FACE RECOGNITION USING EIGENFACES
ELEC4630: IMAGE PROCESSING AND COMPUTER VISION

William E. G. Kvaale
University of Queensland
w.kvaale@uq.net.au
s46301303

May 15, 2020

ABSTRACT

Face recognition is a challenging task due to the similarity in human faces, when taking into account all the varying parameters. These parameters include age, health condition, view point, illumination and more. A common goal for a facial recognition system is to be unaffected to variations in these parameters. However, these variations are often greater than the variations in faces amongst humans. In this paper we will investigate a low-dimensional representation for face images, namely the *eigenface apporach* [1].

Introduction

Turk and Pentland (1991) [2] popularized the *eigenfaces* method, using *Principal Component Analysis* (PCA). PCA is a widely used data (dimensionality) reduction technique with minimal loss. Yet, it is not a data reduction technique at its core. It is a data transformation technique that sets data up to prune away components which yields less information. Which means one could extract as much information as possible, with less data.

To perform PCA, one can perform a singular value decomposition of the data matrix, or eigenvalue decomposition of the covariance matrix. In this paper we will perform the latter.

The dataset

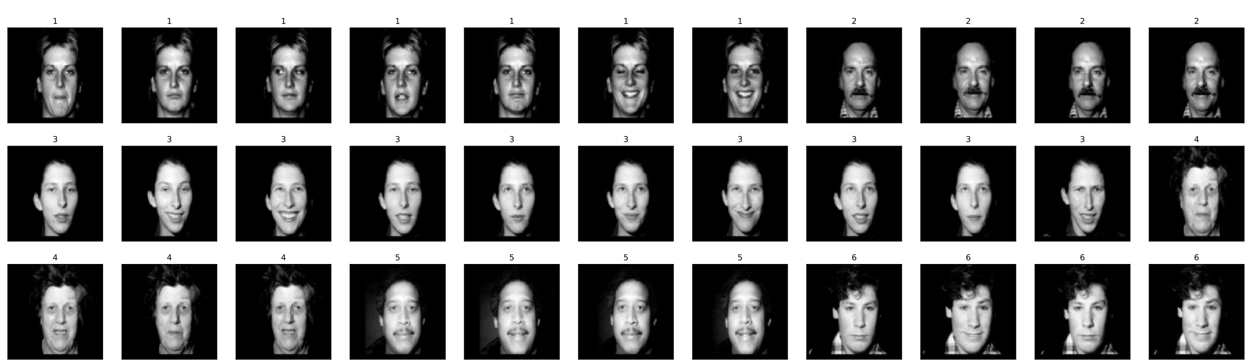

Figure 1: Complete dataset with 33 images

Here we see all 33 images of the six different faces. Some faces have more samples than others, and some have more variation in facial expression. The images are labeled with a number representing their index. The first image of each face will be used for training the model. This is the instructions given in the assignment, however this will lead to *peeking* [3, p.709].

Method

1. Reshape the image matrix from NxN to $1xN^2$
   - In our case from $128x128$ to $1x16384$
2. Subtract the average face
   - $\psi = \frac{1}{M}\sum_{k=1}^{n} I_i$, where I_i is the image set, with nxn pixels, $i \in [1, ..., M]$
   - $\Phi_i = \Gamma_i - \psi$
3. Compute the covariance matrix
   - $C = \frac{1}{M}\sum_{n=1}^{N} \phi_n \phi_n^T = DD^T$ $(N^2 x N^2 matrix)$
   - where $D = [\phi_1, \phi_2, ..., \phi_M]$ $(N^2 x M matrix)$
4. Compute the eigenvectors u_i of DD^T
   - The DD^T matrix is too large, making it impractical to compute
   - Instead, compute the eigenvectors v_i of $D^T D$
   - DD^T and $D^T D$ has the same eigenvalues and eigenvectors [4]
   - **Note**:
     - DD^T can have up to N^2 eigevalues and -vectors.
     - $D^T D$ can have up to M eigevalues and -vectors.
     - $M \ll N^2$
5. Compute the M best eigenvectors of DD^T
   - Pick only the K eigenvectors, which has the K largest eigenvalues

