

Eigenfaces: PCA for Unconstrained Facial Recognition

21-241 Final Project

Nir Pechuk (npechuk) and Rohan Naphade (rnaphade)

December 2025

1 Overview

For our project, we developed a system to classify an image of a face as one of a training set of face images. We did this by applying Principal Component Analysis (PCA) to the construct the best-fit, low-rank subspace. We then projected new face images onto our best-fit subspace and found the closest known face in order to classify them.

2 Introduction

2.1 Unconstrained Facial Recognition

Facial recognition has many integral applications, from private device security to missing person searches. In order to create systems that can classify images as specific people, a *training dataset* is required, in which labeled $n \times n$ -pixel images of known people are stored as vectors in $\mathbb{R}^{n \times n}$, with each entry signifying the brightness of the associated pixel.¹ When a new image is received, it is compared to the training dataset using a similarity metric and labeled as the closest known face.

However, it is not always possible to standardize the lighting and framing of these training images, motivating the problem of *Unconstrained Face Recognition*. In this case, real-world variations such as different training photo angles, lighting, expressions, and occlusions, make classic techniques such as template matching (where similarity is computed as the norm of the difference of two image vectors) ineffective. Small changes in lighting or angle can introduce large amounts of noise and make images that represent the same face have completely different vector representations. Furthermore, with larger datasets required for Unconstrained Facial Recognition, computing similarity scores on full $n \times n$ -sized images for each image in the training dataset becomes computationally intractable, necessitating a more efficient method that somehow compresses images before comparing them.

¹For simplicity, we deal with grayscale images

2.2 Principal Component Analysis

In order to control for image noise and decrease the dimension of our images, we need some way to find key dimensions that faces vary upon, and only compare our images across these dimensions. Intuitively, as the dataset grows larger, these dimensions will symbolize the ways that faces fundamentally differ, regardless of image noise. This is precisely what *Principal Component Analysis* (PCA) offers, motivating its use in encoding training images and comparing these with new images. This is a seminal method in the field of Computer Vision, often referred to as Eigenfaces [3].

PCA is based on best-fit subspace theory. The best-fit k -dimensional subspace for a set of m vectors $\{x_1, \dots, x_m\}$ such that $x_1, \dots, x_m \in \mathbb{R}^n$ is defined as the subspace $W \subseteq \mathbb{R}^n$ that minimizes $\sum_{i=1}^m \|x_i - \text{proj}_W(x_i)\|$, which, by Pythagorean Theorem, is the same as maximizing $\sum_{i=1}^m \|\text{proj}_W(x_i)\|$. We define $A = [x_1 \ \dots \ x_m]^\top \in M_{m \times n}(\mathbb{R})$ and let $A = U\Sigma V^\top$ be a singular value decomposition of A , where $V = [v_1 \dots v_n]$. By lecture, $\text{Span}\{v_1, \dots, v_k\}$ has the property of maximizing the prior sum of projection norms and thus, $W = \text{Span}\{v_1, \dots, v_k\}$. Furthermore, by lecture, these are precisely the top k orthogonal dimensions along which the vectors $\{x_1, \dots, x_m\}$ exhibit the most variation, which by our hypothesis, will be valuable for distinguishing identities given facial image vectors.

3 Methods

Our facial recognition method resembles other PCA and K-Nearest Neighbor classification systems. We implemented the algorithm in Python with the NumPy library for linear algebra—in provided code snippets, specific variable names and file-handling details may be occluded for readability. See Section 6 for a notebook walking through the full code that was used.

3.1 Training

Given a training dataset of color images which are labeled with identities, we begin by converting each image to grayscale using the Python Imaging Library (PIL), and flattening into a vector of pixel brightness values. We also keep track of labels in an auxiliary array. If testing, we exclude 10% of images from training data.

```
train_images = []
train_labels = []
def image_to_vector(path):
    gray = Image.open(path).convert("L")
    arr = np.array(gray, dtype=np.float32)
    return arr.flatten()

for image, label in dataset:
    train_images.append(image_to_vector(image))
    train_labels.append(label)
```

Then, we conduct the PCA. Given image vectors $x_1, \dots, x_m \in \mathbb{R}^n$ representing m faces of n pixels each, we begin by computing the sample mean, $\mathbf{m} = \frac{1}{m} \sum_{i=1}^m x_i$. This represents the mean data point. For each $i \in \mathbb{N}, 1 \leq i \leq m$, we define the centered vector $\hat{x}_i = x_i - \mathbf{m}$. These centered vectors comprise a mean-centered version of our original data. We let $X = [\hat{x}_1 \ \cdots \ \hat{x}_m]^\top \in M_{m \times n}(\mathbb{R})$, where the rows of X are our mean-centered data points. We let W be the matrix of the first k right singular vectors of X . These vectors are the k most important principal components. Thus, the columns of W span best-fit k -dimensional subspace of our mean-centered data points. We then compute the coordinates of every centered data point relative to the subspace $\text{Col}(W)$ by finding the least squares solution to $Ww_i = \hat{x}_i$ for all $i \in \mathbb{N}, 1 \leq i \leq m$. By Theorem 6.10.6 in the lecture notes, this can be done using the pseudoinverse of W as follows²: $\bar{w}_i = W^\dagger \hat{x}_i$.

```
X_train_uncentered = np.vstack(train_images)
y_train = np.array(train_labels)
m = np.mean(X_train_uncentered, axis=0)
X_train = X_train_uncentered - m

U, S, Vt = np.linalg.svd(X_train)
W = (Vt[:k, :]).T

X_train_proj = np.linalg.pinv(W) @ X_train.T
```

