ESE 224: Lab 10 Report

Group 6B: Syed Fehmi, Anton Idhammar, Michael Moriarty Noah Schwab

April 15, 2021

4 Creating training and test sets

4.1 Generating training and test sets

See python script 'data_import'. To create our training and test sets for facial recognition, we read in the AT&T Laboratories Cambridge data set, which consists of 10 distinct images of 40 subjects, with a total of 400 images. To create our sets, we allocated the first 9 images of each subject to the training set and the 10th image to the test set. Additionally, before appending the images to either respective set, we vectorized the images column-wise. As a result, the training set has dimensions 10304×360 and test set has dimensions 10304×40.

In generating the training and test sets, we also kept track of the 'labels' of each image in terms of which subject it corresponded to. This enabled us to quantify the success of our recognition system by comparing predicted labels with the true labels to determine the accuracy.

5 PCA on the training and test sets

For both exercise 5.1 and 5.2, see python script 'PCA.py'. This script contains a class that takes as arguments the training set, the test set, and the number of principle components k, and it contains methods to return the mean of training set, the covariance matrix of the training set, the eigenfaces of the training set (unitary matrix of eigenvectors of the covariance matrix), and the PCA transforms of both the training set and test set.

Since the parameter k determines the number of eigenvectors included in the unitary matrix (associated with k largest eigenvalues), we can track the reduction of dimensionality. Accordingly, we tested our class for four different values of k, [1, 5, 10, 20]. For each, we demonstrate that the unitary matrices have dimensions determined by their associated values of k, $10304 \times k$. Additionally, by examining the matrix multiplication, we can see that the PCA transform of the training set and test set should have dimensions $k \times 360$ and $k \times 40$, respectively.

For both the training and test set, we compute the PCA transform of each column using the following formula:

$$X_i^{PCA_k} = P_k^H (x_i - \bar{x})$$

where $\bar{x}$ is the mean of the training set and x_i is the ith vector of the training or test set, respectively.

6 Nearest neighbor classification

6.1 Nearest neighbor class

See python script 'nearest_neighbor.py'. This script contains a class that takes as arguments the PCA transform of the training set and the PCA transform of the test, and it returns predicted classification labels for each image in the test set. We include a condition that both transforms have the same number of rows, ensuring that the PCA transform of the test set was computed using the unitary matrix and mean of the training test, and the same number of principle components k was applied to both sets.

In order to classify the images of the test set, we use a nearest neighbor classification scheme. We implement this by computing the energy of the difference between a given row of the PCA transform of the test set with every single row of the PCA transform of the training set. We then determine which column of the training set corresponded to the smallest difference (i.e. the nearest neighbor) and assign the given column of the test set the label associated with the nearest neighbor in the training set. Since we computed the PCA transform on every column in the both sets, we know that the rows of the PCA transform correspond to the columns of the original set, so we assign labels accordingly. From there, it is a straight-forward method to use the indices determined by the classification to compare with the true label of the test set and determine the classification accuracy.

For the values of k [1, 5, 10, 20], we applied the classification method and computed the accuracy by dividing the number of correct predictions by the total number of test images (40). Our results are plotted in the figure below.

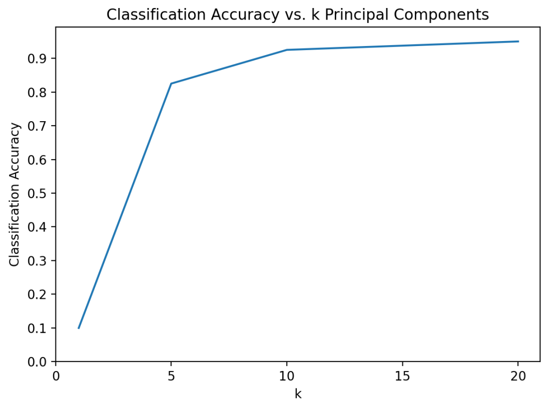

Figure 1: Classification accuracy plotted against k principle components

As we can see, 1 principle component does not yield very accurate results, but as we increase the value of k, our classification accuracy dramatically increases. At k=20, we are able to achieve an accuracy of 95%.