Results

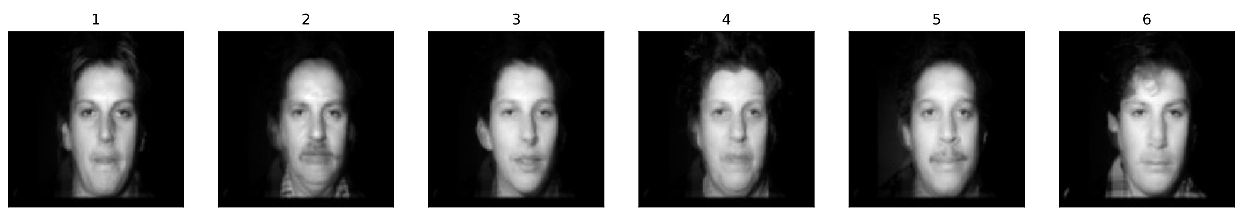

Figure 2: Training data

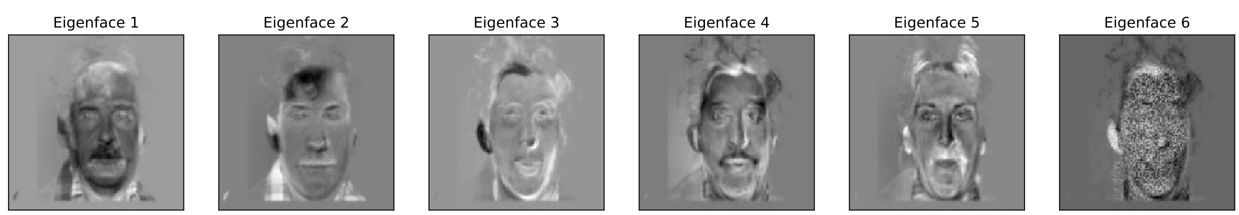

Figure 3: Eigenfaces from training data

In Figure 2 we can see the trainig data used to find the eigenfaces. Using the described method, we find the corresponding 6 eigenfaces, as seen in Figure 3. To recognize a test sample, first we need to project it onto the eigenspace. That way, we can compare the vector returned from that transformation with the eigenfaces' eigenvectors, and find the correct classification. To determine the class of a test sample, the euclidean distance between the test sampled and all $K = N_{PC}$ eigenfaces were found, and the minimum where chosen. To visualize the test sample, one needs to inversely transform it back to an NxN image, from $1xN$ vector.

In the task of using the eigenfaces to recognize faces, the results are as seen in Table 1. In this table, N_{PC} represents the number of principal components used.

The classification results can also be seen in Figure 7. In this result, all 6 principal components have been used. The results with varying K can be seen in the Appendix.

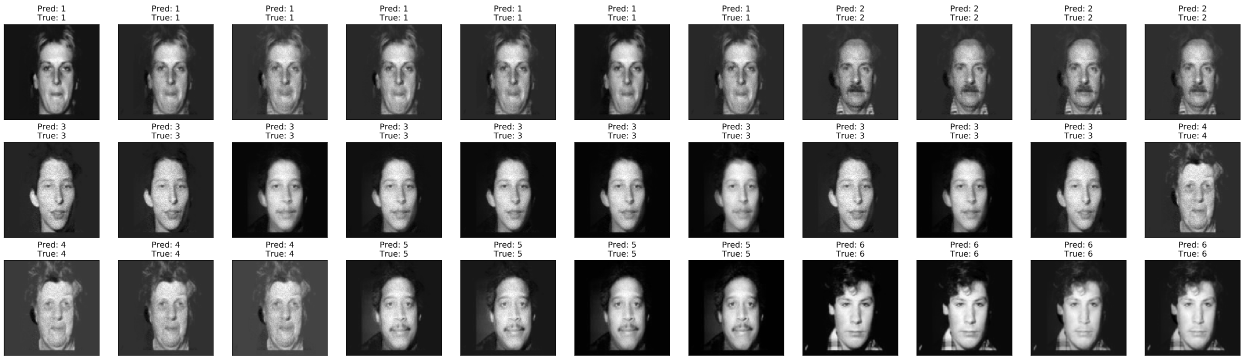

Figure 4: Predictions with N_{PC} = 6

Results		
N_{PC}	Accuracy	Correct/Total
4	75.75 %	25/33
4	81.81 %	27/33
5	96.96 %	32/33
6	100.00 %	33/33

Table 1: Results from varying N_{PC}

Discussion

In this paper we have seen that the eigenfaces method is an efficient way of extracting relevant facial information, and can be used in facial recognition systems. Due to the sparse amount of data used in assignment, the results may have been influenced by peeking. This could be investigated in future work, desirably with more data as well.

