VERDICKT_JULIA-PS7

April 24, 2024

1 Problem Set 7: Unsupervised Learning and Facial Recognition

Warning! Some of the problems in this problem set require heavy computation - you are encouraged to start early so that you don't get stuck at the last minute.

2 Face Recognition

Your goal for this problem set is to explore the basics of a face recognition system. Since the machine learning force is now strong with you, the prompts for this problem set are deliberately vague. Be creative - but be careful! It my be useful to start by implementing this entire problem set on a relatively small subset of all of the images first, before using the full dataset.

2.1 1. Data Preprocessing and Exploration

Download the "Faces in the Wild" data set from this link (roughly 250MB).

2.1.1 1.1. Filter out people with few images

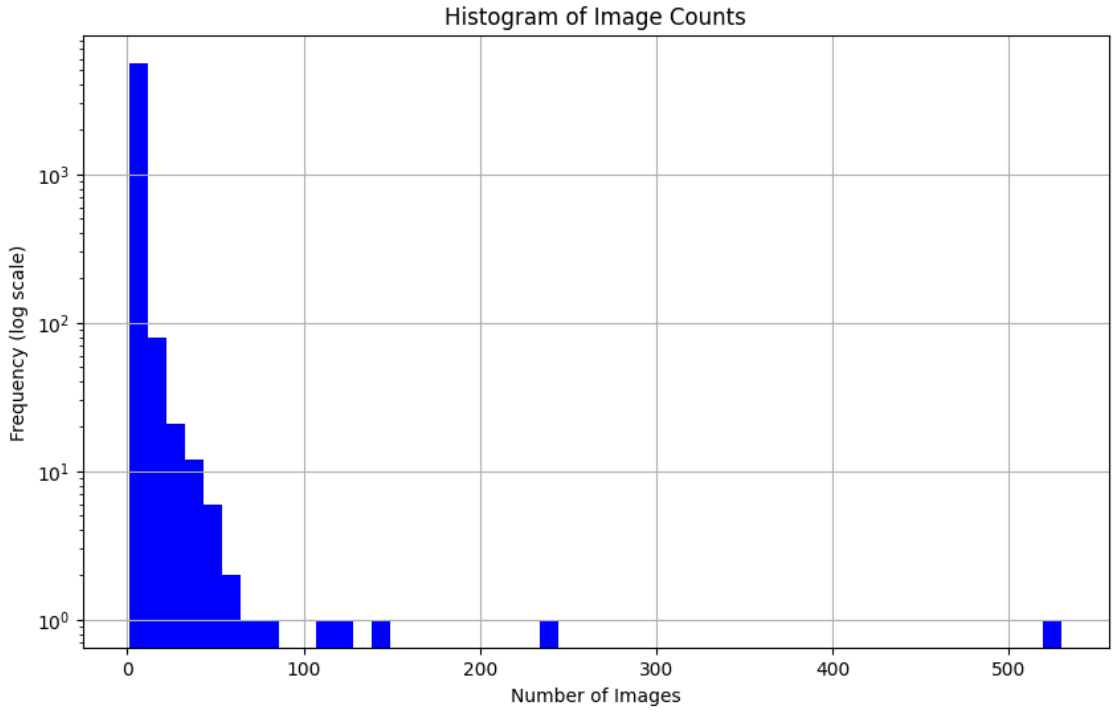
First, display a histogram that shows the number of images per individual (you may use log-scale if you like). Which individual has the most images from your dataset?

Now, remove all individuals for whom you have fewer than 10 images. How many individuals are you left with in the dataset?

[9]:
```
# Your code here

import os
import matplotlib.pyplot as plt
import pandas as pd

# Define the path to the dataset directory
dataset_path = 'lfw_funneled'

# List all subdirectories in the dataset path
people = [d for d in os.listdir(dataset_path) if os.path.isdir(os.path.
 ↪join(dataset_path, d))]

# Dictionary to store people and their respective image counts
data = {'Person': [], 'Image_Count': []}
```

```
for person in people:
    images = os.listdir(os.path.join(dataset_path, person))
    data['Person'].append(person)
    data['Image_Count'].append(len(images))

df = pd.DataFrame(data)
```

[10]:
```
# Plot a histogram of the image counts
plt.figure(figsize=(10, 6))
plt.hist(df['Image_Count'], bins=50, log=True, color='blue')
plt.title('Histogram of Image Counts')
plt.xlabel('Number of Images')
plt.ylabel('Frequency (log scale)')
plt.grid(True)
plt.show()
```