Below, we show select subjects from the test set and their nearest neighbors for k principle components, [1, 5, 10, 20].

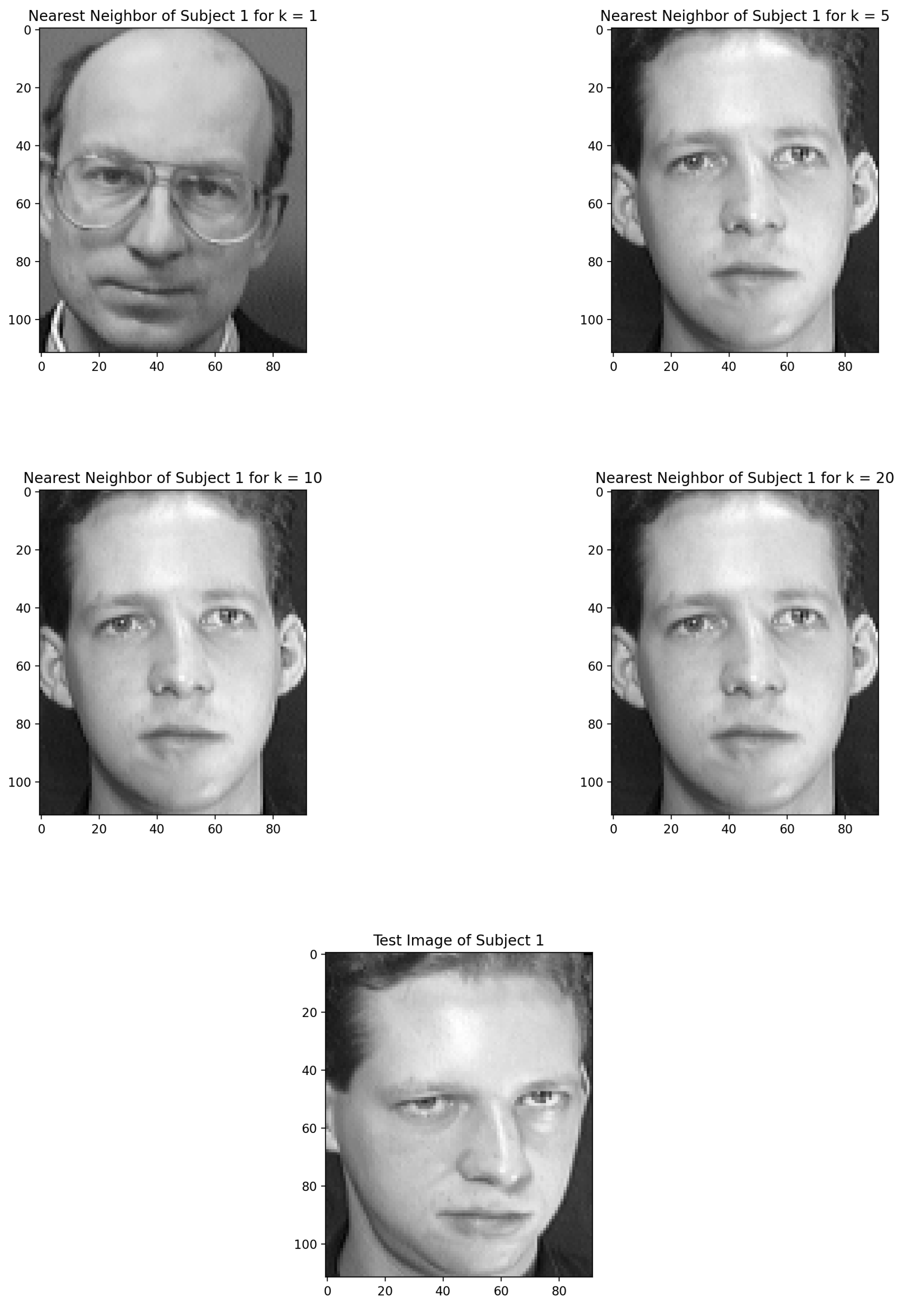
Nearest Neighbor of Subject 1 for k = 1
Nearest Neighbor of Subject 1 for k = 5
Nearest Neighbor of Subject 1 for k = 10
Nearest Neighbor of Subject 1 for k = 20
Test Image of Subject 1