3.2 Classification

Given a new flattened grayscale image vector $x_{test} \in \mathbb{R}^n$, we first compute the corresponding mean-centered data point $\hat{x}_{test} = x_{test} - \mathbf{m}$. Then, we compute the coordinates of x_{test} relative to subspace $\text{Col}(W)$ using the same method: $\bar{w}_{test} = W^\dagger \hat{x}_{test}$. Finally, we find the value $i \in \mathbb{N}, 1 \leq i \leq m$ for which $\|\bar{w}_{test} - \bar{w}_i\|$ is minimized. This corresponds to the data point that is closest, relative to subspace $\text{Col}(W)$, to our new data point. Thus, we classify x_{test} as being most similar to data point x_i , and label it with the label corresponding with x_i . Note that in later trials, we consider the r closest vectors in the training dataset and classify x_{test} as the majority label.

```
x_test_centered = x_test - m
x_test_proj = np.linalg.pinv(W) @ x_test_centered

diffs = X_train_proj - x_test_centered
dists = np.linalg.norm(diffs, axis=0)

classification = y_train[np.argmin(dists)]
```

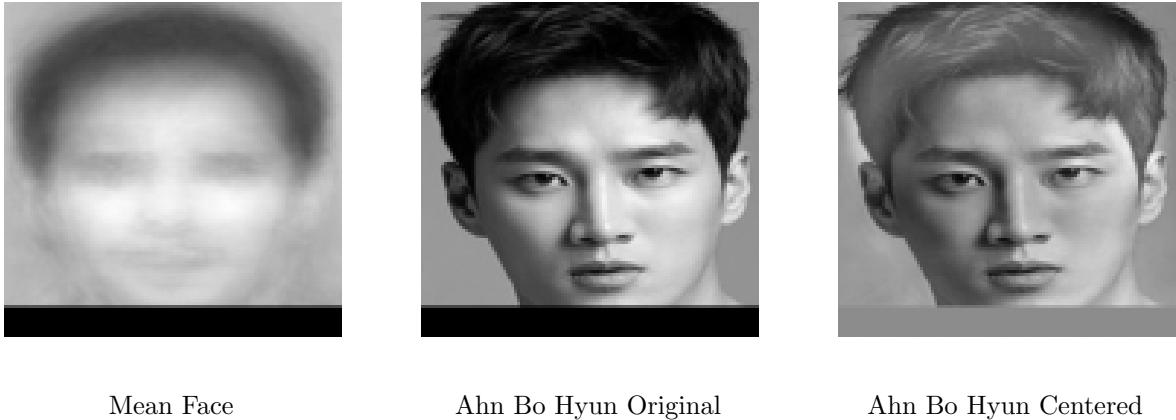
In testing, we repeated this process for each testing vector, comparing against the ground truth label provided by the dataset to compute accuracy and timed this process to compute time. We used the Kaggle "Labelled Faces in the Wild (LFW) Dataset" and "Celebrities Face Dataset Small" [1, 2].

²We used overline to denote least squares solution as to not overload hat notation

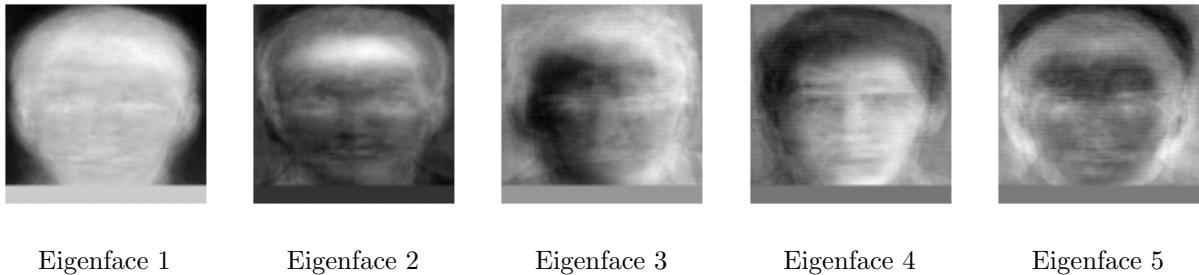
4 Results

4.1 First Attempt: Small Celebrity Dataset

We began by loading a dataset of 140 face images belonging to 14 different celebrities (10 face images per celebrity) [2]. We converted the images to grayscale, then flattened and truncated them so that each flattened image vector had equal dimensionality. We selected one image per celebrity to withhold for test data and used the other nine as training data. We then constructed a train and test matrices $X_{\text{train_uncentered}}$, $X_{\text{test_uncentered}}$, whose rows were the flattened train and test image vectors, respectively. We then subtracted the mean row (mean face) of $X_{\text{train_uncentered}}$ from each row of our train and test matrices, obtaining X_{train} , X_{test} , the matrices representing our mean centered data set.