[11]:
```
# Identify the person with the most images
most_images_person = df.loc[df['Image_Count'].idxmax()]
print("Individual with the most images:", most_images_person['Person'], "with",␣
↪most_images_person['Image_Count'], "images")


# Filter out individuals with fewer than 10 images
filtered_df = df[df['Image_Count'] >= 10]
```

```
filtered_image_paths = []
for person in filtered_df['Person']:
    person_path = os.path.join(dataset_path, person)
    images = os.listdir(person_path)
    filtered_image_paths.extend([os.path.join(person_path, image) for image in␣
 ↪images if image.lower().endswith('.jpg')])

print(f"Number of individuals with at least 10 images: {filtered_df.shape[0]}")
print(f"Total images: {len(filtered_image_paths)}")
```

```
Individual with the most images: George_W_Bush with 530 images
Number of individuals with at least 10 images: 158
Total images: 4324
```

Your comments here

- Which individual has the most images from your dataset?
  - George W. Bush has the most images.
- How many individuals are you left with in the dataset?
  - 158 individuals are left.

2.1.2 1.2. Show some faces!

1. Pick 10 random images from the dataset and display them in a 2 x 5 grid.
2. This time pick 200 random images. Create a single image that shows the "average face" over this random sample – i.e. each pixel should display the average value of that particular pixel across the random sample.

[12]:
```
# Your code here

import random
from PIL import Image

# Randomly select 10 images
selected_images = random.sample(filtered_image_paths, 10)

# Plotting the images in a 2 x 5 grid
plt.figure(figsize=(10, 5))
for i, img_path in enumerate(selected_images):
    img = Image.open(img_path)
    plt.subplot(2, 5, i + 1)  # Rows, columns, index
    plt.imshow(img)
    plt.axis('off')
plt.tight_layout()
plt.show()
```

[13]:
```
import numpy as np

# Randomly select 200 images
selected_images = random.sample(filtered_image_paths, 200)

# Initialize an array to accumulate pixel values
average_image_data = None
count = 0

for img_path in selected_images:
    img = Image.open(img_path)
    img_array = np.array(img)

    # Initialize the array with the shape of the first image
    if average_image_data is None:
        average_image_data = np.zeros_like(img_array, dtype=np.float32)

    # Accumulate image data
    average_image_data += img_array

# Calculate the average
average_image_data /= len(selected_images)

# Convert back to uint8 image format
average_image = np.uint8(average_image_data)

# Display the average face
plt.figure(figsize=(5, 5))
plt.imshow(average_image)
plt.axis('off')
```

```
plt.show()
```

2.2 2. k-Means Clustering

2.2.1 2.1. Implementation

Implement the k-Means clustering algorithm that we discussed in class, using the Euclidean distance function. This will require that you define three new functions: 1. InitializeCentroids(X, k) 2. FindClosestCentroids(X, centroids) 3. ComputeCentroidMeans(X, centroids, k)

Hint: You may find the `pairwise_distances_argmin` function useful.

[14]:
```
# Your code here

from sklearn.metrics import pairwise_distances_argmin


def initialize_centroids(X, k):
    """ Randomly picks k rows of X as initial centroids. """
    indices = np.random.permutation(X.shape[0])
    centroids = X[indices[:k]]
    return centroids
```

```python
#This is an older version of the function
# I don't use but want to keep here for my own reference
#def find_closest_centroids(X, centroids):
   # """ Assigns each sample in X to the closest centroid. """
   # distances = np.sqrt(((X - centroids[:, np.newaxis])**2).sum(axis=2))
   # return np.argmin(distances, axis=0)

def find_closest_centroids(X, centroids, metric = 'euclidean'):
    """ Assigns each sample in X to the closest centroid. """
    return pairwise_distances_argmin(X, centroids, metric = metric)

def compute_centroid_means(X, indices, k):
    """ Recomputes centroids as the mean of all samples assigned to each␣
 ↪cluster. """
    centroids = np.array([X[indices==i].mean(axis=0) for i in range(k)])
    return centroids

def run_k_means(X, k, metric = 'euclidean', max_iters=100, verbose = False):
    """ Runs the k-means algorithm on data X with k clusters for a maximum of␣
 ↪max_iters iterations. """
    np.random.seed(10)
    centroids = initialize_centroids(X, k)
    for i in range(max_iters):
        indices = find_closest_centroids(X, centroids, metric = metric)
        new_centroids = compute_centroid_means(X, indices, k)

        # Optional verbose output to track progress
        if verbose:
            print(f"Iteration {i + 1}/{max_iters}: Centroid positions updated.")
        #if np.all(centroids == new_centroids):
            #break
            # #this was my old stopping criterion. The criterion is strict.
            # I keep it here for my own reference
        if np.allclose(centroids, new_centroids, atol=1e-4):
            if verbose:
                print("Convergence reached.")
            break
        centroids = new_centroids
        if i == max_iters:
            print("No convergence, max_iters reached")
    return centroids, indices

# Example Usage:
# Suppose X is your dataset (an array of shape (num_samples, num_features))
# k is the number of clusters you want
# centroids, indices = run_k_means(X, k)
```