References

[1] L. Sirovich and M. Kirby. "Low-dimensional procedure for the characterization of human faces". In: *J. Opt. Soc. Am. A* 4.3 (Mar. 1987), pp. 519–524. DOI: 10.1364/JOSAA.4.000519. URL: http://josaa.osa.org/abstract.cfm?URI=josaa-4-3-519.

[2] Matthew Turk and Alex Pentland. "Eigenfaces for Recognition". In: *Journal of Cognitive Neuroscience* 3.1 (1991). PMID: 23964806, pp. 71–86. DOI: 10.1162/jocn.1991.3.1.71. eprint: https://doi.org/10.1162/jocn.1991.3.1.71. URL: https://doi.org/10.1162/jocn.1991.3.1.71.

[3] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0136042597.

[4] Brian Lovell and Shaokang Chen. "Robust Face Recognition for Data Mining". In: (Feb. 2005).

[5] Johns Hopkins University Visionlab. *Case Study PCA: Eigenfaces for Face Detection/Recognition*. URL: http://www.vision.jhu.edu/teaching/vision08/Handouts/case_study_pca1.pdf (visited on 05/07/2020).

Appendix

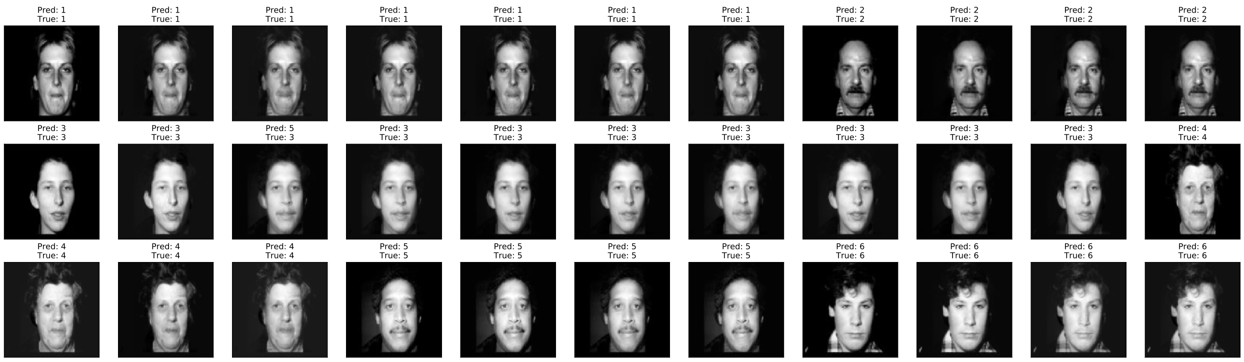

Figure 5: Predictions with N_{PC} = 5

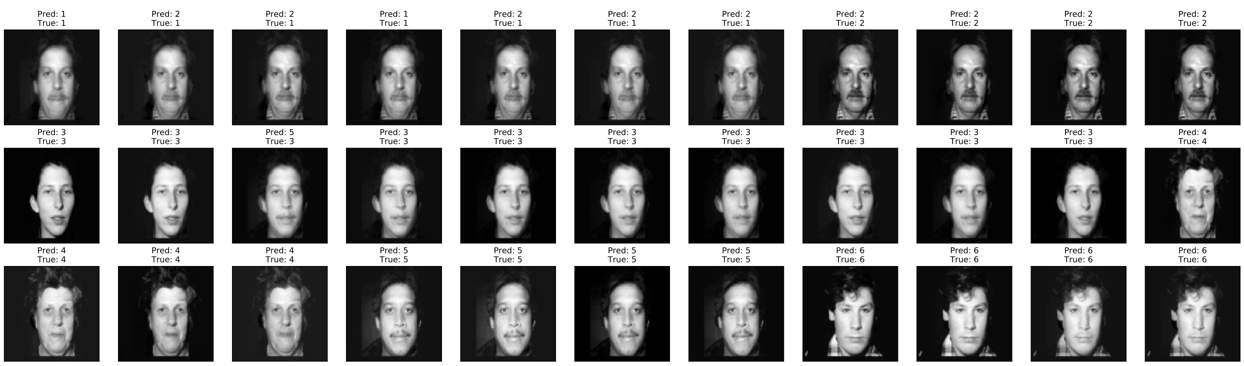

Figure 6: Predictions with N_{PC} = 4

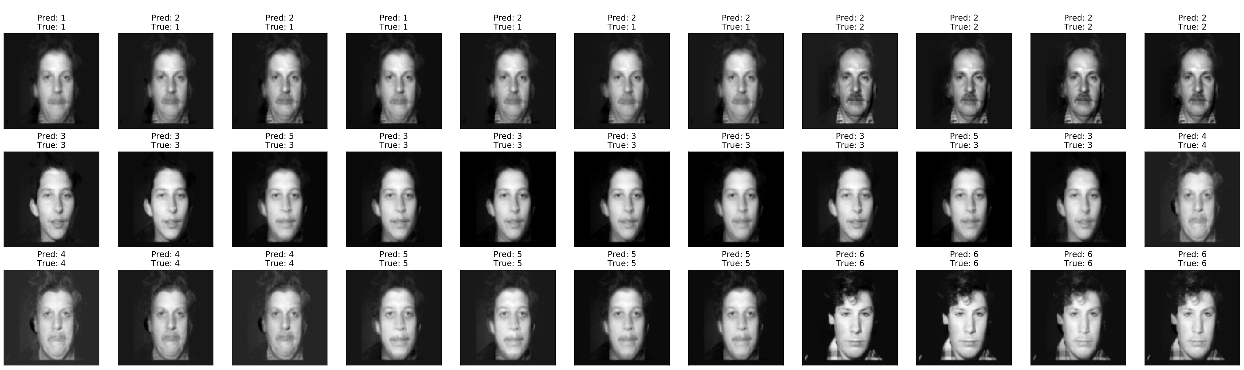

Figure 7: Predictions with N_{PC} = 3

Face Recognition using Eigenfaces (Python)

```python
import utils
import numpy as np

TRAINING = "dataset/training"
TESTING = "dataset/testing"


class PCA:
    def __init__(self):
        self.average_face = None
        self.covariance_matrix = None
        self.principal_components = None
        self.norm_factor = None
        self.img_height = 0
        self.img_width = 0

    def fit(self, X, number_of_components):
        N, H, W = X.shape
        self.img_height = H
        self.img_width = W

        X = X.reshape(N, -1)
        self.average_face = X.mean(axis=0)
        X -= self.average_face

        covariance_matrix = np.dot(X, X.T) / N
        eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)

        indices = np.argsort(eigenvalues)[::-1]
        eigenvectors, eigenvalues = eigenvectors[:, indices], eigenvalues[indices]

        eigenvectors = eigenvectors[:, :number_of_components]
        eigenvalues = eigenvalues[: number_of_components]

        compact = np.dot(X.T, eigenvectors)
        compact /= np.linalg.norm(compact, axis=0)