We then computed the matrix W , whose columns were the $k = 7$ most important principal components, and spanned the best-fit k dimensional subspace of our mean-centered training data. Here is what the images represented by our 5 most important principal components (“eigenfaces” as they are commonly referred to in facial image analysis) looked like:



We then used psuedoinverse W^\dagger to compute the coordinates of our mean-centered train and test data relative to the best-fit subspace by calculating $X_{\text{train_proj}} = W^\dagger X_{\text{train}}^\top$, $X_{\text{test_proj}} = W^\dagger X_{\text{test}}^\top$ ³. To visualize how

³This is the same procedure as detailed prior, as the columns of X_{train}^\top and X_{test}^\top are the mean-centered image vectors

much information we lost in this process, below are two test images reconstructed from their coordinates relative to the best-fit $k = 7$ dimensional subspace:



Ahn Bo Hyun Original



Ahn Bo Hyun Reconstructed



Harry Styles Original



Harry Styles Reconstructed

Finally, we classified our test data using the approach detailed prior. Out of our 14 test images, 4 were correctly classified, giving an accuracy of 28.57%.

4.2 Application to Larger Dataset and Hyperparameter Optimization

We repeated our experiment on a dataset of 1580 images of 158 people from the Labelled Faces in the Wild dataset [1] with a few key differences. In the pre-processing step, in addition to converting our images to grayscale, we cropped them to their central 140×165 pixels, to make the faces more prominent. We then repeated our procedure of withholding one image per person for test data, flattening images, and mean-centering to obtain X_{train} and X_{test} .

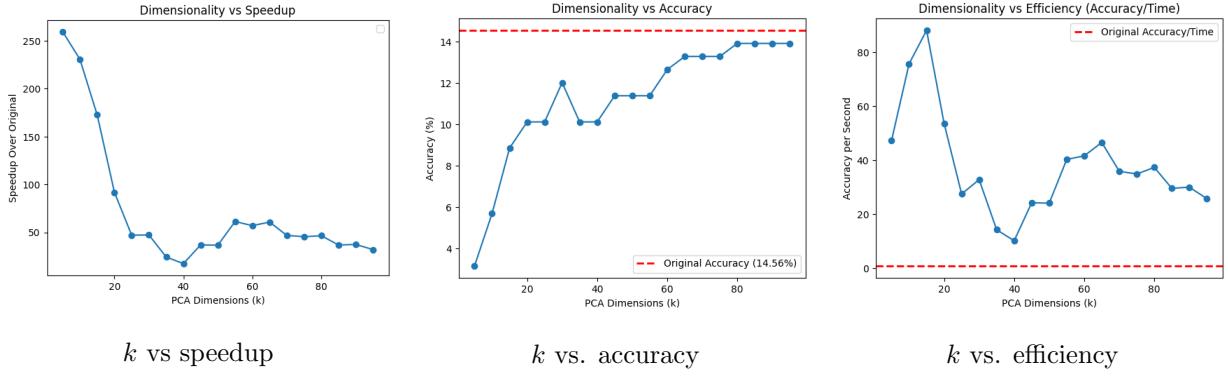
We experimented with the dimensionality k of our best-fit subspace used in the PCA. We used our prior procedure to compute W for values of k ranging from 5 to 100 and classify our test data. We also classified our test data directly against the original image space spanned by $\text{Row}(X_{\text{train, uncentered}})$ as a baseline. We measured the performance of each value k PCA against 3 metrics, defined as follows:

$$\text{speedup}(k) = \frac{\text{time elapsed running baseline approach}}{\text{time elapsed running PCA with dimensionality } k}$$

$$\text{accuracy}(k) = \frac{\text{numer of test images correctly classified by dimensionality } k \text{ PCA}}{\text{total number of test images}}$$

$$\text{effeciency}(k) = \frac{\text{accuracy}(k)}{\text{time elpased running PCA with dimensionality } k}$$

