Maor Blumberg 204924278 and Eylon Shahar 207943747

`Final Project`

Imports

In [3]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib as plt
import os
```

Loading matrices list from my folder

120 matrices for 15 pairs from same household, each subject's facial muscles get recorded using openface during 4 different manipulations

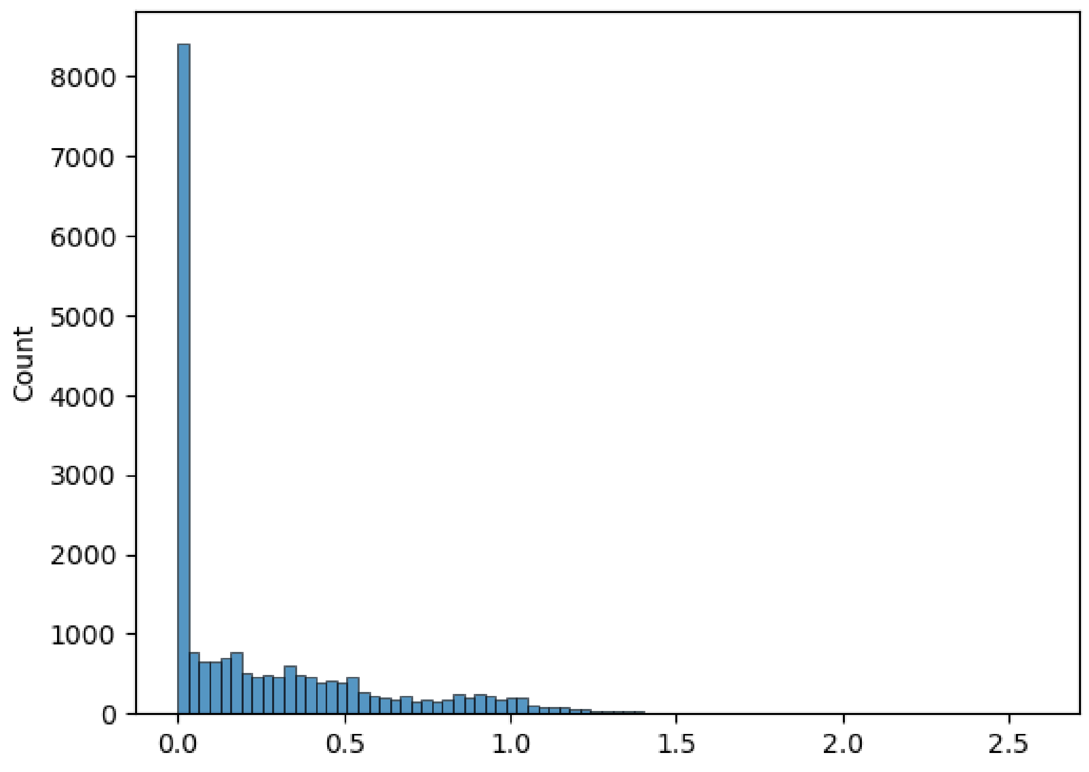
In [4]:
```
# Define the base directory
base_dir = r"C:\Users\maorb\Desktop\Work\CSV_OpenFace-folder\CSV_OpenFace - Main"

# Our columns of interest
columns = [
    'AU02_r', 'AU04_r', 'AU05_r', 'AU06_r', 'AU07_r',
    'AU09_r', 'AU10_r', 'AU12_r', 'AU14_r', 'AU15_r', 'AU17_r',
    'AU20_r', 'AU23_r', 'AU25_r', 'AU26_r', 'AU45_r'
]

# Function to read files and process them
def read_and_process_file(file_path, columns):
    if file_path.endswith('.csv'):
        df = pd.read_csv(file_path)
    elif file_path.endswith('.xlsx'):
        df = pd.read_excel(file_path)
    else:
        return None

    df.columns = df.columns.str.strip() # Strip whitespace from column names
    return df[columns]

# Recursively get all files that start with "Argaman"
files = []
for root, dirs, filenames in os.walk(base_dir):
    for filename in filenames:
        if filename.startswith("Argaman") and filename.endswith(".csv"):
            files.append(os.path.join(root, filename))

# Read and process all files
dataframes = [read_and_process_file(file, columns) for file in files]

# Drop any None values in case some files were not processed
dataframes1_pre = [df for df in dataframes if df is not None]

# Find the minimum length of all dataframes
min_length = min(len(df) for df in dataframes)

# Trim all dataframes to the minimum length
dataframes1 = [df.iloc[:min_length, :] for df in dataframes]
```

In [5]:
```
dataframes1_trans = [df.T for df in dataframes1]
```

Matrix Values for example:

In [6]:
```
sns.histplot(dataframes1_trans[0].values.flatten())
```

Out[6]:
```
<Axes: ylabel='Count'>
```



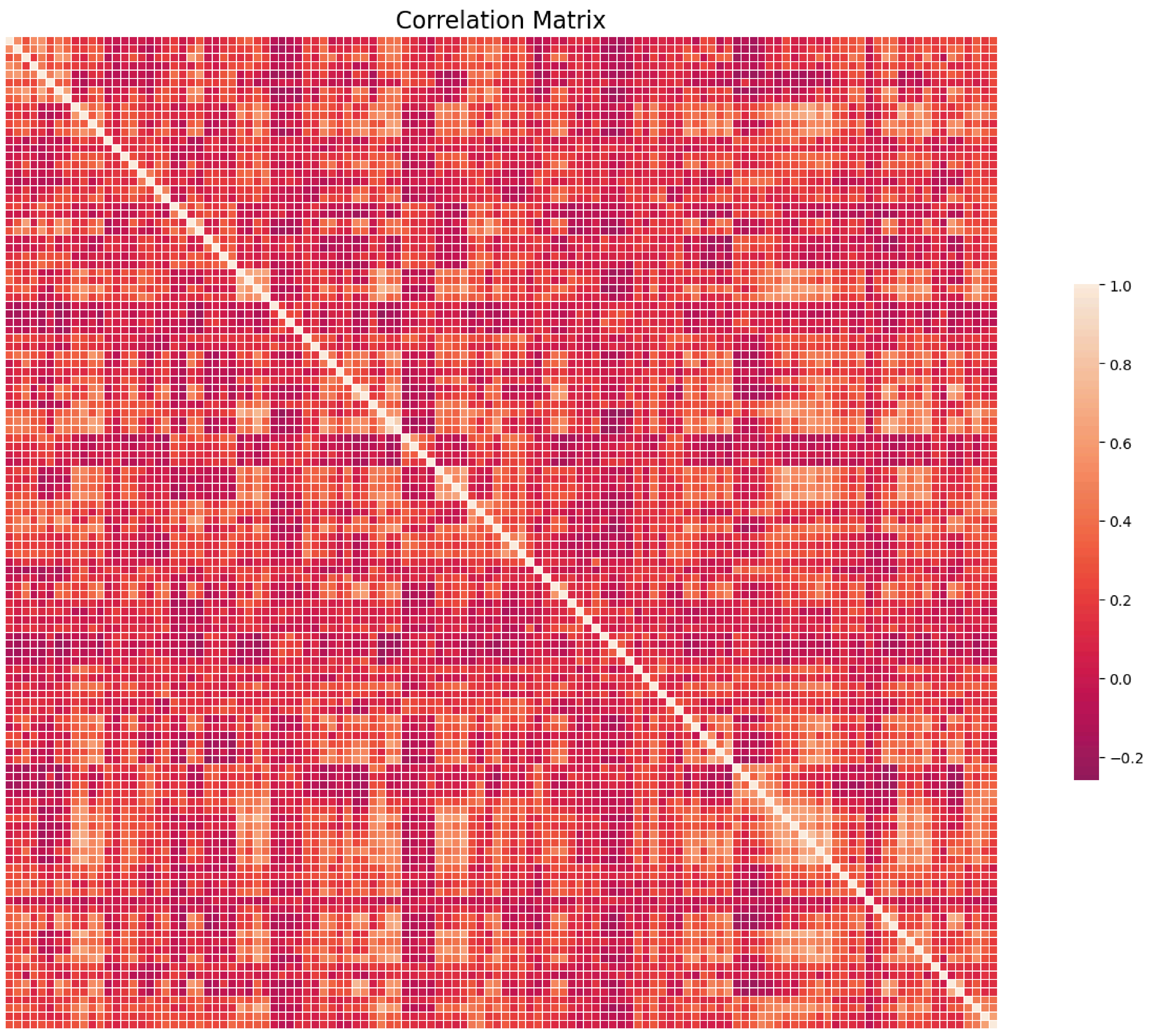
These matrices are very sparse

We'll create a matrix of all the flattened matrices list, in order to perform correlation matrix, PCA and Kmeans

In [9]:
```python
data_frames_flattened = [df.values.flatten() for df in dataframes1_trans]
```

Correlation Matrix:

In [10]:
```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# ----- Step 1: Data Preparation -----
# Assuming data_frames_flattened is already defined and is a list of NumPy arrays or can be converted to one
X = np.array(data_frames_flattened)  # Shape: (n_samples, n_features)

# ----- Step 2: Compute Correlation Matrix -----
# Set rowvar=False since each column represents a feature
corr_matrix = np.corrcoef(X)

# Initialize the matplotlib figure with the desired size
plt.figure(figsize=(15, 10))

# Create the heatmap
sns.heatmap(
    corr_matrix,
    cmap='rocket',
    center=0,
    square=True,
    xticklabels=False,
    yticklabels=False,
    linewidths=0.5,  # Optional: Adds lines between squares for better readability
    cbar_kws={"shrink": 0.5},  # Optional: Adjusts the size of the color bar,
    # title Correlation coefficience
)

# Add title to the heatmap
plt.title("Correlation Matrix", fontsize=16)

# Adjust layout to make room for the title and ensure everything fits well
plt.tight_layout()

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



We generally expect to have higher correlation around the main diagonal (Mainly, every 8 indices belong to the same pair, starting from index 0).

It seems that we don't get what we would expect. Let's further investigate.

Classic K-means:

First, we projected the data into principal components (Manual implementation took too long to run because we have a lot of sparse matrices, we settled on the console function).

Additionally, we've added the group number of each dot to the PCA clustering plots, i.e. the number of the pair's samples.

We chose to implement Offline and Online methods.