Figure 2: Test image of subject 1 and nearest neighbor for different values of k

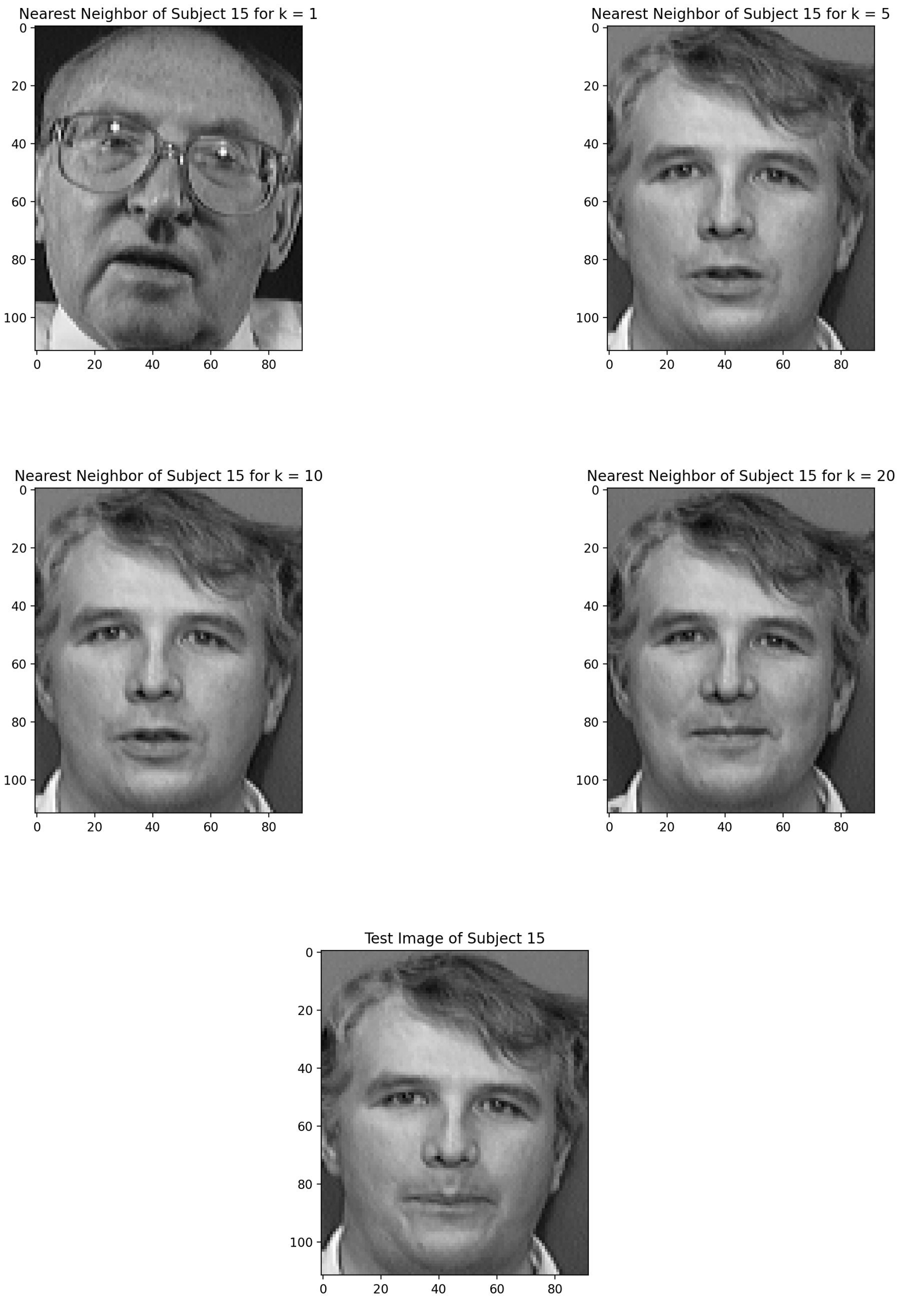
Nearest Neighbor of Subject 15 for k = 1
Nearest Neighbor of Subject 15 for k = 5
Nearest Neighbor of Subject 15 for k = 10
Nearest Neighbor of Subject 15 for k = 20
Test Image of Subject 15