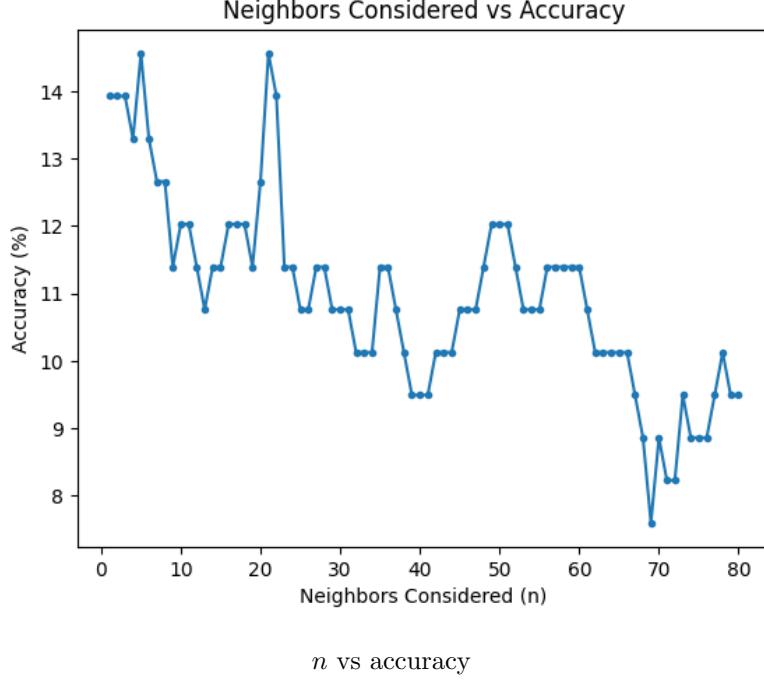
We then graphed our results:



Instead of computing the nearest neighbor of w_{test} against all training coordinates \bar{w}_i , as shown earlier, we decided to use the k -nearest-neighbors algorithm. In this method, we classify based on the “majority vote” among the n training image coordinates⁴ w_i with the smallest Euclidean distances $\|\bar{w}_{\text{test}} - \bar{w}_i\|$. In order to find the optimal value of n we ran the classification algorithm for values of n ranging from 1 to 80, measuring them against

$$\text{accuracy}(n) = \frac{\text{numer of test images correctly classified by } n \text{ nearest neighbors}}{\text{total number of test images}}$$

We then graphed our results:



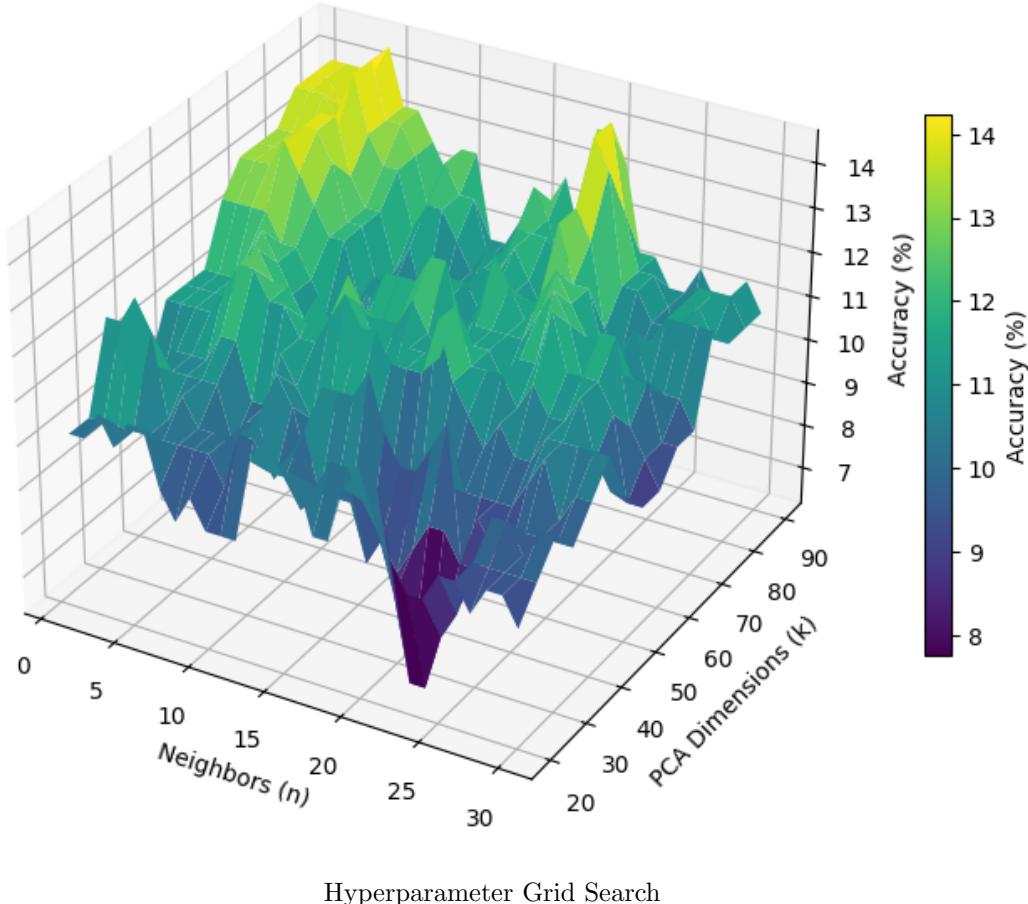
⁴We use the variable n here to avoid overloading k