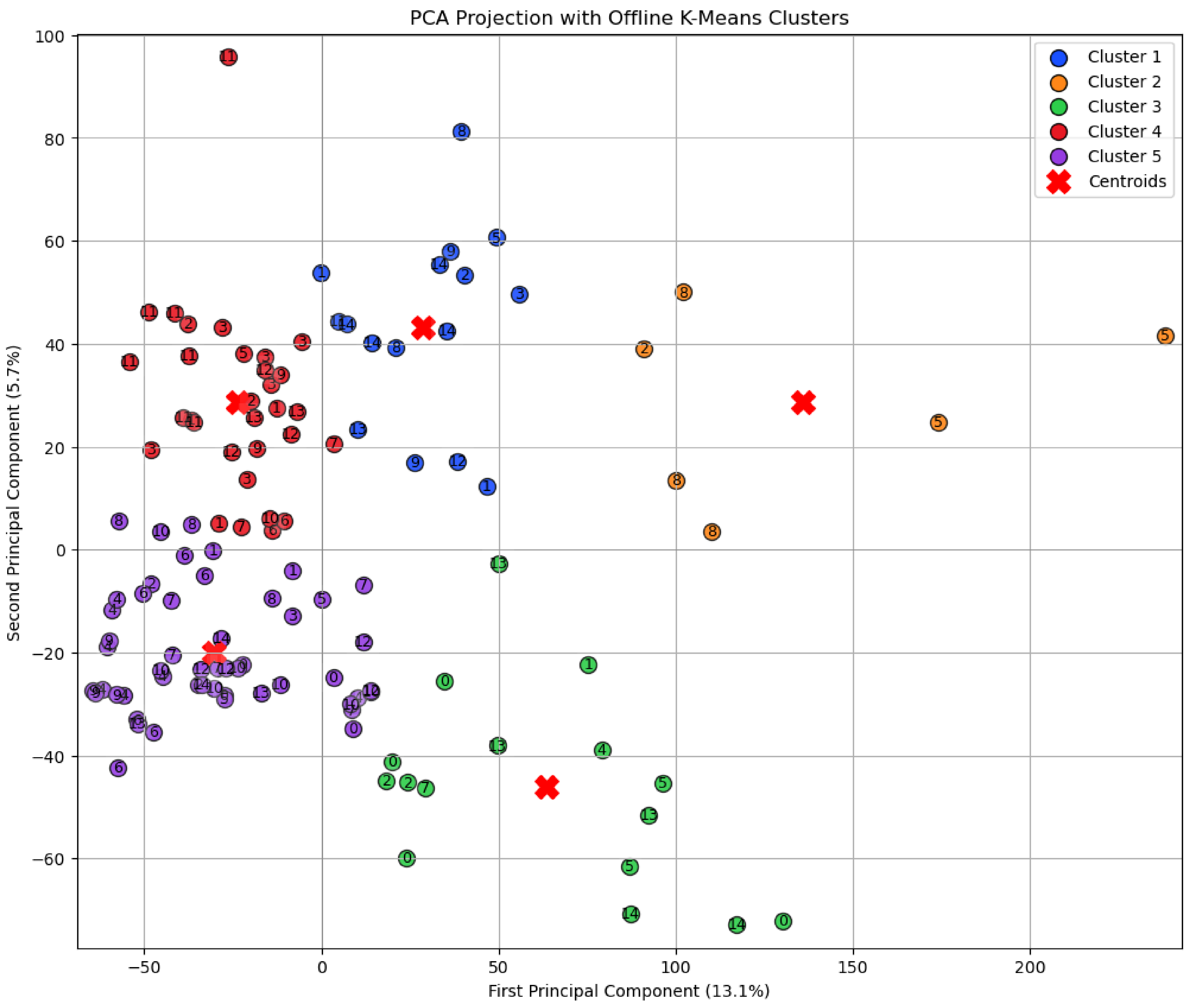
In our clustering algorithms we chose K = 5 to balance between goodness of clasiffication (inertia) and relevance of centroid, i.e. enough samples linked to each cluster

Offline:

In [28]:
```
import numpy as np
import pandas as pd  # Assuming you might need it for handling DataFrames
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler


X = np.array(data_frames_flattened)


# ----- Step 2: Standardize the Data -----
# Standardization ensures that each feature contributes equally to the PCA and K-Means
```

```
scaler = StandardScaler()
X_standardized = scaler.fit_transform(X)

# ----- Step 3: Perform PCA -----
# Initialize PCA to extract the top 2 principal components
pca = PCA(n_components=2, svd_solver='randomized', random_state=42)

# Fit PCA on the standardized data and transform
principal_components = pca.fit_transform(X_standardized)

# Extract the principal components for plotting
principal_component_1 = principal_components[:, 0]
principal_component_2 = principal_components[:, 1]

# Calculate explained variance ratios (optional, useful for interpretation)
explained_variance = pca.explained_variance_ratio_
print(f"Explained Variance by PC1: {explained_variance[0]*100:.2f}%")
print(f"Explained Variance by PC2: {explained_variance[1]*100:.2f}%")

# ----- Step 4: Implement Manual K-Means Clustering -----
def offline_kmeans(X, K, max_iters=100, tol=1e-10, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)

    n_samples, n_features = X.shape

    # Step 1: Initialize centroids by selecting K random samples from X
    initial_indices = np.random.choice(n_samples, K, replace=False)
    centroids = X[initial_indices] # Shape: (K, n_samples)

    for iteration in range(max_iters):
        # Step 2: Assign each data point to the nearest centroid
        distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)  # Shape: (n_samples, K), axis = 2: operation along columns in 3D array
                                                                          # In this case, the distance between each sample and each centroid
        labels = np.argmin(distances, axis=1)  # Shape: (n_samples,)

        # Step 3: Compute new centroids as the mean of assigned points
        new_centroids = np.array([X[labels == k].mean(axis=0) if np.any(labels == k) else centroids[k]
                                  for k in range(K)])

        # Step 4: Check for convergence
        centroid_shifts = np.linalg.norm(new_centroids - centroids, axis=1) *2
        if np.all(centroid_shifts <= tol):
            print(f"K-Means converged in {iteration + 1} iterations.")
            break

        centroids = new_centroids
    else:
        print(f"K-Means reached maximum iterations ({max_iters}).")

    # Calculate final inertia
    final_distances = np.linalg.norm(X - centroids[labels], axis=1)
    inertia = np.sum(final_distances ** 2)

    return centroids, labels, inertia

# ----- Step 5: Determine Optimal Number of Clusters (Optional) -----
# Here, you can implement the Elbow Method or Silhouette Analysis to choose K.
# For simplicity, we'll choose K=3 (you can modify this based on your data).

K = 5  # Number of clusters

# Perform K-Means clustering
centroids, labels, inertia = offline_kmeans(principal_components, K, max_iters=100, tol=1e-10, random_state=42)
print(f"Final inertia: {inertia:.2f}")

#centroids_pca = pca.transform(centroids)
# ----- Step 6: Plotting the PCA Scatter Plot with Cluster Assignments -----
plt.figure(figsize=(12, 10))
palette = sns.color_palette('bright', K)

# Scatter plot of the PCA components, colored by cluster labels
for k in range(K):
    plt.scatter(principal_component_1[labels == k],
                principal_component_2[labels == k],
                s=100,
                alpha=0.9,
                edgecolor='k',
                label=f'Cluster {k+1}',
                color=palette[k])

# plot the centroids
plt.scatter(centroids[:, 0], centroids[:, 1], s=200, color='red', marker='X', label='Centroids')

for i in range(len(principal_component_1)):
    plt.text(principal_component_1[i],
             principal_component_2[i],
             str(i//8),
             fontsize=9,
             ha='center',
             va='center',
             color='black',
            bbox=dict(facecolor='white', alpha=0.1, edgecolor='none'))

# ----- Step 7: Plotting the Principal Component Vectors (Loadings) -----
# Extract the loadings (principal axes in feature space)
# pca.components_ has shape (n_components, n_features)
loadings = pca.components_.T  # Shape: (n_features, n_components)

# Define a scaling factor to make the arrows visible on the plot
scaling_factor = 3  # Adjust based on your data


# ----- Step 8: Adjusting the Plot Limits with Buffer -----
buffer = scaling_factor * 1.5
plt.xlim(principal_component_1.min() - buffer, principal_component_1.max() + buffer)
plt.ylim(principal_component_2.min() - buffer, principal_component_2.max() + buffer)

# ----- Step 9: Adding Labels, Title, Grid, and Legend -----
plt.xlabel(f'First Principal Component ({explained_variance[0]*100:.1f}%)')
plt.ylabel(f'Second Principal Component ({explained_variance[1]*100:.1f}%)')
plt.title('PCA Projection with Offline K-Means Clusters')
plt.grid(True)
plt.legend()

# ----- Step 10: Adding Reference Axes Lines -----
plt.axhline(0, color='grey', linewidth=0.5)
plt.axvline(0, color='grey', linewidth=0.5)

plt.show()
```

```
Explained Variance by PC1: 13.11%
Explained Variance by PC2: 5.67%
K-Means converged in 12 iterations.
Final inertia: 106184.49
```



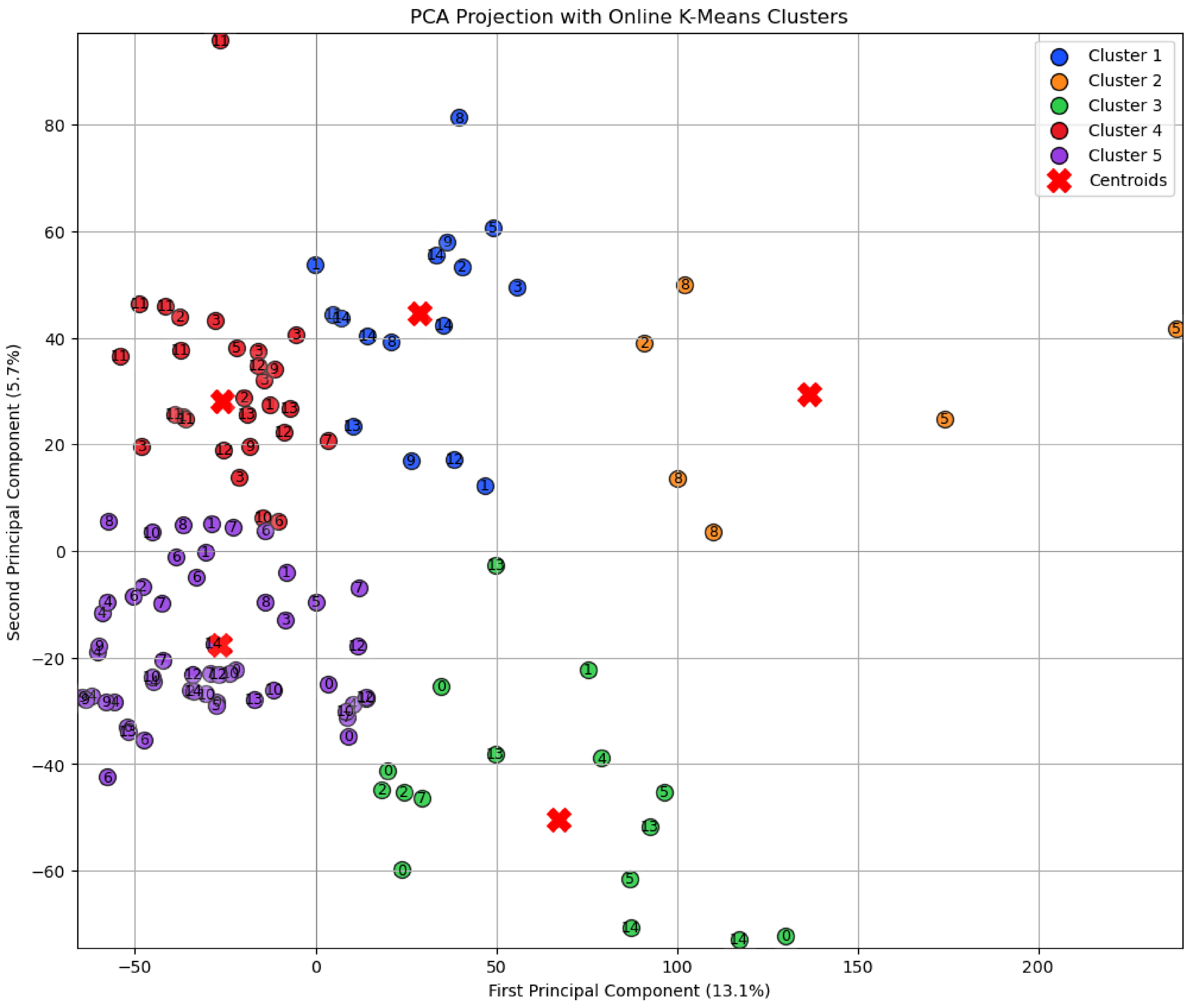
Offline:

We implemented Kmeans with given random centroids and then merely calculated each new centroid $k_i^{(new)}$ based on the mean of the matrices that were closest to $k_i^{(old)}$

Online:

In [21]:
```python
import numpy as np
import pandas as pd  # For handling DataFrames, if needed
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Assuming 'data_frames_flattened' is your data
X = np.array(data_frames_flattened)

# ----- Step 2: Standardize the Data -----
scaler = StandardScaler()
X_standardized = scaler.fit_transform(X)

# ----- Step 3: Perform PCA -----
pca = PCA(n_components=2, svd_solver='randomized', random_state=42)
principal_components = pca.fit_transform(X_standardized)
principal_component_1 = principal_components[:, 0]
principal_component_2 = principal_components[:, 1]
explained_variance = pca.explained_variance_ratio_
print(f"Explained Variance by PC1: {explained_variance[0]*100:.2f}%")
print(f"Explained Variance by PC2: {explained_variance[1]*100:.2f}%")

# ----- Step 4: Implement Online K-Means Clustering -----
def online_kmeans(X, K, max_iters=10, eta=0.1, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)

    n_samples, n_features = X.shape

    # Initialize centroids by selecting K random samples from X
    initial_indices = np.random.choice(n_samples, K, replace=False)
    centroids = X[initial_indices].copy()

    # Run for max_iters epochs over the data
    for iteration in range(max_iters):
        # Shuffle the data
        indices = np.arange(n_samples)
        np.random.shuffle(indices)
        for idx in indices:
            x = X[idx]
            # Find the nearest centroid
            distances = np.linalg.norm(centroids - x, axis=1)
            m = np.argmin(distances)
            # Update centroid m using the learning rule
            centroids[m] += eta * (x - centroids[m])


    # After training, assign labels based on the nearest centroid
    distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)
    labels = np.argmin(distances, axis=1)

    # Calculate final inertia
    final_distances = np.linalg.norm(X - centroids[labels], axis=1)
    inertia = np.sum(final_distances ** 2)

    return centroids, labels, inertia

# ----- Step 5: Determine Optimal Number of Clusters (Optional) -----
K = 5  # Number of clusters

# Perform Online K-Means clustering
centroids, labels, inertia = online_kmeans(principal_components, K, max_iters=10, eta=0.1, random_state=42)
print(f"Final inertia: {inertia:.2f}")

# ----- Step 6: Plotting the PCA Scatter Plot with Cluster Assignments -----
plt.figure(figsize=(12, 10))
palette = sns.color_palette('bright', K)

# Scatter plot of the PCA components, colored by cluster labels
for k in range(K):
    plt.scatter(principal_component_1[labels == k],
                principal_component_2[labels == k],
                s=100,
                alpha=0.9,
                edgecolor='k',
                label=f'Cluster {k+1}',
                color=palette[k])

# Plot the centroids
plt.scatter(centroids[:, 0], centroids[:, 1], s=200, color='red', marker='X', label='Centroids')

# Annotate points (optional)
for i in range(len(principal_component_1)):
    plt.text(principal_component_1[i],
             principal_component_2[i],
             str(i//8),
             fontsize=9,
             ha='center',
             va='center',
             color='black',
            bbox=dict(facecolor='white', alpha=0.1, edgecolor='none'))

# ----- Step 7: Plotting the Principal Component Vectors (Optional) -----
# If you want to plot the loadings, you can implement it here.

# ----- Step 8: Adjusting the Plot Limits with Buffer -----
buffer = 1.5
plt.xlim(principal_component_1.min() - buffer, principal_component_1.max() + buffer)
plt.ylim(principal_component_2.min() - buffer, principal_component_2.max() + buffer)

# ----- Step 9: Adding Labels, Title, Grid, and Legend -----
plt.xlabel(f'First Principal Component ({explained_variance[0]*100:.1f}%)')
plt.ylabel(f'Second Principal Component ({explained_variance[1]*100:.1f}%)')
plt.title('PCA Projection with Online K-Means Clusters')
plt.grid(True)
plt.legend()

# ----- Step 10: Adding Reference Axes Lines -----
plt.axhline(0, color='grey', linewidth=0.5)
plt.axvline(0, color='grey', linewidth=0.5)

plt.show()
```

```
Explained Variance by PC1: 13.11%
Explained Variance by PC2: 5.67%
Final inertia: 107718.42
```



Online:

We implemented Kmeans with given random centroids and then merely calculated each new centroid $K_i^{(new)}$ based on the update rule:

$$\Delta K_i = \eta(X_n - K_i^{(old)})$$

$$K_i^{(new)} = K_i^{(old)} + \Delta K_i$$

Quick detour: K-means++

K-means++ the scikit-learn implementation of K-means uses this 'newer' algorithm from 2007- https://theory.stanford.edu/~sergei/papers/kMeansPP-soda.pdf. This is an article with mind-bogglingly 11740 citations.

The main difference: instead of randomly assigning centroids in the beginning, we're 'carefully seeding' the centroids as the authors said.

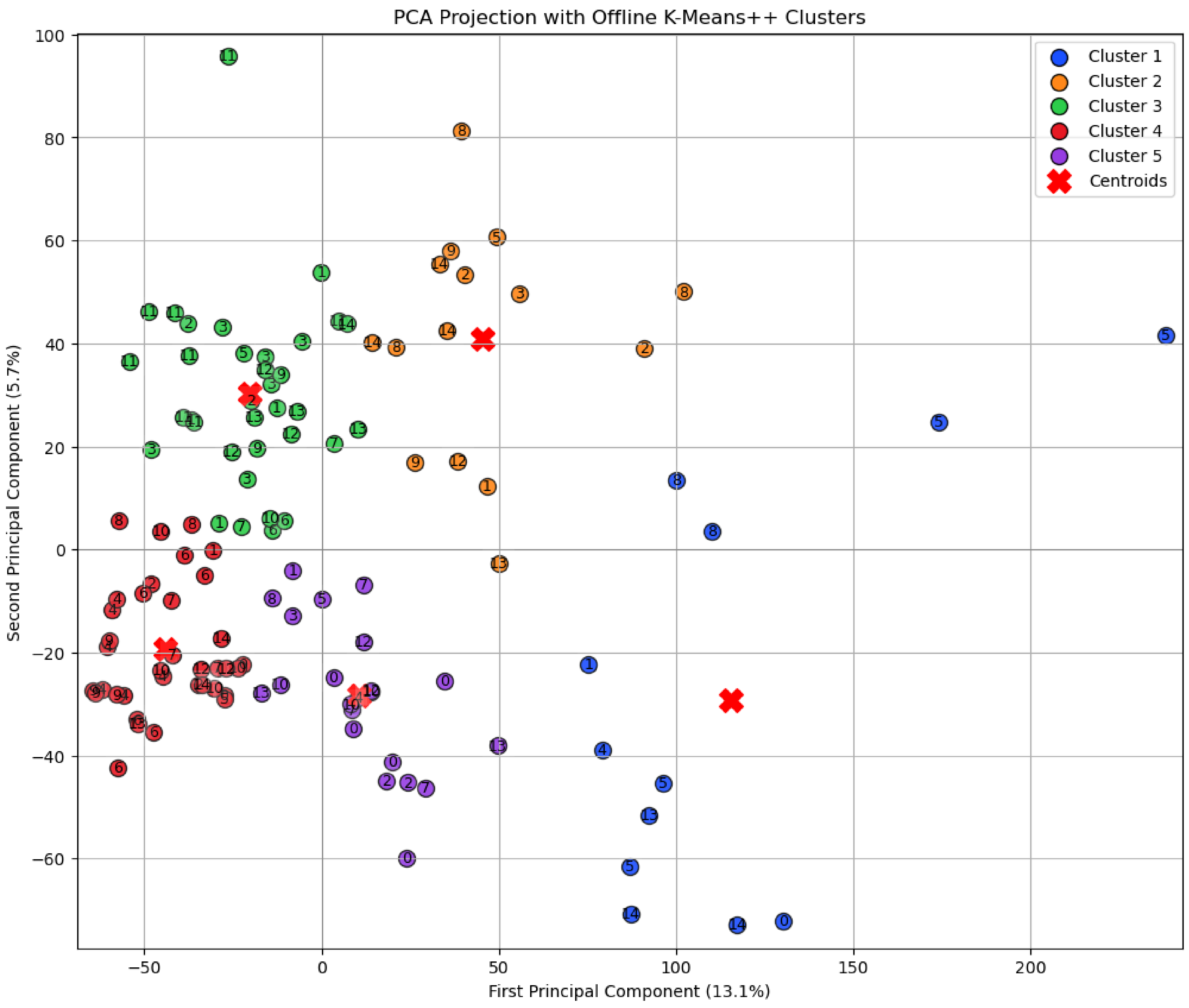
1. First Centroid:
    - Randomly placed
2. Second Centroid Selection:
    - Points farther from the first centroid have higher squared distances. These points are more likely to lie in different regions or clusters. Thus, the second centroid is likely to be placed in a distinct area, promoting cluster diversity.
3. Subsequent Centroid Selections:
    - Each new centroid is chosen based on squared distances to all existing centroids. This ensures that centroids are spread out, covering different data regions.
4. Proceed with standard K-means

Offline/online:

Here, it may be counterintuitive to look at offline and online algorithms since in the initialization stage all the samples are being used but for the sake of 'interest' let's compare the results