Figure 3: Test image of subject 15 and nearest neighbor for different values of k

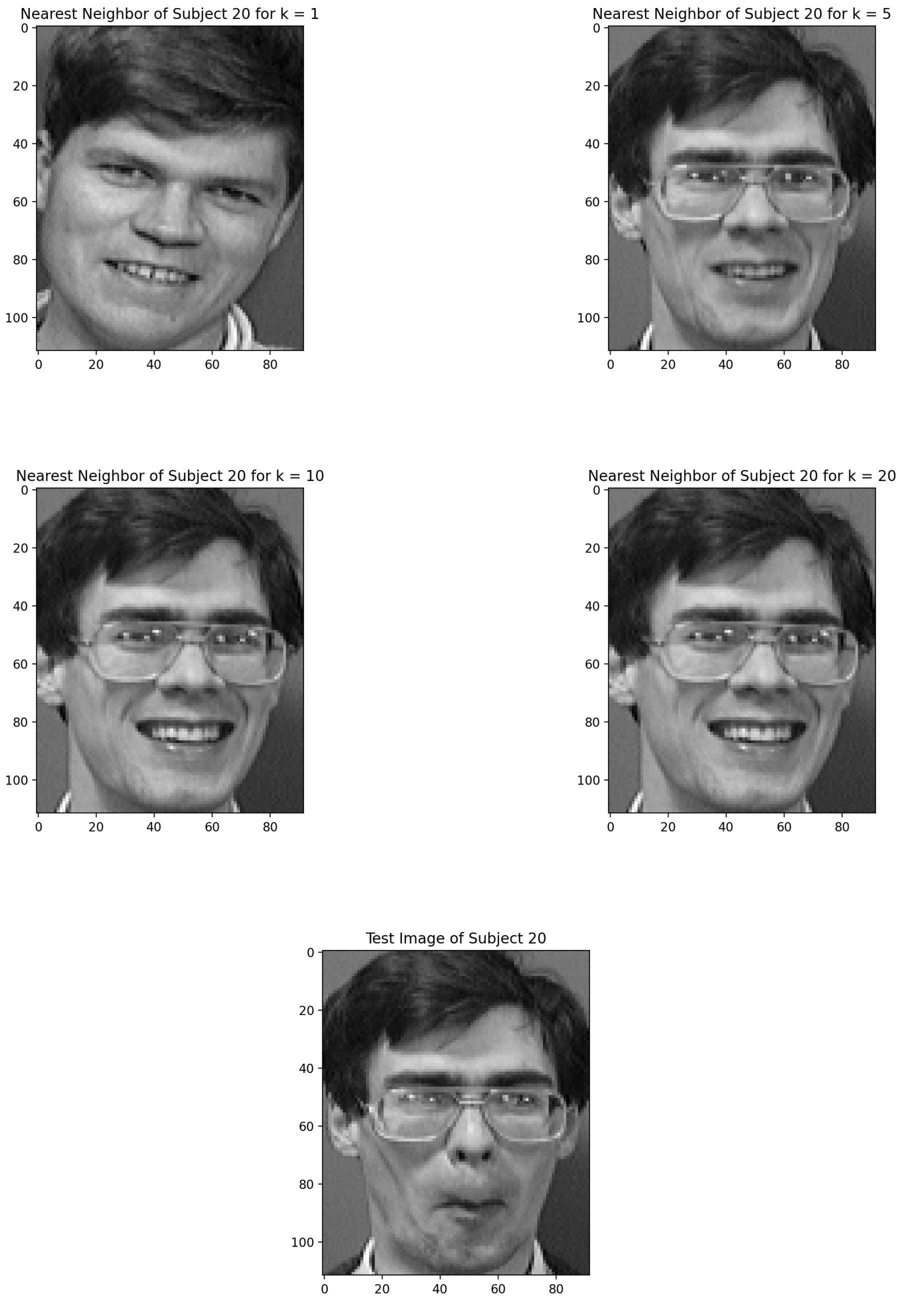
Nearest Neighbor of Subject 20 for k = 1
Nearest Neighbor of Subject 20 for k = 5
Nearest Neighbor of Subject 20 for k = 10
Nearest Neighbor of Subject 20 for k = 20
Test Image of Subject 20