Finally, to [3] concretely optimize hyper parameters n and k , we performed a grid search on the most promising ranges we observed on the above graphs: 1 to 30 neighbors and 20 to 90 dimensions, respectively, evaluating against

$$\text{accuracy}(n, k) = \frac{\text{numer of test images correctly classified by dimensionality } k \text{ PCA and } n \text{ nearest neighbors}}{\text{total number of test images}}$$

We then graphed our results:

Grid Search: Dimensions vs Neighbors vs Accuracy



Hyperparameter Grid Search

Our grid search concluded that $k = 80$ and $n = 5$ were the optimal hyper parameter values, giving a final accuracy of 14.56%. Seeing as there were 158 different possible people to classify a test image as for this dataset, compared to the 14 in the celebrity face dataset, this is a significant improvement.

5 Discussion

5.1 Accuracy and Hyperparameter Optimization

Both datasets were quite scarce, providing much fewer face photos per identity than overall identities (in LFW, this ratio was less than 10%). Thus, the PCA subspace was sparsely populated and did not have "face subregions" develop as in Turk's original Eigenfaces experiment [3], so slight changes between a training and test image for one identity could lead to a different nearest neighbors and incorrect classification. This explains the high accuracy variance between trials with different PCA dimensions, as a change in dimension could drastically change the structure of the sparse PCA space. Furthermore, it explains why considering 5 neighbors was more accurate, as this technique is more robust to slight changes in relative position of training images in the face subspace.

The optimal PCA dimension for both trials coincided with approximately half the number of identities in the training dataset. This is numerically only explained by the low values of singular values below this point (and thus nonsignificance of the associated singular vectors). Additionally, though, since later Eigenfaces begin to look like specific people, it seems like later vectors (past the first 5 – 10, which account for lighting, angle, and prominent features) specifically account for certain hard-to-classify identities.

Overall, while accuracy remains relatively similar to template matching, the efficiency increase provided by this technique is extremely promising, especially on even larger datasets. Even on the LFW dataset, template matching becomes computationally infeasible, but the PCA method maintains accuracy with orders of magnitude less computation power.

5.2 Further Improving Accuracy

In future recreations, a few techniques could be used to increase accuracy:

1. Face Alignment. With edge detection and affine transformations on the images during pre-processing, facial boundaries would be clearer and allow for better learning of features.
2. Background Removal. With similar edge detection algorithms, we could white-out the background of all training and test images, which would allow variance to focus on facial features instead of background features.
3. Data Augmentation. By synthesizing additional versions of each training face (with rotations, brightness shifts, random crops), we could have greater control over noise in test images.
4. Linear Discriminant Analysis. Rather than using PCA, which is an unsupervised technique that maximizes variance of the images themselves, we could use the labels to conduct the similar-but-supervised technique *Linear Discriminant Analysis*, which maximizes class separability rather than total variance.

5.3 Other Applications

Interestingly, this form of PCA is mathematically equivalent to common hashing-based vector-search algorithms, in which high-dimensional vectors are converted to lower-dimensional representations based on directions of high variance and stored in memory according to these in order to allow for faster search. From this perspective, it makes sense that the largest gains came in efficiency rather than accuracy—we aren't gaining new information about the images, but instead focusing on only pertinent information and allowing for more efficient template matching.

6 Code

See our code here or in Appendix A: <https://github.com/nirpechuk/eigenfaces>

References

- [1] Jessica Li. Labelled faces in the wild (lfw) dataset.
<https://www.kaggle.com/datasets/jessicali9530/lfw-dataset>.
- [2] Azhan Mohammed. Celebrities face dataset small.
<https://www.kaggle.com/datasets/sheikhazhanmohammed/celebrities-face-dataset-small>.
- [3] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991.