2.2.2 2.2. Sanity check

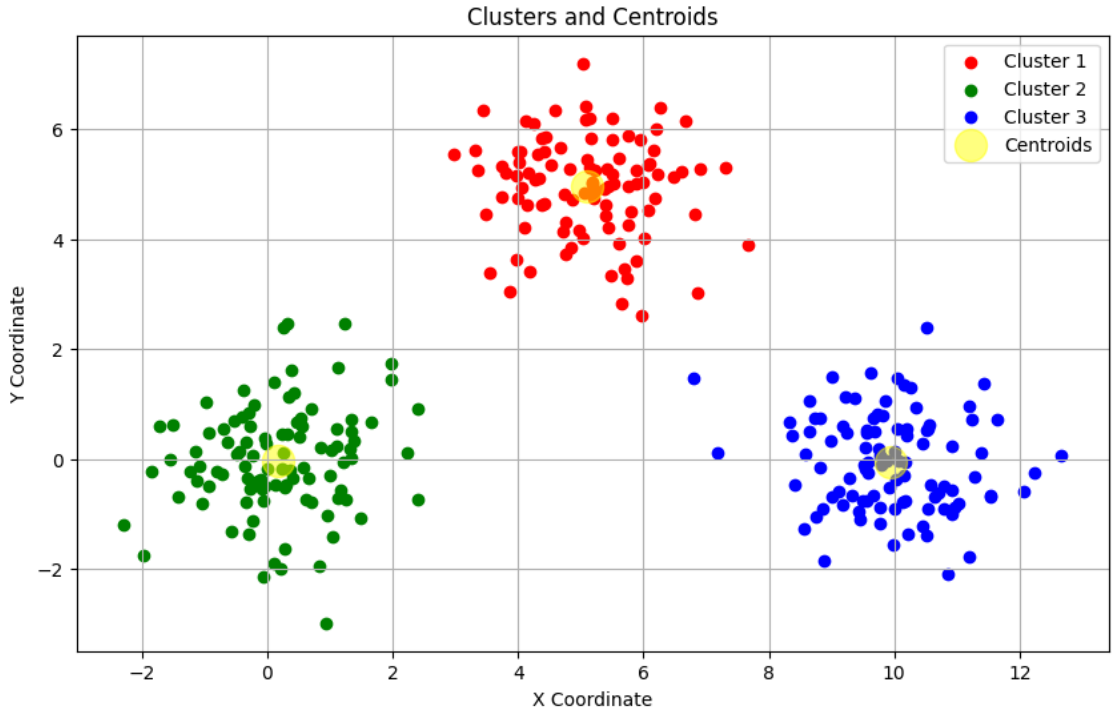
Apply your k-Means algorithm to a toy dataset to make sure it works properly. Also create a scatterplot that shows these datapoints, colored by cluster.

[15]:
```
# Your code here

# Generate random data around multiple centers
np.random.seed(10)  # for reproducibility
data1 = np.random.randn(100, 2) + np.array([0, 0])
data2 = np.random.randn(100, 2) + np.array([5, 5])
data3 = np.random.randn(100, 2) + np.array([10, 0])

# Combine the data into one larger dataset
X = np.vstack([data1, data2, data3])

# Run k-means on the toy dataset with k=3
centroids, indices = run_k_means(X, k=3, verbose = True)
```

```
Iteration 1/100: Centroid positions updated.
Iteration 2/100: Centroid positions updated.
Iteration 3/100: Centroid positions updated.
Iteration 4/100: Centroid positions updated.
Iteration 5/100: Centroid positions updated.
Iteration 6/100: Centroid positions updated.
Iteration 7/100: Centroid positions updated.
Iteration 8/100: Centroid positions updated.
Iteration 9/100: Centroid positions updated.
Iteration 10/100: Centroid positions updated.
Iteration 11/100: Centroid positions updated.
Iteration 12/100: Centroid positions updated.
Convergence reached.
```