Figure 4: Test image of subject 20 and nearest neighbor for different values of k

```
# This script reads in all of the images and creates a training set and testing set

import numpy as np
import matplotlib.pyplot as plt

# Exercise 4.1

# instantiate emtpy matrices for sets
training_set = np.zeros((10304, 360))   # 9 vectorized images of each of the 40 people
test_set = np.zeros((10304, 40))        # 1 vectorized image of each of the 40 people

# instantiate empty lists for labels
training_labels = np.zeros(360)
test_labels = np.zeros(40)

for i in range(1, 41):

    # reads in folder for each person
    person = "/Users/noahhschwab/PycharmProjects/ESE224_Lab11/att_faces/s" + str(i) + "/"

    # iterate through the first 9 images of a given person
    for j in range(1, 10):
        # define a given image
        image = plt.imread(person + str(j) + ".pgm")

        # vectorize the given image column-wise
        column = np.ravel(image, order='F')

        # append the vectorized image to the training set at the correct column index
        training_set[:, (i - 1) * 9 + j - 1] = column
        # append the label of the image to the label list
        training_labels[(i - 1) * 9 + j - 1] = int(i)

    # repeat the same process as above but only for the 10th image of each person for
    # the test set
    test_image = plt.imread(person + str(10) + ".pgm")
    test_column = np.ravel(test_image, order='F')
    test_set[:, i - 1] = test_column
    test_labels[i - 1] = int(i)

# save the sets as numpy files
np.save('training_set', training_set)
np.save('test_set', test_set)
np.save('training_labels', training_labels)
np.save('test_labels', test_labels)
```

```
# This script contains two classes, one that computes the PCA Transform of the training set
# and one that computes the PCA Transform of the test set

import numpy as np
import matplotlib.pyplot as plt
import scipy.sparse.linalg

# Exercise 5


class PCA():

    def __init__(self, train, test, k):

        # training set must be a numpy array with size (10304, 360)
        if not isinstance(train, np.ndarray) or train.shape != (10304, 360):
            raise ValueError("Training set must be a numpy array with size (10304, 360)")

        # test set must be a numpy array with size (10304, 40)
        if not isinstance(test, np.ndarray) or test.shape != (10304, 40):
            raise ValueError("Test set must be a numpy array with size (10304, 40)")

        # k must be a positive integer
        if not isinstance(k, int) or k < 0:
            raise ValueError("k must be a positive integer")

        self.train = train
        self.test = test
        self.k = k
        self.M = train.shape[1]        # 360 columns
        self.N = test.shape[1]        # 40 columns
        self.mu = self.mean()
        self.P = self.unitary()

    # method to compute the mean of the training set
    def mean(self):
        return self.train.mean(axis=1)

    # method to compute covariance matrix
    def covariance(self):
        return np.cov(self.train)

    # method to compute unitary matrix from covariance matrix
    def unitary(self):

        # compute the eigenvalues and eigenvectors using SciPy method
        w, v = scipy.sparse.linalg.eigs(self.covariance(), k=self.k)

        # return the first k eigenvectors
        return v

    # method to compute the transformed training set by applying the PCA transform
    # to each column
    def training_transform(self):

        # initialize empty transform matrix
        X = np.zeros(self.train.shape)

        # iterate through each column and populate the variance matrix
        for i in range(self.M):
            X[:, i] = self.train[:, i] - self.mu

        # compute the PCA transform of the training set
        P_H = np.conjugate(np.transpose(self.P))
        D = np.matmul(P_H, X)

        return D

    # method to compute the transformed test set by applying the PCA transform
    # to each column
    def test_transform(self):

        # initialize empty transform matrix
        X = np.zeros(self.test.shape)

        # iterate through each column and populate the variance matrix
        for i in range(self.N):
            X[:, i] = self.test[:, i] - self.mu

        # compute the PCA transform of test set
        P_H = np.conjugate(np.transpose(self.P))
        D = np.matmul(P_H, X)

        return D


if __name__ == "__main__":

    # Exercise 5.1