Appendix A: Full Code

```
# -*- coding: utf-8 -*-
"""Eigenfaces Project.ipynb

Automatically generated by Colab.

Original file is located at
https://colab.research.google.com/drive/1tP6gn5cqvUqo2887nLFjuqnCfXtZEPCH

# Eigenfaces i1 i2
## By Rohan Naphade and Nir Pechuk
Follow along to see how we can use PCA to perform unconstrained facial recognition!

# Initial Walkthrough

First, we go through the training and classification steps for a fixed matching algorithm
and dimensionality reduction, allowing us to establish the techniques of Eigenfaces.

## Step 1: Load data
"""

import kagglehub

def image_to_vector(path):
    gray = Image.open(path).convert("L")
    arr = np.array(gray, dtype=np.float32)
    return arr.flatten()

# Download our dataset
# https://www.kaggle.com/datasets/sheikhazhanmohammed/celebrities-face-dataset-small?resource=download
path = kagglehub.dataset_download("sheikhazhanmohammed/celebrities-face-dataset-small") + "/Dataset"

print(path)

from PIL import Image
import os
import numpy as np

train_images = []
train_labels = []
test_images = []
test_labels = []

folders = sorted(os.listdir(path))

for _, folder in enumerate(folders):
```

```

person = os.path.join(path, folder)
if not os.path.join(person):
    continue
for im_num, im_name in enumerate(sorted(os.listdir(person))):
    img_path = os.path.join(person, im_name)
    if (im_num == 0):
        test_images.append(image_to_vector(img_path)[:44000])
        test_labels.append(folder)
    else:
        train_images.append(image_to_vector(img_path)[:44000])
        train_labels.append(folder)

X_train_uncentered = np.vstack(train_images)
y_train = np.array(train_labels)
X_test_uncentered = np.vstack(test_images)
y_test = np.array(test_labels)

"""## Step 2: Mean-center Data

"""

m = np.mean(X_train_uncentered, axis=0)
X_train = X_train_uncentered - m
X_test = X_test_uncentered - m

"""Let's print the "average face," along with some faces before/after mean-centering!"""

import matplotlib.pyplot as plt

def show_gray_vector(vec, height=220, width=220, normalize=True):
    v = np.asarray(vec, dtype=np.float32).ravel()
    target_len = height * width

    # If truncated, pad with zeros at the end
    if v.size < target_len:
        v = np.pad(v, (0, target_len - v.size), mode='constant')
    # If somehow longer, cut off the extra tail
    elif v.size > target_len:
        v = v[:target_len]

    img = v.reshape((height, width))

    # Normalization to [0, 1] for nicer display
    if normalize:
        vmin, vmax = img.min(), img.max()
        if vmax > vmin:
            img = (img - vmin) / (vmax - vmin)

```

```

plt.figure(figsize=(1.5,1.5))
plt.imshow(img, cmap='gray')
plt.axis('off')
plt.show()

print("Average Face: ")
show_gray_vector(m);
for i in range(1): # Change if you want to see more
    print(f"{y_test[i]}'s uncentered face:")
    show_gray_vector(X_test_uncentered[i])
    print(f"{y_test[i]}'s centered face:")
    show_gray_vector(X_test[i])

"""## Step 3: Perform a PCA to find Top-K Variance Dimensions"""

U, S, Vt = np.linalg.svd(X_train, full_matrices=False)
k = 7
W = (Vt[:k, :]).T

"""What do our newly-found "eigenfaces" look like? What are the main "sources of variance" for these faces?"""

for i in range(len(W[0])):
    show_gray_vector(W[:, i])

"""## Step 4: Convert Data to Least Squares Projection Onto Subspace Using the Pseudoinverse"""

X_train_proj = np.linalg.pinv(W) @ X_train.T
X_test_proj = np.linalg.pinv(W) @ X_test.T
print(X_train_proj.shape)
print(X_test_proj.shape)

"""What have we lost? Let's try to reconstruct a face and compare! The original projection is called the *Forward Projection* and this process is *Backwards Projection.*"""

# X_train_recon = (W @ X_train_proj).T + m
X_test_recon = (W @ X_test_proj).T + m
for i in range(X_test_recon.shape[0]):
    print(f"Original for {y_test[i]}")
    show_gray_vector(X_test_uncentered[i])
    print(f"Reconstructed for {y_test[i]}")
    show_gray_vector(X_test_recon[i])

"""## Step 5: Test Accuracy by Classifying Test Data!"""

# Classify based on smallest norm and the associated label
def closest_column(x, train):

```

```

x = np.asarray(x).reshape(-1, 1)
diffs = train - x
dists = np.linalg.norm(diffs, axis=0)
return np.argmin(dists)

# Loop through test images
correct = 0
for i in range(len(X_test_proj[0])):
    classification = y_train[closest_column(X_test_proj[:, i], X_train_proj)]
    if (classification == y_test[i]):
        correct += 1
print(f'Total correct: {correct}. Total accuracy: {correct/len(X_test_proj[0])}!')

# Let's see the accuracy if we didn't use this technique
correct = 0
for i in range(len(X_test_uncentered)):
    classification = y_train[closest_column(X_test_uncentered[i].T, X_train_uncentered.T)]
    if (classification == y_test[i]):
        correct += 1
print(f'Original total correct: {correct}. Original accuracy: {correct/len(X_test_uncentered)}')

"""Hmmm, so although our new method is much more efficient (it takes less than 10% of the computation!), it doesn't show any gains in accuracy. Can we change that by optimizing our algorithm and increasing the size of our dataset?

# Hyperparameter Optimization

Let's repeat the steps, but start with a dataset much larger than before. We'll start with steps 1-3.
"""

import kagglehub
from kagglehub import KaggleDatasetAdapter
import os
import numpy as np
from PIL import Image

# Load the csv of file names and frequencies
csv_path = "lfw_allnames.csv"
df = kagglehub.dataset_load(
    KaggleDatasetAdapter.PANDAS,
    "jessicali9530/lfw-dataset",
    path=csv_path
)
# Sort the csv by number of images per face
df.sort_values(by=['images'], inplace=True, ascending=False)
# We want each face to have at least 10 samples, and we'll take only 10 from # each to ensure we don't overfit to more common faces. This yields