[16]:
```
plt.figure(figsize=(10, 6))
colors = ['r', 'g', 'b']

for i in range(3):
    # Plot data points that are assigned to each cluster
    plt.scatter(X[indices == i, 0], X[indices == i, 1], color=colors[i],␣
 ↪label=f'Cluster {i+1}')

# Also plot centroids
plt.scatter(centroids[:, 0], centroids[:, 1], s=300, c='yellow',␣
 ↪label='Centroids', alpha=0.5)

plt.title('Clusters and Centroids')
plt.xlabel('X Coordinate')
plt.ylabel('Y Coordinate')
```

```
plt.legend()
plt.grid(True)
plt.show()
```



2.2.3 2.3. Image centroids

Apply your k-Means algorithm to the images dataset, using k=10. Make sure to standardize your data first! Show a 10 x 2 grid of images where the first column contains (a) the image that represents the centroid for each of those clusters, and the second column contains (b) the closest image in the original dataset to that centroid. Use Euclidean distance. What do you notice?

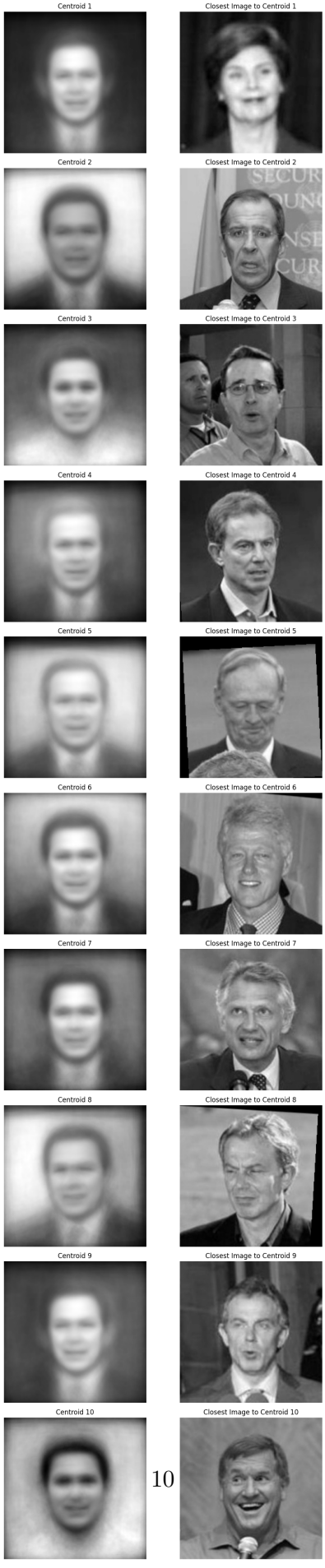
Note: As you may notice, there are 250x250x3 = 187500 features for each image. If your k-Means algorithm is not implemented efficiently – and even if it is – it might take a long time for your algorithm to converge. If your computer is slow, it might even take a very long time for you to simply standardize your data. We recommend you convert your RGB images into grayscale first (using, for instance, the `rgb2grey` function, or any other way to convert to graysacle) before standardizing.

[17]:
```
# Your code here

from skimage.color import rgb2gray
from skimage.io import imread
from sklearn.preprocessing import StandardScaler
```

```
verbose = False
# Load images from the filtered paths, convert to grayscale, and flatten
image_vectors = []
for index, img_path in enumerate(filtered_image_paths):
    image = imread(img_path)
    grayscale_image = rgb2gray(image)  # Convert to grayscale
    image_vectors.append(grayscale_image.flatten())
    if verbose:
        # Print the progress
        print(f"Processed {index + 1}/{len(filtered_image_paths)} images")

# Convert list to numpy array
X = np.array(image_vectors)

# Standardize the data
scaler = StandardScaler()
X_std = scaler.fit_transform(X)
```

[18]:
```
# Apply k-means
k = 10
centroids, indices = run_k_means(X_std, k, verbose = False)
```