        self.norm_factor = np.sqrt(eigenvalues.reshape(1, -1))
        self.principal_components = compact

        return compact, eigenvectors, eigenvalues

    def transform(self, X):
        # Project data to eigenspace
        X = X.reshape(X.shape[0], -1)
        X -= self.average_face
        X = np.dot(X, self.principal_components) / self.norm_factor
```

```python
            return X

    def inverse_transform(self, X):
        # Project data back to "imagespace"
        X = np.dot(X * self.norm_factor, self.principal_components.T)
        X += self.average_face
        X = X.reshape((X.shape[0], self.img_height, self.img_width))
        return X


if __name__ == '__main__':
    data = utils.load_data(TRAINING)
    names = list(data.keys())
    X_train = np.array(list(data.values()), dtype=np.float64)
    N, H, W = X_train.shape
    #utils.plot_faces(X_train, titles=names, height=H, width=W, n_row=1, n_col=6)

    pca = PCA()
    num_of_comp = 6
    pc, e_vector, e_value = pca.fit(X_train, num_of_comp)
    rec = pca.inverse_transform(e_vector)
    #utils.plot_faces(rec, utils.get_label(names), H, W, 1, num_of_comp, save=False)

    eigenfaces = pc.T.reshape((num_of_comp, H, W))
    eigenfaces_titles = ["Eigenface %i" % e for e in range(1,N+1)]
    #utils.plot_faces(eigenfaces, eigenfaces_titles, H, W, 1, num_of_comp, save=False)

    # TESTING
    test_data = utils.load_data(TESTING)
    test_names = list(test_data.keys())
    X_test = np.array(list(test_data.values()), dtype=np.float64)
    N, H, W = X_test.shape
    #utils.plot_faces(X_test, titles=utils.get_label(test_names), height=H, width=W,
    ↪  n_row=3, n_col=11, save=False)

    transformed = pca.transform(X_test)
    reconstructed = pca.inverse_transform(transformed)
    #utils.plot_faces(reconstructed, titles=test_names, height=H, width=W, n_row=4,
    ↪  n_col=8)

    preds = utils.predict(e_vector, transformed)
    actual = utils.get_label(test_names)
    acc = utils.get_accuracy(actual, preds)
    print(f"Accuracy: {acc}%")
    utils.plot_pred(reconstructed, titles=preds, labels=actual, height=H, width=W,
    ↪  n_row=3, n_col=11, save=1)
```

Utility functions for Face Recognition using Eigenfaces (Python)

```python
import os