    k_list = [1, 5, 10, 20]

    # read in training set and test set
    training_set = np.load("training_set.npy")
    test_set = np.load("test_set.npy")

    # iterate through each value of k and print the unitary and transform matrices
    for k in k_list:
        object = PCA(training_set, test_set, k)
        print("-------------------------")
        print(f"k = {k}")
        print("-------------------------")
        print("Unitary Matrix:")
        print(object.unitary().shape)
        print("-------------------------")
        print("PCA Transform of Training Set")
        print(object.training_transform().shape)
        print("-------------------------")
        print("PCA Transform of Test Set")
        print(object.test_transform().shape)
```

```
# This script contains a class that classifies each image in the test set according to the nearest neighbor of its PCA
# transform from the transformed training set

import numpy as np
import matplotlib.pyplot as plt
from PCA import PCA

# Exercise 6.1


class NN():

    def __init__(self, PCA_train, PCA_test):

        # Transformed sets must have same number of rows
        if PCA_train.shape[0] != PCA_test.shape[0]:
            raise ValueError("Transformed sets must have the same number of rows")

        self.k = PCA_train.shape[0]
        self.PCA_train = PCA_train
        self.PCA_test = PCA_test

    # Nearest neighbor method to classify the columns of the test set
    def classify(self):

        # instantiate empty list of labels corresponding to training set columns
        label_list = []

        # iterate through each column of the test set
        for j in self.PCA_test.T:
            # instantiate empty list of differences
            difference_list = []

            # iterate through each column of the training set
            for i in self.PCA_train.T:
                energy_diff = np.linalg.norm(j - i) ** 2
                difference_list.append(energy_diff)

            # define the label index as minimum value of differences
            label_ind = np.argmin(difference_list)

            label_list.append(label_ind)

        return label_list


if __name__ == "__main__":
    # test our classification method for different values of k
    k_list = [1, 5, 10, 20]

    # read in training set and test set and labels
    training_set = np.load("training_set.npy")
    test_set = np.load("test_set.npy")
```

```
training_labels = np.load("training_labels.npy")
test_labels = np.load("test_labels.npy")

# instantiate empty list for accuracy of classification
accuracy_list = []

# iterate through each value of k and return the predicted labels
for k in k_list:
    object = PCA(training_set, test_set, k)
    training_transform = object.training_transform()
    test_transform = object. test_transform()

    pred_labels = NN(training_transform, test_transform).classify()

    # determine accuracy of classifications
    # append a 1 if correct, append a 0 if incorrect
    classification_list = []

    for test_ind, training_ind in enumerate(pred_labels):
        if test_labels[test_ind] == training_labels[training_ind]:
            classification_list.append(1)
        else:
            classification_list.append(0)

    # classification accuracy is mean of classification list
    accuracy = np.mean(classification_list)

    accuracy_list.append(accuracy)

    print(f"The classification accuracy for {k} principal components is {accuracy}")

# plot classification accuracy vs. k
plt.plot(k_list, accuracy_list)
plt.title("Classification Accuracy vs. k Principal Components")
plt.xlabel("k")
plt.ylabel("Classification Accuracy")
plt.xticks(np.arange(0, 21, 5))
plt.yticks(np.arange(0, 1, 0.1))
plt.show()

# iterate through different values of k for different images to show
# effect of k on facial recognition

image_list = [0, 14, 19]

for k in k_list:
    object = PCA(training_set, test_set, k)
    training_transform = object.training_transform()
    test_transform = object.test_transform()

    pred_labels = NN(training_transform, test_transform).classify()

    for i in image_list:
        plt.imshow(test_set[:, i].reshape(-1, 112).T, cmap='gray')
        plt.title(f"Test Image of Subject {int(i + 1)}")
        plt.show()

        plt.imshow(training_set[:, pred_labels[i]].reshape(-1, 112).T, cmap='gray')
        plt.title(f"Nearest Neighbor of Subject {int(i + 1)} for k = {k}")
        plt.show()
```