[19]:
```
# Reshape centroids to original dimensions
centroid_images = scaler.inverse_transform(centroids).reshape(-1, 250, 250)
closest_image_indices = pairwise_distances_argmin(centroids, X_std)

# Plot the images
fig, axs = plt.subplots(nrows=10, ncols=2, figsize=(10, 40))
for i, centroid in enumerate(centroid_images):

    # Display the centroid image
    axs[i, 0].imshow(centroid, cmap='gray')
    axs[i, 0].axis('off')
    axs[i, 0].set_title(f'Centroid {i+1}')

    # Display the closest original image to the centroid
    closest_image_path = filtered_image_paths[closest_image_indices[i]]
    closest_image = imread(closest_image_path, as_gray=True)

    # Display the closest image to the centroid in grayscale
    axs[i, 1].imshow(closest_image, cmap='gray')
    axs[i, 1].axis('off')
    axs[i, 1].set_title(f'Closest Image to Centroid {i+1}')

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


Centroid 1
Closest Image to Centroid 1
Centroid 2
Closest Image to Centroid 2
Centroid 3
Closest Image to Centroid 3
Centroid 4
Closest Image to Centroid 4
Centroid 5
Closest Image to Centroid 5
Centroid 6
Closest Image to Centroid 6
Centroid 7
Closest Image to Centroid 7
Centroid 8
Closest Image to Centroid 8
Centroid 9
Closest Image to Centroid 9
Centroid 10
Closest Image to Centroid 10

Your comments here

- What do you notice?
  - The majority of the closest images in the right column except the first are older white men.

2.2.4 2.4 Classify yourself

Take a picture of yourself and scale it so that it is in a similar format to the images in the dataset. Show four images: 1. The image of yourself. 2. The cluster centroid closest to that image. 3. The closest image in the dataset to that cluster centroid. 4. The closest image in the dataset to the image of yourself.

[20]:
```
# Your code here
from skimage.transform import resize


my_image_path = "IMG_0944_Small.jpeg"
my_image = imread(my_image_path)

resized_my_image = resize(my_image, (250, 250), anti_aliasing=True)

# Convert the image to grayscale
my_gray_image = rgb2gray(resized_my_image)

# Reshape the image to match the dimensions of the dataset images
scaled_my_image = my_gray_image.flatten()

# Find the closest centroid to my image
closest_centroid_index = pairwise_distances_argmin(scaled_my_image.reshape(1,␣
 ↪-1), centroids)

# Find the closest image in the dataset to the centroid
closest_dataset_image_index =␣
 ↪pairwise_distances_argmin(centroids[closest_centroid_index], X_std)
```

[21]:
```
# Plot the images
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(10, 10))

# Plot your image
axs[0, 0].imshow(my_gray_image, cmap='gray')
axs[0, 0].set_title('My Image')
axs[0, 0].axis('off')

# Plot the centroid closest to my image
axs[0, 1].imshow(centroid_images[closest_centroid_index][0], cmap='gray')
```

```
axs[0, 1].set_title('Closest Centroid')
axs[0, 1].axis('off')

# Plot the closest image in the dataset to the centroid
closest_image_path = filtered_image_paths[closest_dataset_image_index[0]]
closest_image = imread(closest_image_path, as_gray=True)
axs[1, 0].imshow(closest_image, cmap='gray')
axs[1, 0].set_title('Closest Image to Centroid')
axs[1, 0].axis('off')

# Plot the closest image in the dataset to my image
closest_image_index = pairwise_distances_argmin(scaled_my_image.reshape(1, -1),␣
 ↪X_std)
axs[1, 1].imshow(X[closest_image_index[0]].reshape(250, 250), cmap='gray')
axs[1, 1].set_title('Closest Image to My Image')
axs[1, 1].axis('off')

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

2.2.5 2.5. (Extra credit): k-Means++

- In 2.1 you implemented k-Means clustering with random initialization of the centroids. In this part implement the k-Means++ version of the algorithm that uses a "smarter" initialization of the centroids in order to achieve faster convergence. Compare the number of iterations it took k-Means in 2.3 to converge with random initialization to the number of iterations it takes k-Means++. Also compare the sum of squared errors that you obtain for both methods. Use for both k=20 clusters. The following link shows the paper that proposed k-Means++. http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf

[22]:
```
# Your code here
```

```python
import numpy as np

def k_means_plus_plus_init(X, k):
    """ Initialize centroids using the k-Means++ algorithm. """
    centroids = [X[np.random.randint(X.shape[0])]]
    for _ in range(1, k):
        distances = np.array([min([np.inner(c-x, c-x) for c in centroids]) for x in X])
        probabilities = distances / distances.sum()
        cumulative_probabilities = np.cumsum(probabilities)
        r = np.random.rand()
        for j, p in enumerate(cumulative_probabilities):
            if r < p:
                centroids.append(X[j])
                break
    return np.array(centroids)

def run_k_means_extra(X, k, initialization='random', max_iters=100, verbose=False, random_state = 10):
    """ Run the k-Means algorithm with the option of 'random' or 'k-means++' initialization. """
    np.random.seed(10)
    if initialization == 'k-means++':
        centroids = k_means_plus_plus_init(X, k)
    else:
        centroids = initialize_centroids(X, k)

    for i in range(max_iters):
        indices = find_closest_centroids(X, centroids)
        new_centroids = compute_centroid_means(X, indices, k)
        if np.allclose(centroids, new_centroids, atol=1e-4):
            if verbose:
                print(f"Convergence reached at iteration {i+1} with method {initialization}.")
            break
        centroids = new_centroids

    if verbose:
        print(f"Total iterations: {i+1}")

    # Calculate the sum of squared errors
    sse = sum(np.sum((X[indices == idx] - centroids[idx]) ** 2) for idx in range(k))
    return centroids, indices, sse

# Usage
X = np.random.rand(100, 2)  # Example data
```

```
k = 20
centroids_random, indices_random, sse_random = run_k_means_extra(X, k,␣
 ↪'random', verbose=True)
centroids_kpp, indices_kpp, sse_kpp = run_k_means_extra(X, k, 'k-means++',␣
 ↪verbose=True)

print(f"Random initialization SSE: {sse_random}")
print(f"k-Means++ initialization SSE: {sse_kpp}")
```

```
Convergence reached at iteration 7 with method random.
Total iterations: 7
Convergence reached at iteration 4 with method k-means++.
Total iterations: 4
Random initialization SSE: 0.7498099177547045
k-Means++ initialization SSE: 0.5404287019917858
```

Your comments here

Compare the number of iterations it took k-Means in 2.3 to converge with random initialization to the number of iterations it takes k-Means++. Also compare the sum of squared errors that you obtain for both methods. Use for both k=20 clusters. The following link shows the paper that proposed k-Means++.

- Convergence reached at iteration 7 with method random. Total iterations: 7
- Convergence reached at iteration 4 with method k-means++. Total iterations: 4
- Random initialization SSE: 0.7498099177547045
- k-Means++ initialization SSE: 0.5404287019917858

We see that k-Means ++ has fewer iterations and lower SSE, which makes it preferable.

2.3 3. PCA and Eigenfaces

Ensure you are using standardized data. Then, set aside 50% of the images as "test" data and using the remaining images as "training" data.

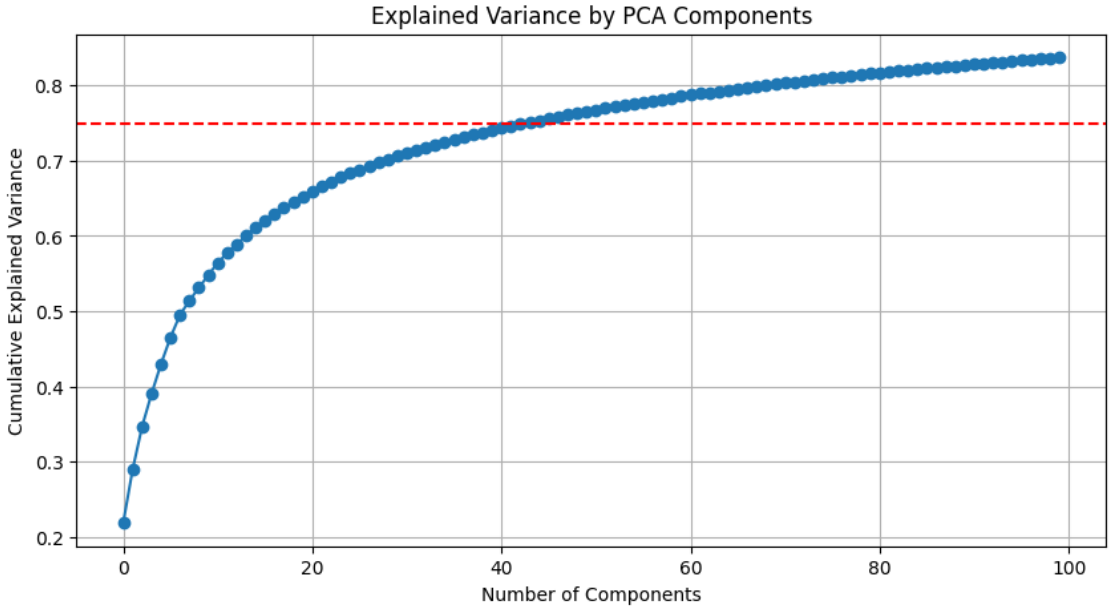
2.3.1 3.1. How many components?