In [36]:
```python
import numpy as np
import pandas as pd  # Assuming you might need it for handling DataFrames
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler


X = np.array(data_frames_flattened)


# ----- Step 2: Standardize the Data -----
# Standardization ensures that each feature contributes equally to the PCA and K-Means
scaler = StandardScaler()
X_standardized = scaler.fit_transform(X)

# ----- Step 3: Perform PCA -----
# Initialize PCA to extract the top 2 principal components
pca = PCA(n_components=2, svd_solver='randomized', random_state=42)

# Fit PCA on the standardized data and transform
principal_components = pca.fit_transform(X_standardized)

# Extract the principal components for plotting
principal_component_1 = principal_components[:, 0]
principal_component_2 = principal_components[:, 1]

# Calculate explained variance ratios (optional, useful for interpretation)
explained_variance = pca.explained_variance_ratio_
print(f"Explained Variance by PC1: {explained_variance[0]*100:.2f}%")
print(f"Explained Variance by PC2: {explained_variance[1]*100:.2f}%")

# ----- Step 4: Implement K-Means Clustering with k-means++ Initialization -----

def kmeans_plus_plus_initialization(X, K, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)
    n_samples, n_features = X.shape
    centroids = np.empty((K, n_features))

    # Step 1: Choose the first centroid randomly
    first_centroid_idx = np.random.randint(n_samples)
    centroids[0] = X[first_centroid_idx]

    # Initialize an array to store the closest distances squared
    closest_dist_sq = np.full(n_samples, np.inf)

    for c in range(1, K):
        # Compute distances from each point to the nearest existing centroid
        distances = np.linalg.norm(X - centroids[c-1], axis=1)**2
        closest_dist_sq = np.minimum(closest_dist_sq, distances) # returns minimum of two arrays element-wise

        # Compute probabilities proportional to the squared distances
        probabilities = closest_dist_sq / closest_dist_sq.sum()

        # Select the next centroid
        cumulative_probabilities = np.cumsum(probabilities)
        r = np.random.rand()
        next_centroid_idx = np.searchsorted(cumulative_probabilities, r)
        centroids[c] = X[next_centroid_idx]

    return centroids

def offline_kmeans_PP(X, K, max_iters=300, tol=1e-10, random_state=None):
    # Step 1: Initialize centroids using k-means++ algorithm
    centroids = kmeans_plus_plus_initialization(X, K, random_state=random_state)

    for iteration in range(max_iters):
        # Step 2: Assign each data point to the nearest centroid
        distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)  # Shape: (n_samples, K)
        labels = np.argmin(distances, axis=1)  # Shape: (n_samples,)

        # Step 3: Compute new centroids as the mean of assigned points
        new_centroids = np.zeros_like(centroids)
        for k in range(K):
            if np.any(labels == k):
                new_centroids[k] = X[labels == k].mean(axis=0)
            else:
                # Reinitialize to the point with the highest inertia
                farthest_point_idx = np.argmax([np.min(np.linalg.norm(X - centroid, axis=1)) for centroid in centroids])
                new_centroids[k] = X[farthest_point_idx]
                print(f"Reinitialized centroid {k} to point index {farthest_point_idx}.")

        # Step 4: Check for convergence based on relative shifts
        centroid_shifts = np.linalg.norm(new_centroids - centroids, axis=1)
        relative_shifts = centroid_shifts / (np.linalg.norm(centroids, axis=1) + 1e-10)
        average_shift = np.mean(relative_shifts)

        print(f"Iteration {iteration + 1}: Average Relative Shift = {average_shift:.6f}")

        if np.all(relative_shifts <= tol):
            print(f"K-Means converged in {iteration + 1} iterations.")
            break

        centroids = new_centroids
    else:
        print(f"K-Means reached maximum iterations ({max_iters}).")

    # Calculate final inertia
    final_distances = np.linalg.norm(X - centroids[labels], axis=1)
    inertia = np.sum(final_distances ** 2)

    return centroids, labels, inertia
# ----- Step 5: Determine Optimal Number of Clusters (Optional) -----
# Here, you can implement the Elbow Method or Silhouette Analysis to choose K.
# For simplicity, we'll choose K=3 (you can modify this based on your data).

K = 5  # Number of clusters

# Perform K-Means clustering
centroids, labels_final, inertia = offline_kmeans_PP(principal_components, K, max_iters=100, tol=1e-10, random_state=42)
print(f"Final inertia: {inertia:.2f}")


#centroids_pca = pca.transform(centroids)

# ----- Step 6: Plotting the PCA Scatter Plot with Cluster Assignments -----
plt.figure(figsize=(12, 10))
palette = sns.color_palette('bright', K)

# Scatter plot of the PCA components, colored by cluster labels
for k in range(K):
    plt.scatter(principal_component_1[labels_final == k],
                principal_component_2[labels_final == k],
                s=100,
                alpha=0.9,
                edgecolor='k',
                label=f'Cluster {k+1}',
                color=palette[k])

#Plot centroids
plt.scatter(centroids[:, 0], centroids[:, 1], s=200, c='r', marker='X', label='Centroids')

for i in range(len(principal_component_1)):
    plt.text(principal_component_1[i],
             principal_component_2[i],
             str(i//8),
             fontsize=9,
             ha='center',
             va='center',
             color='black',
             bbox=dict(facecolor='white', alpha=0.1, edgecolor='none'))




# ----- Step 7: Plotting the Principal Component Vectors (Loadings) -----
# Extract the loadings (principal axes in feature space)
# pca.components_ has shape (n_components, n_features)

# Define a scaling factor to make the arrows visible on the plot
scaling_factor = 3  # Adjust based on your data

# (Optional code to plot the loadings if needed)
# for i in range(loadings.shape[0]):
#     plt.arrow(0, 0, loadings[i, 0]*scaling_factor, loadings[i, 1]*scaling_factor,
#                color='r', alpha=0.5)
#     plt.text(loadings[i, 0]*scaling_factor*1.15, loadings[i, 1]*scaling_factor*1.15,
#              f"Var{i+1}", color='g', ha='center', va='center')

# ----- Step 8: Adjusting the Plot Limits with Buffer -----
buffer = scaling_factor * 1.5
plt.xlim(principal_component_1.min() - buffer, principal_component_1.max() + buffer)
plt.ylim(principal_component_2.min() - buffer, principal_component_2.max() + buffer)

# ----- Step 9: Adding Labels, Title, Grid, and Legend -----
plt.xlabel(f'First Principal Component ({explained_variance[0]*100:.1f}%)')
plt.ylabel(f'Second Principal Component ({explained_variance[1]*100:.1f}%)')
plt.title('PCA Projection with Offline K-Means++ Clusters')
plt.grid(True)
plt.legend()

# ----- Step 10: Adding Reference Axes Lines -----
plt.axhline(0, color='grey', linewidth=0.5)
plt.axvline(0, color='grey', linewidth=0.5)

plt.show()
```

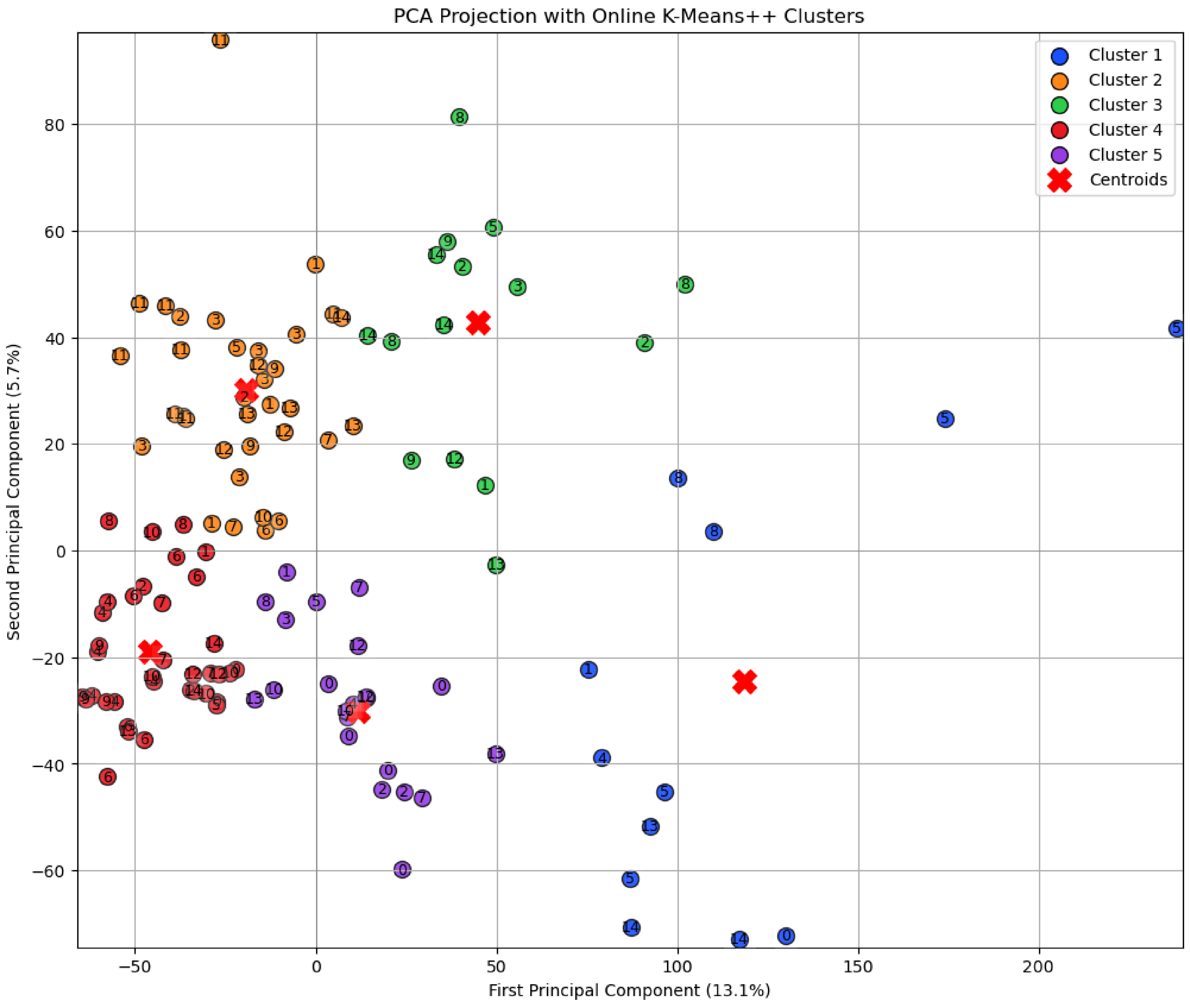
```
Explained Variance by PC1: 13.11%
Explained Variance by PC2: 5.67%
Iteration 1: Average Relative Shift = 0.453148
Iteration 2: Average Relative Shift = 0.215635
Iteration 3: Average Relative Shift = 0.208670
Iteration 4: Average Relative Shift = 0.145940
Iteration 5: Average Relative Shift = 0.117331
Iteration 6: Average Relative Shift = 0.126046
Iteration 7: Average Relative Shift = 0.053944
Iteration 8: Average Relative Shift = 0.000000
K-Means converged in 8 iterations.
Final inertia: 98093.83
```



In [31]:
```
import numpy as np
import pandas as pd  # Assuming you might need it for handling DataFrames
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
```

```python
# Assuming 'data_frames_flattened' is your data
X = np.array(data_frames_flattened)

# ----- Step 1: Standardize the Data -----
scaler = StandardScaler()
X_standardized = scaler.fit_transform(X)

# ----- Step 2: Perform PCA -----
pca = PCA(n_components=2, svd_solver='randomized', random_state=42)
principal_components = pca.fit_transform(X_standardized)
principal_component_1 = principal_components[:, 0]
principal_component_2 = principal_components[:, 1]
explained_variance = pca.explained_variance_ratio_
print(f"Explained Variance by PC1: {explained_variance[0]*100:.2f}%")
print(f"Explained Variance by PC2: {explained_variance[1]*100:.2f}%")

# ----- Step 3: Implement Online K-Means Clustering with K-Means++ Initialization -----
def kmeans_plus_plus_initialization(X, K, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)
    n_samples, n_features = X.shape
    centroids = np.empty((K, n_features))

    # Step 1: Choose the first centroid randomly
    first_centroid_idx = np.random.randint(n_samples)
    centroids[0] = X[first_centroid_idx]

    # Initialize an array to store the closest distances squared
    closest_dist_sq = np.full(n_samples, np.inf)

    for c in range(1, K):
        # Compute distances from each point to the nearest existing centroid
        distances = np.linalg.norm(X - centroids[c-1], axis=1)**2
        closest_dist_sq = np.minimum(closest_dist_sq, distances)

        # Compute probabilities proportional to the squared distances
        probabilities = closest_dist_sq / closest_dist_sq.sum()

        # Select the next centroid
        cumulative_probabilities = np.cumsum(probabilities)
        r = np.random.rand()
        next_centroid_idx = np.searchsorted(cumulative_probabilities, r)
        centroids[c] = X[next_centroid_idx]

    return centroids

def online_kmeans_PP(X, K, max_iters=10, eta=0.1, random_state=None):
    n_samples, n_features = X.shape

    # Initialize centroids using K-Means++ initialization
    centroids = kmeans_plus_plus_initialization(X, K, random_state=random_state)

    for iteration in range(max_iters):
        # Shuffle the data
        indices = np.arange(n_samples)
        np.random.shuffle(indices)
        for idx in indices:
            x = X[idx]
            # Find the nearest centroid
            distances = np.linalg.norm(centroids - x, axis=1)
            m = np.argmin(distances)
            # Update centroid m using the learning rule
            centroids[m] += eta * (x - centroids[m])


    # After training, assign labels based on the nearest centroid
    distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)
    labels = np.argmin(distances, axis=1)

    # Calculate final inertia
    final_distances = np.linalg.norm(X - centroids[labels], axis=1)
    inertia = np.sum(final_distances ** 2)

    return centroids, labels, inertia

# ----- Step 4: Determine Optimal Number of Clusters (Optional) -----
K = 5  # Number of clusters

# Perform Online K-Means clustering with K-Means++ initialization
centroids, labels, inertia = online_kmeans_PP(principal_components, K, max_iters=10, eta=0.1, random_state=42)
print(f"Final inertia: {inertia:.2f}")

# ----- Step 5: Plotting the PCA Scatter Plot with Cluster Assignments -----
plt.figure(figsize=(12, 10))
palette = sns.color_palette('bright', K)

# Scatter plot of the PCA components, colored by cluster labels
for k in range(K):
    plt.scatter(principal_component_1[labels == k],
                principal_component_2[labels == k],
                s=100,
                alpha=0.9,
                 edgecolor='k',
                 label=f'Cluster {k+1}',
                 color=palette[k])

# Plot the centroids
plt.scatter(centroids[:, 0], centroids[:, 1], s=200, c='r', marker='X', label='Centroids')

# Annotate points (optional)
for i in range(len(principal_component_1)):
    plt.text(principal_component_1[i],
             principal_component_2[i],
             str(i//8),
             fontsize=9,
             ha='center',
             va='center',
             color='black',
             bbox=dict(facecolor='white', alpha=0.1, edgecolor='none'))

# ----- Step 6: Adjusting the Plot Limits with Buffer -----
buffer = 1.5
plt.xlim(principal_component_1.min() - buffer, principal_component_1.max() + buffer)
plt.ylim(principal_component_2.min() - buffer, principal_component_2.max() + buffer)

# ----- Step 7: Adding Labels, Title, Grid, and Legend -----
plt.xlabel(f'First Principal Component ({explained_variance[0]*100:.1f}%)')
plt.ylabel(f'Second Principal Component ({explained_variance[1]*100:.1f}%)')
plt.title('PCA Projection with Online K-Means++ Clusters')
plt.grid(True)
plt.legend()

# ----- Step 8: Adding Reference Axes Lines -----
plt.axhline(0, color='grey', linewidth=0.5)
plt.axvline(0, color='grey', linewidth=0.5)

plt.show()
```