```

```

# 1580 images of 158 individuals.
df = df[df['images'] >= 10]

def image_to_vector(path, width, height):
    gray = Image.open(path).convert("L")
    # optimization: crop to the center to make the face more prominent
    # 150 pixels width and 200 pixels height
    w, h = gray.size
    cropped = gray.crop((
        (w - width) // 2,
        (h - height) // 2,
        (w + width) // 2,
        (h + height) // 2
    ))
    arr = np.array(cropped, dtype=np.float32)
    return arr.flatten()

height, width = 165, 140
train_images = []
train_labels = []
test_images = []
test_labels = []
data_path = kagglehub.dataset_download("jessicali9530/lfw-dataset") + \
            "/lfw-deepfneled/lfw-deepfneled/"
# Perform the same process as earlier to convert to train and test matrices
for _, row in df.iterrows():
    for i in range(1, 11):
        img_path = data_path + \
                   row['name'] + "/" + row['name'] + \
                   "_" + f"{i:04d}" + ".jpg"
        if (i == 1):
            test_images.append(image_to_vector(img_path, width, height))
            test_labels.append(row['name'])
        else:
            train_images.append(image_to_vector(img_path, width, height))
            train_labels.append(row['name'])

X_train_uncentered = np.vstack(train_images)
y_train = np.array(train_labels)
X_test_uncentered = np.vstack(test_images)
y_test = np.array(test_labels)

# Mean-center and get SVD
m = np.mean(X_train_uncentered, axis=0)
X_train = X_train_uncentered - m
X_test = X_test_uncentered - m
U, S, Vt = np.linalg.svd(X_train, full_matrices=False)

```

```

"""Now, there are two hyperparameters we can test. First, we see which dimensionality performs best.

## Dimensionality
"""

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

def closest_column(x, train):
    x = np.asarray(x).reshape(-1, 1)
    diffs = train - x
    dists = np.linalg.norm(diffs, axis=0)
    return np.argmin(dists)

k_values = []
accuracies = []
times = []
efficiency = []

# Accuracy for each k
for k in range(5, 100, 5):
    start = time.time()

    k_values.append(k)
    W = (Vt[:k, :]).T
    Wpinv = np.linalg.pinv(W)
    X_train_proj = Wpinv @ X_train.T
    X_test_proj = Wpinv @ X_test.T

    correct = 0
    for i in range(len(X_test_proj[0])):
        classification = y_train[closest_column(X_test_proj[:, i], X_train_proj)]
        if classification == y_test[i]:
            correct += 1

    acc = (correct / len(X_test_proj[0])) * 100
    accuracies.append(acc)

    end = time.time()

    elapsed = end - start
    times.append(elapsed)
    efficiency.append(acc / elapsed)

# Non-PCA Method
start = time.time()

```

```

correct = 0
for i in range(len(X_test_uncentered)):
    classification = y_train[closest_column(X_test_uncentered[i].T, X_train_uncentered.T)]
    if classification == y_test[i]:
        correct += 1

y = (correct / len(X_test_uncentered)) * 100

original_time = time.time() - start
original_efficiency = y / original_time

# Accuracy Graph
plt.plot(k_values, accuracies, marker='o')
plt.axhline(y, color='red', linestyle='--', linewidth=2,
            label=f"Original Accuracy ({y:.2f}%)")
plt.xlabel("PCA Dimensions (k)")
plt.ylabel("Accuracy (%)")
plt.title("Dimensionality vs Accuracy")
plt.legend()
plt.show()

# Speedup Graph
speedups = []
for i in range(len(times)):
    speedups.append(original_time / times[i])
plt.plot(k_values, speedups, marker='o')
plt.xlabel("PCA Dimensions (k)")
plt.ylabel("Speedup Over Original")
plt.title("Dimensionality vs Speedup")
plt.legend()
plt.show()

# Efficiency Graph
plt.plot(k_values, efficiency, marker='o')
plt.axhline(original_efficiency, color='red', linestyle='--', linewidth=2,
            label="Original Accuracy/Time")
plt.xlabel("PCA Dimensions (k)")
plt.ylabel("Accuracy per Second")
plt.title("Dimensionality vs Efficiency (Accuracy/Time)")
plt.legend()
plt.show()

"""Next, we'll fix k = 80 (the optimal dimensionality) and focus on how many neighbors
we use in our k-nearest-neighbors classification.

## Classification Algorithm
"""