Run principal component analysis using PCA from sklearn on the training dataset. Create a figure showing how the amount of variance explained by your components increases as you increase the number of components from 1 to 100. How many components are required to explain 75% of the variation in your original data?

[23]:
```
# Your code here
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA


X_train, X_test = train_test_split(X_std, test_size=0.5, random_state=10)
```

[24]:
```
# Initialize PCA with enough components to possibly explain 100% of variance
pca = PCA(n_components=100)
pca.fit(X_train)

# Calculate the cumulative variance explained by the principal components
cumulative_variance = np.cumsum(pca.explained_variance_ratio_)
```

[25]:
```
# Plot the explained variance
plt.figure(figsize=(10, 5))
plt.plot(cumulative_variance, marker='o')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Explained Variance by PCA Components')
plt.grid(True)
plt.axhline(y=0.75, color='r', linestyle='--')  # Line to indicate 75% variance␣
 ↪explained
plt.show()

# Find the number of components required to explain at least 75% of the variance
components_required = np.where(cumulative_variance >= 0.75)[0][0] + 1
print("Number of components required to explain at least 75% of variance:",␣
 ↪components_required)
```




```
Number of components required to explain at least 75% of variance: 44
```

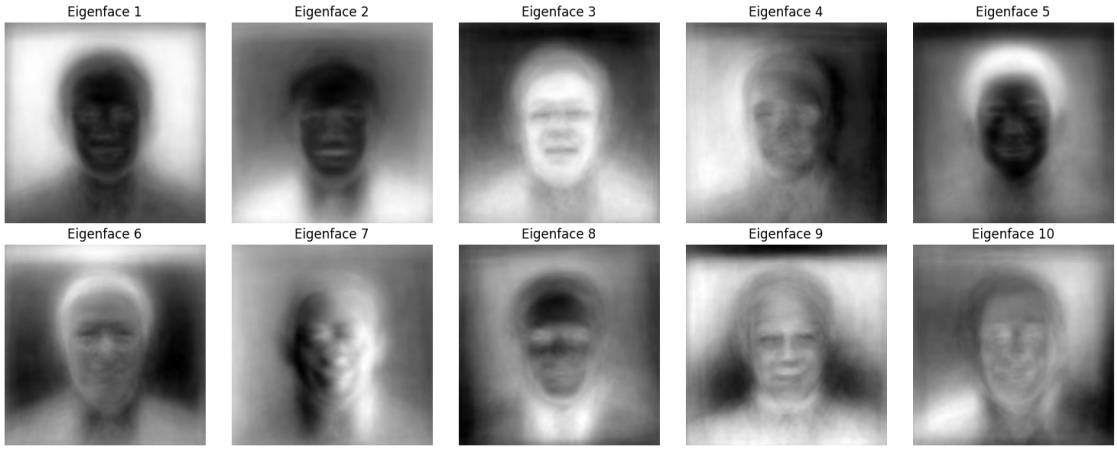
Your comments here

Number of components required to explain at least 75% of variance: 44

2.3.2 3.2. The Eigenfaces

Extract the 10 first principal components (the "eigenfaces") and display them below.

[26]:
```
# Your code here

pca_10 = PCA(n_components=10)
pca_10.fit(X_train)

# Extract the eigenfaces
eigenfaces = pca_10.components_

image_shape = (250, 250)
eigenfaces_images = eigenfaces.reshape((10, *image_shape))

# Plot the first 10 eigenfaces
fig, axes = plt.subplots(2, 5, figsize=(15, 6))  # Create a 2x5 grid for display
axes = axes.flatten()
for i, eigenface in enumerate(eigenfaces_images):
    ax = axes[i]
    ax.imshow(eigenface, cmap='gray')
    ax.axis('off')  # Hide axes
    ax.set_title(f'Eigenface {i+1}')

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



2.3.3 3.3. Projections, clustering and PCA

- Project all of the training and test data into the 40-dimensional space defined by the first 40 principal components.
- Apply k-means clustering, with k=16, to the 40-dimensional projections of the training data.

- Display a 4 x 4 grid of images that shows what each of the 16 centroids look like after the centroid is projected back into the original-image space.

[27]:
```
# Your code here

pca_40 = PCA(n_components=40)
X_train_pca_40 = pca_40.fit_transform(X_train)  # Fit and transform training␣
↪data
X_test_pca_40 = pca_40.transform(X_test)
```

[28]:
```
k = 16
centroids_pca_40, indices = run_k_means(X_train_pca_40, k, verbose=False)
```

[29]:
```
centroids_original = pca_40.inverse_transform(centroids_pca_40)
```

[30]:
```
import matplotlib.pyplot as plt

image_shape = (250, 250)
centroids_images = centroids_original.reshape((-1, *image_shape))