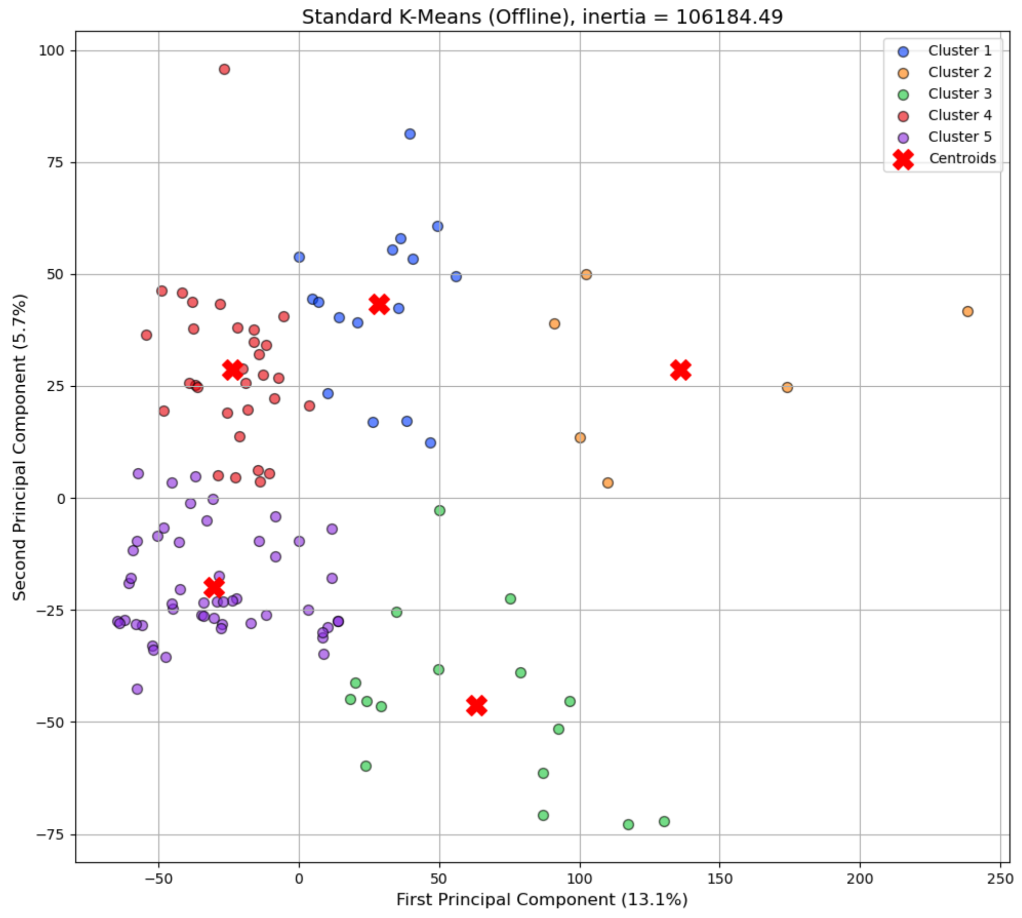
```
Explained Variance by PC1: 13.11%
Explained Variance by PC2: 5.67%
Final inertia: 98848.00
```

