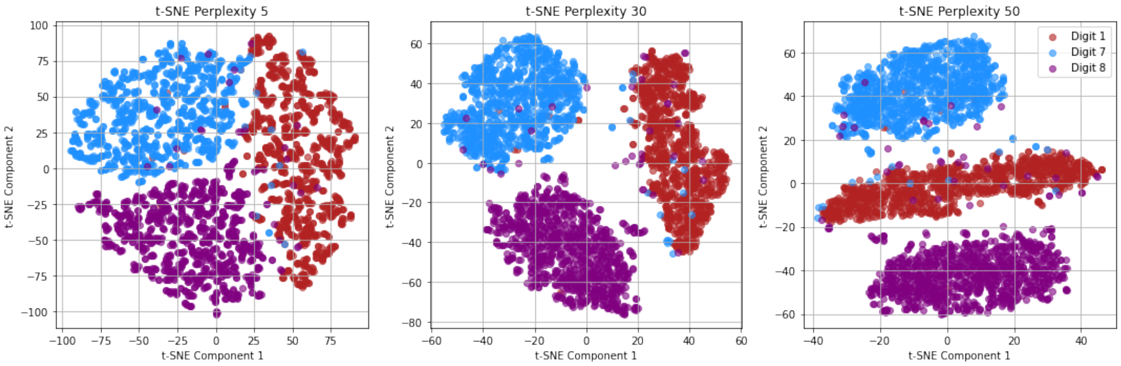
Waw ! The t-SNE algorithm has successfully separated the digits into distinct clusters

Let's try with another perplexity value to see if we can get better results.

[128]:

```
# Define the perplexity values to explore
perplexities = [5, 30, 50]

# Define a color map for the classes
colors = ['firebrick', 'dodgerblue', 'purple']

# Create a figure with subplots, one for each perplexity
plt.figure(figsize=(15, 5))

# Loop over each perplexity value and plot the t-SNE projection in a subplot
for idx, perplexity in enumerate(perplexities):
    # Initialize t-SNE with the current perplexity value
    tsne = TSNE(n_components=2, random_state=42, perplexity=perplexity)

    # Fit t-SNE on the digits data and transform the data to 2 dimensions
    x2_tsne = tsne.fit_transform(x2)
    # Create a subplot for each perplexity value
    plt.subplot(1, len(perplexities), idx + 1)

    # Scatter plot of the projected samples
    for i, digit in enumerate(np.unique(y2)):
        plt.scatter(x2_tsne[y2 == digit, 0], x2_tsne[y2 == digit, 1],
                    color=colors[i], label=f'Digit {digit}', alpha=0.6)

    # Add title and labels
    plt.title(f't-SNE Perplexity {perplexity}')
    plt.xlabel('t-SNE Component 1')
    plt.ylabel('t-SNE Component 2')
    plt.grid(True)

# Adjust layout and show the figure
plt.tight_layout()
plt.legend(loc='best', bbox_to_anchor=(1, 1))  # Legend outside the plot
plt.show()
```

```
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:795: FutureWarning: The default
initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:805: FutureWarning: The default learning
rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:795: FutureWarning: The default
initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:805: FutureWarning: The default learning
rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:795: FutureWarning: The default
initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
c:\Users\moham\AppData\Local\Programs\Python\Python310\lib\site-
packages\sklearn\manifold\_t_sne.py:805: FutureWarning: The default learning
rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
```



Effect of Perplexity in t-SNE:

- **Low perplexity (e.g., 5)**: Focuses on local structures, emphasizing small clusters but potentially losing global relationships.
- **High perplexity (e.g., 50)**: Balances local and global structure, revealing larger patterns but sometimes merging small clusters.

Is t-SNE more discriminant than PCA in 2D?

- **t-SNE**:
  - Better at separating clusters in 2D due to capturing **non-linear relationships**.
  - More effective for distinguishing between different digits, especially with the right perplexity setting.
- **PCA**:
  - Focuses on **linear variance** and gives a broader view but typically offers **less clear separation** of clusters compared to t-SNE.

1.4 Conclusion :

1.4.1 Challenges Faced

Throughout this project, I encountered several challenges. Understanding and implementing clustering algorithms, particularly K-means and Gaussian Mixture Models (GMM), initially posed difficulties. The nuances of selecting appropriate hyperparameters, such as the number of clusters (K) and perplexity, required considerable experimentation and reflection. Additionally, the dimensionality reduction techniques, including PCA and t-SNE, presented their own learning curves. Visualizing the results and interpreting them in the context of the data was sometimes overwhelming, especially when higher dimensions obscured meaningful insights.

1.4.2 Future Improvements

If I were to approach this project again, a more thorough review of the underlying mathematical principles behind these algorithms would enhance my understanding and application of these techniques. I also aim to focus on integrating bonus questions more systematically, as my initial attempts yielded suboptimal results.

1.4.3 New Learnings

This session introduced me to valuable tools and libraries, such as Scikit-learn, which streamline the implementation of complex algorithms.

1.4.4 Relation to Course

The practical application of theoretical concepts from our course solidified my understanding of machine learning techniques. The experience of interpreting clustering results in relation to geographical locations or class labels emphasized the importance of contextual knowledge when analyzing data.

1.4.5 Professional Application

In a professional setting, the skills I developed in this session are crucial for data analysis and machine learning projects. The ability to preprocess data, apply various clustering methods, and interpret results will be invaluable in roles focused on data science and analytics. Understanding how to manipulate and visualize data effectively is essential for communicating insights to stakeholders and making informed decisions.

1.4.6 Final Thoughts

Overall, this practical session was an enriching experience that not only enhanced my technical skills but also provided a deeper appreciation for the intricacies of unsupervised machine learning techniques. I look forward to leveraging these tools in future academic and professional endeavors, continually striving to refine my understanding and application of these powerful methodologies.