import matplotlib.pyplot as plt
import skimage.io as io
import numpy as np
from datetime import datetime


def plot_pred(images, titles, labels, height, width, n_row, n_col, save=False):
    NOW  = datetime.now().strftime("%m%d%Y_%H:%M%S")

    plt.figure(figsize=(3 * n_col, 3 * n_row))
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((height, width)), cmap='gray')
        plt.title(f"Pred: {titles[i]}\n"
                  f"True: {labels[i]}")
        plt.xticks(())
        plt.yticks(())
    if save:
        plt.savefig(f"output/{NOW}.pdf", bbox_inches='tight')
    plt.show()


def plot_faces(images, titles, height, width, n_row, n_col, save=False):
    NOW  = datetime.now().strftime("%m%d%Y_%H:%M%S")

    plt.figure(figsize=(3 * n_col, 3 * n_row))
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((height, width)), cmap='gray')
        plt.title(titles[i])
        plt.xticks(())
        plt.yticks(())
    if save:
        plt.savefig(f"output/{NOW}.pdf", bbox_inches='tight')
    plt.show()


def load_data(path):
    images = {}
    for root, dirs, files in sorted(os.walk(path)):
        for file in sorted(files):
            filepath = root + os.sep + file
            filename = file
            if filename.endswith(".bmp"):
                im = io.imread(filepath, as_gray=True)
                images[filename] = im
```

```
    return images


def get_label(labels):
    out = []
    for lbl in labels:
        out.append(lbl[:1])
    return out


def predict(train, test):
    clf = []
    for i, pred in enumerate(test):
        image_index = check_distance(pred, train)
        clf.append(image_index + 1)
    return clf


def check_distance(test_vector, all_weight_vectors):
    dst = {}
    for i, candidate in enumerate(all_weight_vectors):
        dst[i] = np.linalg.norm(test_vector - candidate)

    dst_values = list(dst.values())
    lowest_dst = np.min(dst_values)
    index = dst_values.index(lowest_dst)
    return index


def get_accuracy(y, y_pred):
    hit = 0
    for i, pred in enumerate(y_pred):
        print(f"Predicted: {int(pred)} \t Actual: {int(y[i])}")
        if int(pred) == int(y[i]):
            hit += 1
    print(f"Score: {hit}/{len(y_pred)}")
    return hit/len(y_pred)*100
```