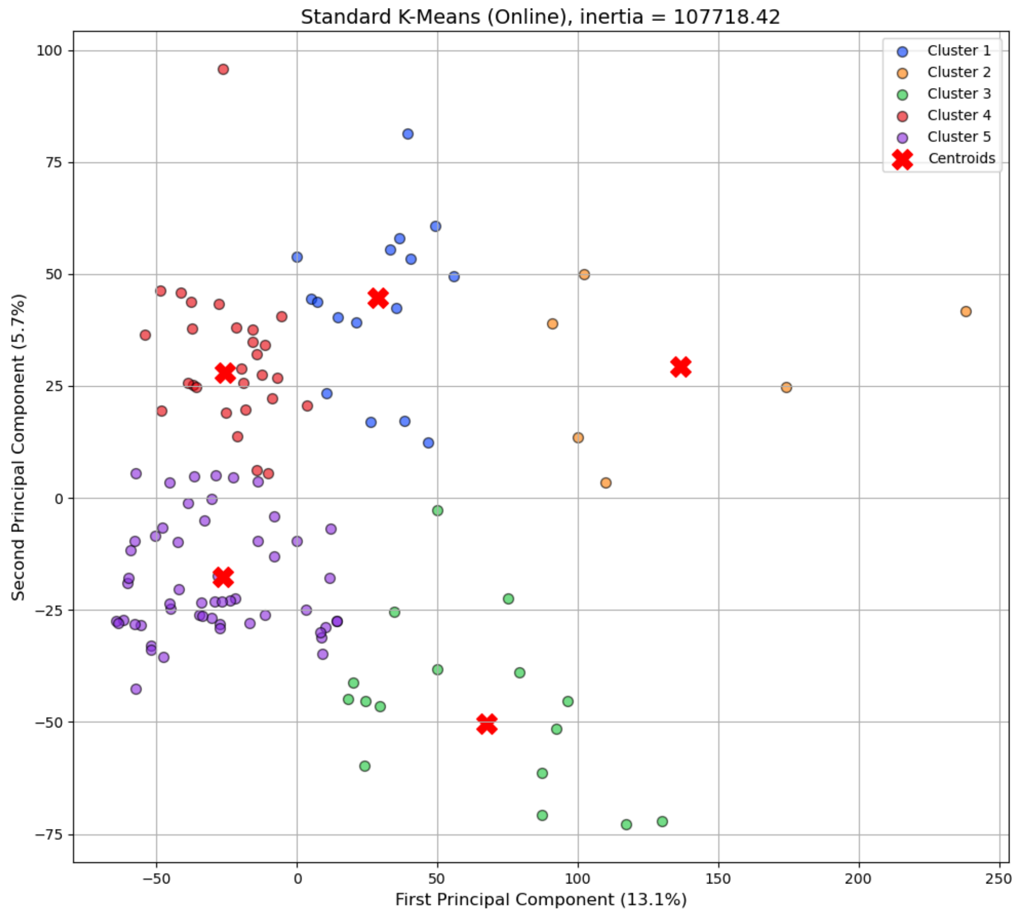


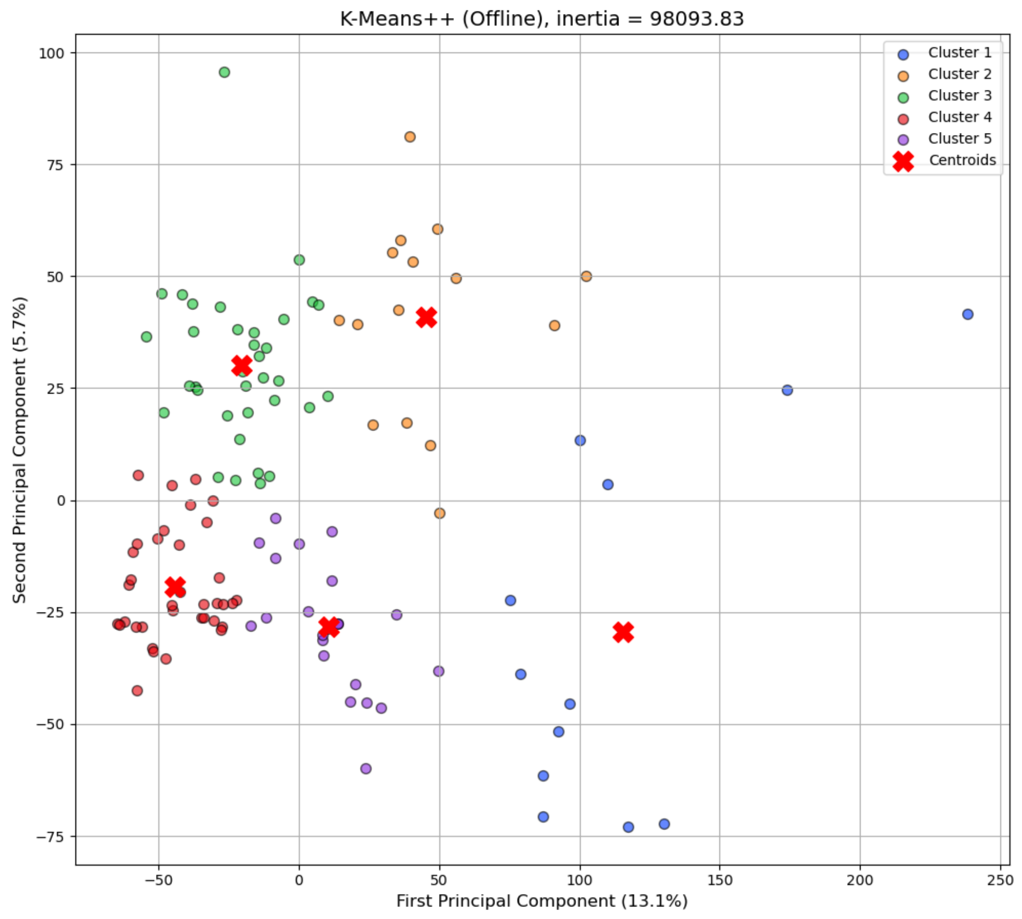
Combined plot

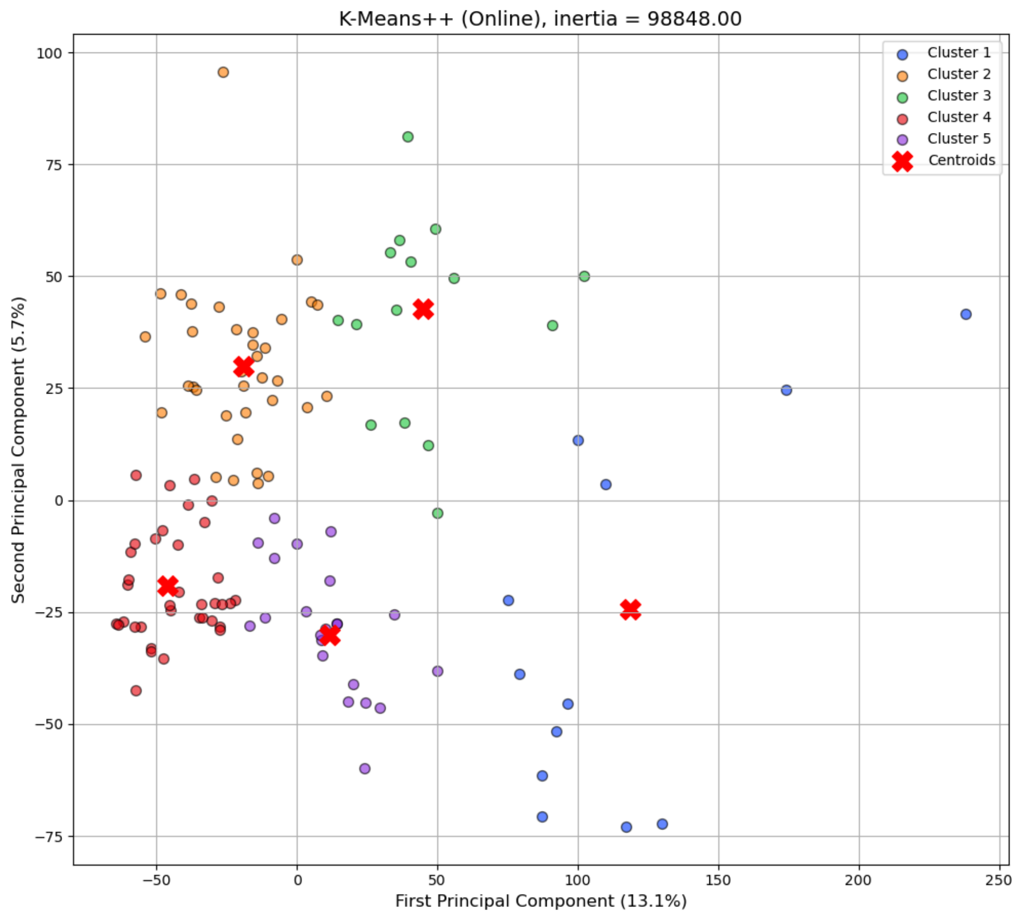
In [33]:
```python
def offline_kmeans(X, K, max_iters=100, tol=1e-4, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)

    n_samples, n_features = X.shape

    # Step 1: Initialize centroids by selecting K random samples from X
    initial_indices = np.random.choice(n_samples, K, replace=False)
    centroids = X[initial_indices].copy()

    for iteration in range(max_iters):
        # Step 2: Assign each data point to the nearest centroid
        distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)  # Shape: (n_samples, K)
        labels = np.argmin(distances, axis=1)  # Shape: (n_samples,)

        # Step 3: Compute new centroids as the mean of assigned points
        new_centroids = np.array([
            X[labels == k].mean(axis=0) if np.any(labels == k) else centroids[k]
            for k in range(K)
        ])

        # Step 4: Check for convergence
        centroid_shifts = np.linalg.norm(new_centroids - centroids, axis=1)
        if np.all(centroid_shifts <= tol):
            print(f"Offline K-Means converged in {iteration + 1} iterations.")
            break

        centroids = new_centroids
    else:
        print(f"Offline K-Means reached maximum iterations ({max_iters}).")

    # Calculate final inertia
    final_distances = np.linalg.norm(X - centroids[labels], axis=1)
    inertia = np.sum(final_distances ** 2)

    return centroids, labels, inertia

def online_kmeans(X, K, max_iters=10, eta=0.1, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)

    n_samples, n_features = X.shape

    # Step 1: Initialize centroids by selecting K random samples from X
    initial_indices = np.random.choice(n_samples, K, replace=False)
    centroids = X[initial_indices].copy()

    # Run for max_iters epochs over the data
    for iteration in range(max_iters):
        # Shuffle the data
        indices = np.arange(n_samples)
        np.random.shuffle(indices)
        for idx in indices:
            x = X[idx]
            # Find the nearest centroid
            distances = np.linalg.norm(centroids - x, axis=1)
            m = np.argmin(distances)
            # Update centroid m using the learning rule
            centroids[m] += eta * (x - centroids[m])
        print(f"Online K-Means completed iteration {iteration + 1}.")

    # After training, assign labels based on the nearest centroid
    distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)
    labels = np.argmin(distances, axis=1)

    # Calculate final inertia
    final_distances = np.linalg.norm(X - centroids[labels], axis=1)
    inertia = np.sum(final_distances ** 2)

    return centroids, labels, inertia

def kmeans_plus_plus_initialization(X, K, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)
    n_samples, n_features = X.shape
    centroids = np.empty((K, n_features))

    # Step 1: Choose the first centroid randomly
    first_centroid_idx = np.random.randint(n_samples)
    centroids[0] = X[first_centroid_idx]

    # Initialize an array to store the closest distances squared
    closest_dist_sq = np.full(n_samples, np.inf)

    for c in range(1, K):
        # Compute squared distances from each point to the nearest existing centroid
        distances = np.linalg.norm(X - centroids[c-1], axis=1)**2
        closest_dist_sq = np.minimum(closest_dist_sq, distances)

        # Compute probabilities proportional to the squared distances
        probabilities = closest_dist_sq / closest_dist_sq.sum()
        # Select the next centroid
        cumulative_probabilities = np.cumsum(probabilities)
        r = np.random.rand()
        next_centroid_idx = np.searchsorted(cumulative_probabilities, r)
        centroids[c] = X[next_centroid_idx]

    return centroids

def offline_kmeans_PP(X, K, max_iters=100, tol=1e-4, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)

    # Step 1: Initialize centroids using K-Means++ algorithm
    centroids = kmeans_plus_plus_initialization(X, K, random_state=random_state)

    for iteration in range(max_iters):
        # Step 2: Assign each data point to the nearest centroid
        distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)  # Shape: (n_samples, K)
        labels = np.argmin(distances, axis=1)  # Shape: (n_samples,)

        # Step 3: Compute new centroids as the mean of assigned points
        new_centroids = np.array([
            X[labels == k].mean(axis=0) if np.any(labels == k) else centroids[k]
            for k in range(K)
        ])

        # Step 4: Check for convergence based on relative shifts
        centroid_shifts = np.linalg.norm(new_centroids - centroids, axis=1)
        if np.all(centroid_shifts <= tol):
            print(f"K-Means++ (Offline) converged in {iteration + 1} iterations.")
            break

        centroids = new_centroids
    else:
        print(f"K-Means++ (Offline) reached maximum iterations ({max_iters}).")

    # Calculate final inertia
    final_distances = np.linalg.norm(X - centroids[labels], axis=1)
    inertia = np.sum(final_distances ** 2)

    return centroids, labels, inertia

# ----- K-Means++ (Online) -----
def online_kmeans_PP(X, K, max_iters=10, eta=0.1, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)

    n_samples, n_features = X.shape

    # Step 1: Initialize centroids using K-Means++ initialization
    centroids = kmeans_plus_plus_initialization(X, K, random_state=random_state)

    # Run for max_iters epochs over the data
    for iteration in range(max_iters):
        # Shuffle the data
        indices = np.arange(n_samples)
        np.random.shuffle(indices)
        for idx in indices:
            x = X[idx]
            # Find the nearest centroid
            distances = np.linalg.norm(centroids - x, axis=1)
            m = np.argmin(distances)
            # Update centroid m using the learning rule
            centroids[m] += eta * (x - centroids[m])
        print(f"K-Means++ (Online) completed iteration {iteration + 1}.")

    # After training, assign labels based on the nearest centroid
    distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)
    labels = np.argmin(distances, axis=1)

    # Calculate final inertia
    final_distances = np.linalg.norm(X - centroids[labels], axis=1)
    inertia = np.sum(final_distances ** 2)

    return centroids, labels, inertia


K = 5

centroids_offline, labels_offline, inertia_offline = offline_kmeans(
    principal_components, K, max_iters=100, tol=1e-4, random_state=42
)
print(f"Offline K-Means Final Inertia: {inertia_offline:.2f}")

centroids_online, labels_online, inertia_online = online_kmeans(
    principal_components, K, max_iters=10, eta=0.1, random_state=42
)
print(f"Online K-Means Final Inertia: {inertia_online:.2f}")

centroids_offline_PP, labels_offline_PP, inertia_offline_PP = offline_kmeans_PP(
    principal_components, K, max_iters=100, tol=1e-4, random_state=42
)
print(f"K-Means++ (Offline) Final Inertia: {inertia_offline_PP:.2f}")

centroids_online_PP, labels_online_PP, inertia_online_PP = online_kmeans_PP(
    principal_components, K, max_iters=10, eta=0.1, random_state=42
)
print(f"K-Means++ (Online) Final Inertia: {inertia_online_PP:.2f}")


# Define a function to plot clustering results
def plot_clusters(ax, pc1, pc2, labels, centroids, title, palette):
    K = centroids.shape[0]
    for k in range(K):
        ax.scatter(pc1[labels == k],
                   pc2[labels == k],
                   s=50,
                   alpha=0.6,
                   edgecolor='k',
                   label=f'Cluster {k+1}',
                   color=palette[k])
    ax.scatter(centroids[:, 0], centroids[:, 1],
               s=200, c='red', marker='X', label='Centroids')
    ax.set_title(title, fontsize=14)
    ax.set_xlabel(f'First Principal Component ({explained_variance[0]*100:.1f}%)', fontsize=12)
    ax.set_ylabel(f'Second Principal Component ({explained_variance[1]*100:.1f}%)', fontsize=12)
    ax.grid(True)
    ax.legend()

# Create a 2x2 subplot grid
fig, axes = plt.subplots(2, 2, figsize=(20, 18))

# Define a color palette
palette = sns.color_palette('bright', K)

# Top-Left: Standard K-Means (Offline)
plot_clusters(
    ax=axes[0, 0],
    pc1=principal_component_1,
    pc2=principal_component_2,
    labels=labels_offline,
    centroids=centroids_offline,
    title='Standard K-Means (Offline), inertia = {:.2f}'.format(inertia_offline),
    palette=palette
)

# Top-Right: Standard K-Means (Online)
plot_clusters(
    ax=axes[0, 1],
    pc1=principal_component_1,
    pc2=principal_component_2,
    labels=labels_online,
    centroids=centroids_online,
    title='Standard K-Means (Online), inertia = {:.2f}'.format(inertia_online),
    palette=palette
)

# Bottom-Left: K-Means++ (Offline)
plot_clusters(
    ax=axes[1, 0],
    pc1=principal_component_1,
    pc2=principal_component_2,
    labels=labels_offline_PP,
    centroids=centroids_offline_PP,
    title='K-Means++ (Offline), inertia = {:.2f}'.format(inertia_offline_PP),
    palette=palette
)

# Bottom-Right: K-Means++ (Online)
plot_clusters(
    ax=axes[1, 1],
    pc1=principal_component_1,
    pc2=principal_component_2,
    labels=labels_online_PP,
    centroids=centroids_online_PP,
    title='K-Means++ (Online), inertia = {:.2f}'.format(inertia_online_PP),
    palette=palette
)

# Adjust layout
plt.tight_layout()
plt.show()
```

```
Offline K-Means converged in 12 iterations.
Offline K-Means Final Inertia: 106184.49
Online K-Means completed iteration 1.
Online K-Means completed iteration 2.
Online K-Means completed iteration 3.
Online K-Means completed iteration 4.
Online K-Means completed iteration 5.
Online K-Means completed iteration 6.
Online K-Means completed iteration 7.
Online K-Means completed iteration 8.
Online K-Means completed iteration 9.
Online K-Means completed iteration 10.
Online K-Means Final Inertia: 107718.42
K-Means++ (Offline) converged in 8 iterations.
K-Means++ (Offline) Final Inertia: 98093.83
K-Means++ (Online) completed iteration 1.
K-Means++ (Online) completed iteration 2.
K-Means++ (Online) completed iteration 3.
K-Means++ (Online) completed iteration 4.
K-Means++ (Online) completed iteration 5.
K-Means++ (Online) completed iteration 6.
K-Means++ (Online) completed iteration 7.
K-Means++ (Online) completed iteration 8.
K-Means++ (Online) completed iteration 9.
K-Means++ (Online) completed iteration 10.
K-Means++ (Online) Final Inertia: 98848.00
```