```

```

from collections import defaultdict

def sorted_nearest_columns(x, train):
    x = np.asarray(x).reshape(-1,1)
    diffs = train - x
    dists = np.linalg.norm(diffs, axis=0)
    indices = np.argsort(dists)
    return indices

def majority_vote(indices, labels, k):
    votes = defaultdict(int)
    for i in range(k):
        votes[labels[indices[i]]] += 1
    return max(votes, key=votes.get)

max_considered = 80
k_values = [i for i in range(1, max_considered + 1)]
correct_count = [0] * max_considered
W = (Vt[:80, :]).T
Wpinv = np.linalg.pinv(W)
X_train_proj = Wpinv @ X_train.T
X_test_proj = Wpinv @ X_test.T

for i in range(len(X_test_proj[0])):
    for k in range(1, max_considered + 1):
        classification = majority_vote(sorted_nearest_columns(X_test_proj[:, i], \
            X_train_proj), y_train, k)
        if classification == y_test[i]:
            correct_count[k-1] += 1

accuracy = [100 * count / len(X_test_proj[0]) for count in correct_count]

# Plot values
plt.plot(k_values, accuracy, marker='.')
plt.xlabel("Neighbors Considered (n)")
plt.ylabel("Accuracy (%)")
plt.title("Neighbors Considered vs Accuracy")
plt.show()

"""## Grid Search: Putting it All Together

Let's do a grid search on the most promising range; 1-30 neighbors considered and 20-90 dimensions.

"""

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from collections import defaultdict

```

```

def sorted_nearest_columns(x, train):
    x = np.asarray(x).reshape(-1, 1)
    diffs = train - x
    dists = np.linalg.norm(diffs, axis=0)
    indices = np.argsort(dists)
    return indices

def majority_vote(indices, labels, k):
    votes = defaultdict(int)
    for i in range(k):
        votes[labels[indices[i]]] += 1
    return max(votes, key=votes.get)

dim_values = list(range(20, 91, 5))
k_values = list(range(1, 31))

accuracy_grid = np.zeros((len(dim_values), len(k_values))) # rows: k, cols: n

for di, dim in enumerate(dim_values):
    W = (Vt[:dim, :]).T
    Wpinv = np.linalg.pinv(W)
    X_train_proj = Wpinv @ X_train.T
    X_test_proj = Wpinv @ X_test.T

    sorted_indices_list = []
    for j in range(X_test_proj.shape[1]):
        sorted_indices_list.append(
            sorted_nearest_columns(X_test_proj[:, j], X_train_proj)
        )

    for ki, k in enumerate(k_values):
        correct = 0
        for j, indices in enumerate(sorted_indices_list):
            pred = majority_vote(indices, y_train, k)
            if pred == y_test[j]:
                correct += 1

        acc = correct / len(y_test) * 100.0
        accuracy_grid[di, ki] = acc

best_di, best_ki = np.unravel_index(np.argmax(accuracy_grid), accuracy_grid.shape)
best_dim = dim_values[best_di]
best_k = k_values[best_ki]
best_acc = accuracy_grid[best_di, best_ki]

print(f"Best accuracy: {best_acc:.2f}%")

```

```

print(f"Best PCA dimensionality (k): {best_dim}")
print(f"Best number of neighbors (n): {best_k}")

K, D = np.meshgrid(k_values, dim_values)

fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')

surf = ax.plot_surface(
    K, D, accuracy_grid,
    cmap='viridis',
    edgecolor='none',
    antialiased=True
)

ax.set_xlabel("Neighbors (n)")
ax.set_ylabel("PCA Dimensions (k)")
ax.set_zlabel("Accuracy (%)")
ax.set_title("Grid Search: Dimensions vs Neighbors vs Accuracy")

fig.colorbar(surf, ax=ax, shrink=0.6, label="Accuracy (%)")

plt.show()

"""## More Visualization

There's a lot more data here, so what do some of the top eigenfaces look like now?
"""

# Redefining the function just in case we're only running this section
def show_gray_vector(vec, height=220, width=220, normalize=True):
    v = np.asarray(vec, dtype=np.float32).ravel()
    target_len = height * width

    # If truncated, pad with zeros at the end
    if v.size < target_len:
        v = np.pad(v, (0, target_len - v.size), mode='constant')
    # If somehow longer, cut off the extra tail
    elif v.size > target_len:
        v = v[:target_len]

    img = v.reshape((height, width))

    # Normalization to [0, 1] for nicer display
    if normalize:
        vmin, vmax = img.min(), img.max()
        if vmax > vmin:

```

```
    img = (img - vmin) / (vmax - vmin)

    plt.figure(figsize=(1.5,1.5))
    plt.imshow(img, cmap='gray')
    plt.axis('off')
    plt.show()

# Show mean face
print("Average Face: ")
show_gray_vector(m, height, width)

# Show top 20 eigenfaces
print("Top 20 Eigenfaces: ")
for i in range(20):
    show_gray_vector(W[:, i], height, width)
```