# Plot the centroids as images in a 4x4 grid
fig, axes = plt.subplots(4, 4, figsize=(10, 10))
for i, ax in enumerate(axes.flatten()):
    ax.imshow(centroids_images[i], cmap='gray')
    ax.axis('off')
plt.tight_layout()
plt.show()
```

2.3.4 3.4. (Extra Credit): Recognition

Create a set of 10 images using (i) 8 images randomly selected from the test data, (ii) any other non-face image you can find that has the right dimensions, and (iii) an image of yourself. Create a 10 x 5 grid of images, with one row for each of these images, and 5 columns that contain: 1. The original image. 2. The reconstruction of that image after it is projected onto the 40-dimensional eigenface-space and then re-projected back into the original image space. 3. Find the nearest centroid (from 3.3) to the image, and show the reconstruction of that nearest centroid. 4. Find the image in the training data whose 40-dimensional representation is closest to that centroid, and show the reconstruction of that image. 5. Show the original training image that was selected in above (step 4 in this list)

Post the five images corresponding to your headshot on piazza to share with the rest of the class.

[31]:
```
# Your code here

non_face_image_path = "Antwerpen_Brabo.jpg"
non_face_image = imread(non_face_image_path)

resized_non_face_image = resize(non_face_image, (250, 250), anti_aliasing=True)

# Convert the image to grayscale
non_face_gray_image = rgb2gray(resized_non_face_image)

# Reshape your image to match the dimensions of the dataset images
scaled_non_face = non_face_gray_image.flatten()

# Select 8 random images from the test set
indices = np.random.choice(range(len(X_test_pca_40)), 8, replace=False)
random_test_images = X_test[indices]

# Combine all images into one array
images = np.vstack([random_test_images, scaled_non_face, scaled_my_image])

# Project these images to 40D space and back
images_pca_40 = pca_40.transform(images)
images_projected = pca_40.inverse_transform(images_pca_40)

# Find nearest centroids and corresponding closest training image
nearest_centroid_indices = pairwise_distances_argmin(images_pca_40,␣
 ↪centroids_pca_40)
nearest_images_indices =␣
 ↪pairwise_distances_argmin(centroids_pca_40[nearest_centroid_indices],␣
 ↪X_train_pca_40)
```

[32]:
```
# Plotting
fig, axes = plt.subplots(10, 5, figsize=(25, 50))

for i in range(10):
    # Original image
    axes[i, 0].imshow(images[i].reshape(250, 250), cmap='gray')
    axes[i, 0].axis('off')
    axes[i, 0].set_title('Original Test Image')

    # Reconstructed from PCA
    axes[i, 1].imshow(images_projected[i].reshape(250, 250), cmap='gray')
    axes[i, 1].axis('off')
    axes[i, 1].set_title('Reconstruction from PCA')

    # Nearest centroid reconstruction
```

```
    axes[i, 2].imshow(centroids_images[nearest_centroid_indices[i]].
 ↪reshape(250, 250), cmap='gray')
    axes[i, 2].axis('off')
    axes[i, 2].set_title('Nearest Centroid')

    # Closest training image to that centroid, reconstructed
    closest_image_pca = X_train_pca_40[nearest_images_indices[i]]
    closest_image_reconstructed = pca_40.inverse_transform(closest_image_pca)
    axes[i, 3].imshow(closest_image_reconstructed.reshape(250, 250),␣
 ↪cmap='gray')
    axes[i, 3].axis('off')
    axes[i, 3].set_title('Closest Train Image Reconstructed')

    # Original training image that was selected
    axes[i, 4].imshow(X_train[nearest_images_indices[i]].reshape(250, 250),␣
 ↪cmap='gray')
    axes[i, 4].axis('off')
    axes[i, 4].set_title('Original Training Image')

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


Original Test Image
Reconstruction from PCA
Nearest Centroid
Closest Train Image Reconstructed
Original Training Image