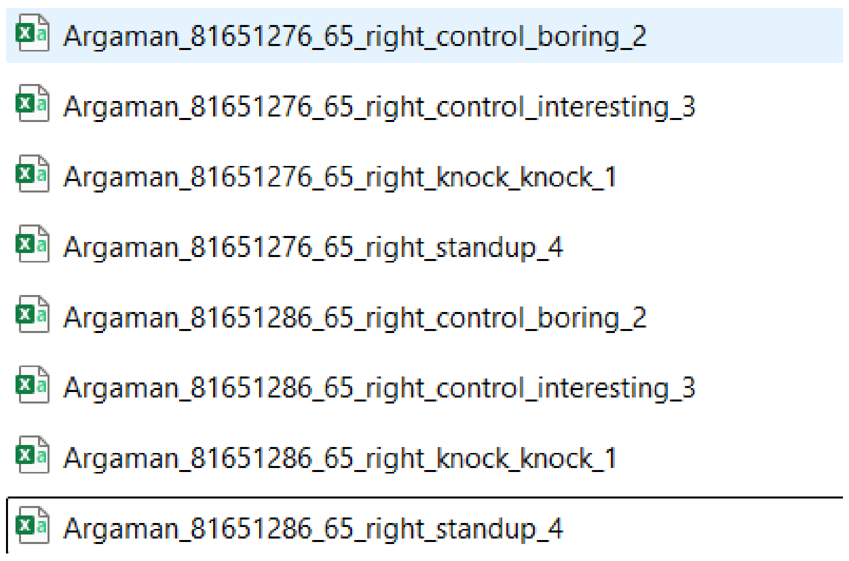
K-means Summary

Offline methods did better (lower inertia), K-means++ did much better than the standard version

Statistical tests

Now, we're going the address each pair correlation, using distance in the pca plot and cluster classifying from the K-means algorithm

We're going to look at indices with jumps of 4 because our folder looks like this:


Argaman_81651276_65_right_control_boring_2
Argaman_81651276_65_right_control_interesting_3
Argaman_81651276_65_right_knock_knock_1
Argaman_81651276_65_right_standup_4
Argaman_81651286_65_right_control_boring_2
Argaman_81651286_65_right_control_interesting_3
Argaman_81651286_65_right_knock_knock_1
Argaman_81651286_65_right_standup_4

Here we have pair 65. We can see the matrices for subject 81651276 and then the recordings for subject 81651286.

Distance Between same manipulations for related pair of subjects

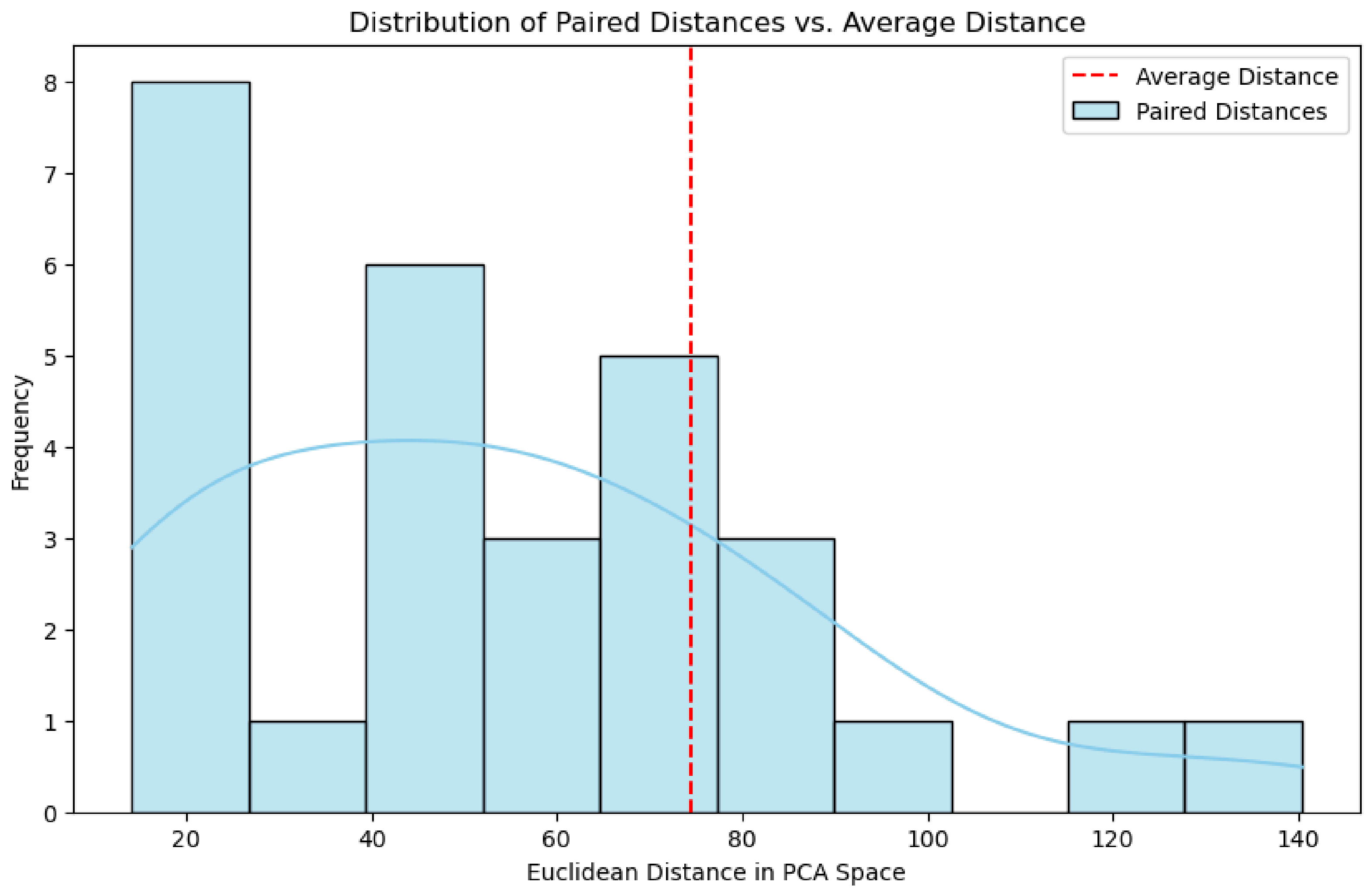
We're going to investigate the relationship between samples of related people in the same manipulation, compared to all other combination of 2 arbitrary samples. We're going to examine it using the distance between the dots in the principal components projected graph.

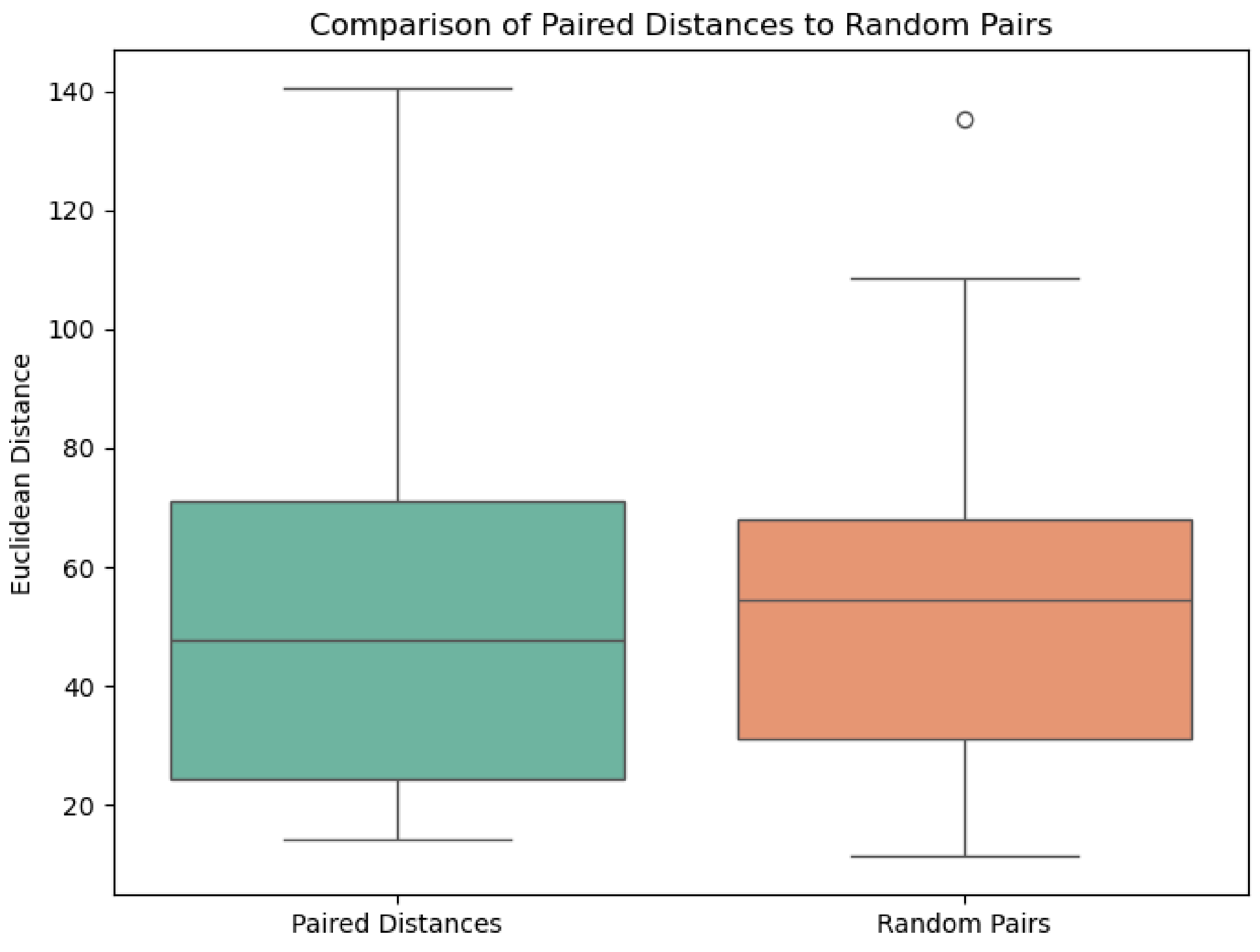
In [37]:
```python
import numpy as np

# Assuming principal_components is a NumPy array of shape (n_samples, 2)
principal_components = principal_components  # Replace with your PCA data

# Define the paired indices (e.g., 0 and 4, 1 and 5, 2 and 6, etc.)
paired_indices = [(i, i + 4) for i in range(0, len(principal_components) - 4, 4)]

# Calculate Euclidean distances for each pair
pair_distances = []
for idx1, idx2 in paired_indices:
    point1 = principal_components[idx1]
    point2 = principal_components[idx2]
    distance = np.linalg.norm(point1 - point2)
    pair_distances.append(distance)
    print(f"Distance between index {idx1} and {idx2}: {distance:.4f}")

# Calculate the average Euclidean distance between all unique pairs
from itertools import combinations

all_pairs = combinations(range(len(principal_components)), 2)
all_distances = [np.linalg.norm(principal_components[i] - principal_components[j])
                for i, j in all_pairs]

average_distance = np.mean(all_distances)
print(f"\nAverage Euclidean distance between all pairs: {average_distance:.4f}")

# Compare paired distances to the average
paired_below_avg = [d for d in pair_distances if d < average_distance]
paired_above_avg = [d for d in pair_distances if d >= average_distance]

print(f"\nNumber of paired distances below average: {len(paired_below_avg)}")
print(f"Number of paired distances above average: {len(paired_above_avg)}")

#Check significance
from scipy.stats import ttest_1samp

# Perform one-sample t-test to check if the paired distances are significantly different from the average
t_stat, p_value = ttest_1samp(pair_distances, average_distance)
print(f"\nOne-sample t-test results:")
print(f"t-statistic: {t_stat:.4f}")
print(f"p-value: {p_value:.4f}")
print("Significant at 5% level" if p_value < 0.05 else "Not significant at 5% level")
```

```
Distance between index 0 and 4: 40.5742
Distance between index 4 and 8: 94.6677
Distance between index 8 and 12: 24.2154
Distance between index 12 and 16: 17.2366
Distance between index 16 and 20: 56.9818
Distance between index 20 and 24: 47.6685
Distance between index 24 and 28: 47.6155
Distance between index 28 and 32: 88.1517
Distance between index 32 and 36: 17.2710
Distance between index 36 and 40: 17.9306
Distance between index 40 and 44: 67.3482
Distance between index 44 and 48: 88.1368
Distance between index 48 and 52: 44.8823
Distance between index 52 and 56: 44.9433
Distance between index 56 and 60: 54.5136
Distance between index 60 and 64: 21.3975
Distance between index 64 and 68: 122.9160
Distance between index 68 and 72: 140.3931
Distance between index 72 and 76: 71.0145
Distance between index 76 and 80: 66.8274
Distance between index 80 and 84: 21.6640
Distance between index 84 and 88: 73.6434
Distance between index 88 and 92: 14.1251
Distance between index 92 and 96: 22.3064
Distance between index 96 and 100: 42.1477
Distance between index 100 and 104: 27.1613
Distance between index 104 and 108: 67.9416
Distance between index 108 and 112: 53.9653
Distance between index 112 and 116: 82.1635

Average Euclidean distance between all pairs: 74.5434

Number of paired distances below average: 23
Number of paired distances above average: 6

One-sample t-test results:
t-statistic: -3.3530
p-value: 0.0023
Significant at 5% level
```

In [39]:

```python
import matplotlib.pyplot as plt
import seaborn as sns

# Plot histogram of paired distances
plt.figure(figsize=(10, 6))
sns.histplot(pair_distances, bins=10, kde=True, color='skyblue', label='Paired Distances')
plt.axvline(average_distance, color='red', linestyle='--', label='Average Distance')
plt.title('Distribution of Paired Distances vs. Average Distance')
plt.xlabel('Euclidean Distance in PCA Space')
plt.ylabel('Frequency')
plt.legend()
plt.show()

# Boxplot comparison
plt.figure(figsize=(8, 6))
sns.boxplot(data=[pair_distances, all_distances[:len(pair_distances)]], palette='Set2')
plt.xticks([0, 1], ['Paired Distances', 'Random Pairs'])
plt.ylabel('Euclidean Distance')
plt.title('Comparison of Paired Distances to Random Pairs')
plt.show()
```





Checking same cluster assignment for related observations

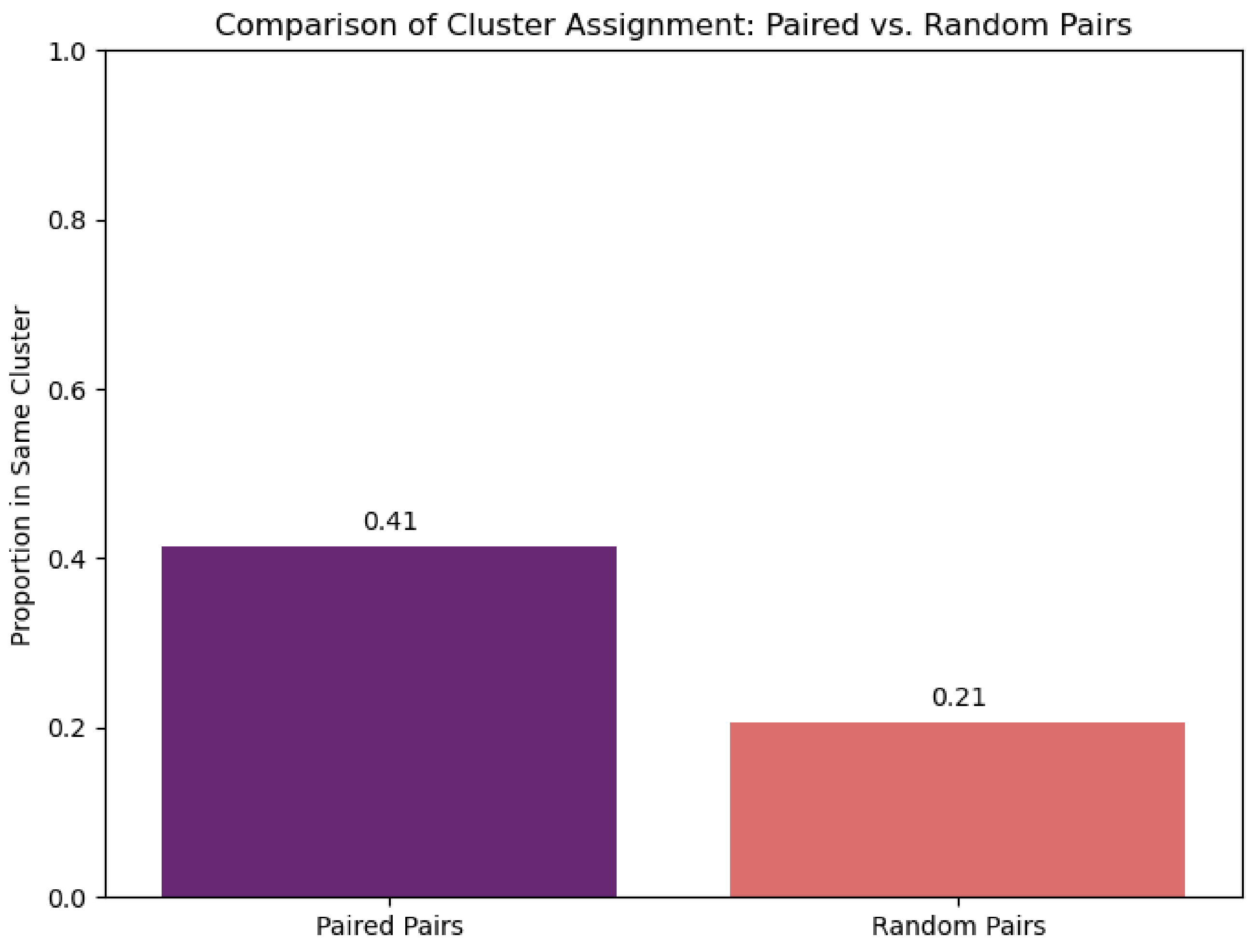
For same manipulation among related people (same pair), we're checking if both matrices were classified in the same cluster

In [61]:
```
import numpy as np
import random
from itertools import combinations
from scipy.stats import chi2_contingency, norm

# Assuming 'principal_components' and 'labels_final' are already defined
# Replace 'labels_final' with the appropriate variable if different
labels_final = labels  # Ensure 'labels_final' refers to your cluster labels

# Define the paired indices
paired_indices = [(i, i + 4) for i in range(0, len(principal_components) - 4, 4)]

# Calculate how many paired indices are in the same cluster
paired_same_cluster = sum([labels_final[idx1] == labels_final[idx2] for idx1, idx2 in paired_indices])
total_paired = len(paired_indices)
paired_proportion = paired_same_cluster / total_paired
print(f"Paired Indices: {paired_same_cluster}/{total_paired} pairs in the same cluster (Proportion: {paired_proportion:.4f})")

# Generate random pairs (same number as paired_indices) without overlapping the paired indices
all_possible_indices = set(range(len(labels_final)))
excluded_indices = set([idx for pair in paired_indices for idx in pair])
remaining_indices = list(all_possible_indices - excluded_indices)
```

```python
# Initialize counters for random pairs
total_random_same_cluster = 0
total_random_paired = 0

# Generate random pairs and accumulate counts
num_iterations = 1000
for _ in range(num_iterations):

    # Shuffle the remaining indices
    sampled_indices = random.sample(remaining_indices, total_paired * 2)

    # Form pairs
    random_pairs = list(zip(sampled_indices[::2], sampled_indices[1::2]))

    # Calculate same-cluster pairs
    same_cluster = sum([labels_final[idx1] == labels_final[idx2] for idx1, idx2 in random_pairs])
    total_random_same_cluster += same_cluster
    total_random_paired += total_paired

# Calculate total counts
random_same_cluster_total = total_random_same_cluster
random_not_same_cluster_total = total_random_paired - random_same_cluster_total

# Create contingency table with integer counts
contingency_table = [
    [paired_same_cluster, total_paired - paired_same_cluster],
    [random_same_cluster_total, random_not_same_cluster_total]
]

# Perform Chi-Square Test
chi2, p, dof, ex = chi2_contingency(contingency_table)
print(f"Chi-Square Test p-value: {p:.4f}")

if p < 0.05:
    print("There is a statistically significant difference in clustering between paired and random pairs (p < 0.05).")
else:
    print("There is no statistically significant difference in clustering between paired and random pairs (p >= 0.05).")
```

```
Paired Indices: 12/29 pairs in the same cluster (Proportion: 0.4138)
Chi-Square Test p-value: 0.0114
There is a statistically significant difference in clustering between paired and random pairs (p < 0.05).
```

In [62]:
```python
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
# Data for plotting
random_proportion = random_same_cluster_total / total_random_paired
categories = ['Paired Pairs', 'Random Pairs']
proportions = [paired_proportion, random_proportion]

# Plotting
plt.figure(figsize=(8, 6))
sns.barplot(x=categories, y=proportions, palette='magma')
plt.ylim(0, 1)
plt.ylabel('Proportion in Same Cluster')
plt.title('Comparison of Cluster Assignment: Paired vs. Random Pairs')
for i, v in enumerate(proportions):
    plt.text(i, v + 0.02, f"{v:.2f}", ha='center')
plt.show()
```



Conclusions

We can see that matrices of same manipulation for a related pair of people have significantly lower distance in between them in the PCA projection graph.

Moreover, these same linked matrices are significantly more likely to be paired in the same cluster compared to arbitrary pairs randomly picked from our list of matrices.