Facial Recognition on Dimensionally-Reduced Datasets Using Principal Component Analysis and Linear Discriminant Analysis Dimensionality Reduction Methods

Amr Mustafa — Mostafa Mahmoud

***Abstract*— Our goal is to perform facial recognition on a dimensionally-reduced dataset using principal component analysis (PCA) and linear discriminant analysis (LDA) dimensionality reduction methods. This report illustrates the performance of a simple classifier given a dataset preprocessed by the aforementioned methods.**

I. INTRODUCTION

We intend to perform facial recognition, i.e. for a given image our algorithm should be able to tell the subject label. We will be using AT&T's database of faces which consists of 40 subjects, each having 10 images, giving us 400 samples to train and test our classifier with.

II. GENERATING THE DATA MATRIX AND THE LABEL VECTOR

A. FORMAT

The dataset has 10 images per 40 subjects. Every image is a grayscale image of size 92x112 pixels. The images are stored in PGM file format. A PGM file is a grayscale image file saved in the portable gray map (PGM) format and encoded with one or two bytes (8 or 16 bits) per pixel. It contains header information and a grid of numbers that represent different shades of gray from black (0) to white (up to 65,536). In order to read the images in the specified format, we used the Python Imaging Library (PIL).

B. DATA MATRIX

To generate the data matrix, we need to convert each of the 400 images of size 92x112 pixels into a vector of 10304 dimensions corresponding to the individual pixels. A straightforward way to do that is by stacking the rows of the image vertically on top of each other resulting in the desired vector. This can either be done manually or using Numpy's reshape method as shown in the snippet below.

```
# Convert each image into a
    10304-dimensional vector.
vec = np.array(img).reshape(10304)
```

After obtaining the vector representaion of each image in the dataset, we move on to the next step which is to stack the 400 vectors into a single data matrix D.

```
# Generate the data matrix D.
D = np.array(dataset)
D = D.reshape((400, 10304))
```

C. LABEL VECTOR

To form the label vector for our data we made use of the way we accessed the dataset to generate the data matrix. That data matrix was constructed in such a way that all the samples with label i appear in the data matrix before all the samples with label $i + 1$.

```
# Generate the label vector y.
labels = []

for i in range(1, 41):
  labels.append(i * np.ones((10, 1)))

y = np.stack(labels, axis=0).reshape(400,
    1)
```

III. SPLITTING THE DATASET

In splitting the dataset into training and test sets we opted for a 50-50 split such that the odd rows were kept for training, and the even rows were used for testing. This gave us 5 instances per person for training and 5 instances also per person for testing. We also split the label vector accordingly.

```
# Keep the odd rows for training.
X_train = D[::2]
y_train = y[::2]

# And the even rows for testing.
X_test = D[1::2]
y_test = y[1::2]
```

IV. CLASSIFICATION USING LDA

A. DIMENSIONALITY REDUCTION

LDA is a dimensionality reduction method that seeks to reduce the dimensionality of the data while preserving as much of the class discriminatory information as possible. The output of the LDA algorithm is a projection matrix U that is used to project the dataset into the computed low-dimensional subspace.

The pseudocode for 2-class LDA can be easily modified to be generalized into k-class LDA, for example, instead of using the single most dominant eigenvector, we use the 39 most dominant eigenvectors because we have 40 classes in our case.

After computing the projection matrix U, we project the training set and the test set separately as shown below.

```
# Project the training set.
projection_train = X_train.dot(U)

# Project the testing set.
projection_test = X_test.dot(U)
```

B. CLASSIFICATION USING THE 1-NN CLASSIFIER

We used the 1-NN (first nearest neighbor) classifier to classify the data points in our dataset after performing LDA on the original data. This resulted in an accuracy of 94% on the face recognition dataset.

```
# Use a first Nearest Neighbor classifier
    to classify the data.
classifier =
    KNeighborsClassifier(n_neighbors=1)
classifier.fit(projection_train, y_train)
y_pred =
    classifier.predict(projection_test)
print(metrics.accuracy_score(y_test,
    y_pred))
```

C. CLASSIFIER TUNING

To find the optimal value of the hyperparameter k of the kNN classifier, we evaluated the classification at different values of k and plotted the results below. It is obvious that as the number of neighbours is increased, the accuracy of the classification is reduced.

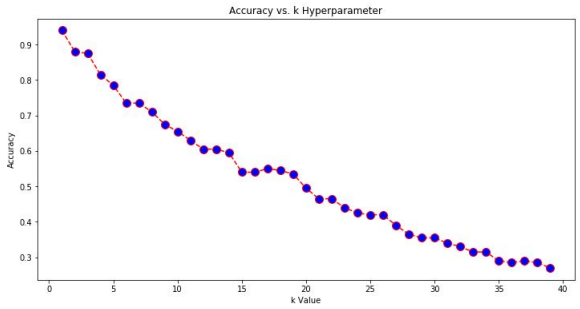

Fig. 1. Accuracy vs. k Value

V. CLASSIFICATION USING PCA

A. DIMENSIONALITY REDUCTION

Similar to linear discriminant analysis, the principal component analysis method aims to reduce the dimensionality of the given data. However, PCA is an unsupervised method that does not make use of any class discriminatory information, instead it aims to find the directions of maximum variance in high-dimensional data and projects it onto a new subspace usually with fewer dimensions.

The number of new dimensions (principal components) depends upon the fraction of total variance α we want to be captured in the projected dataset. The following plot shows the explained variance ratio plotted against the number of principal components.

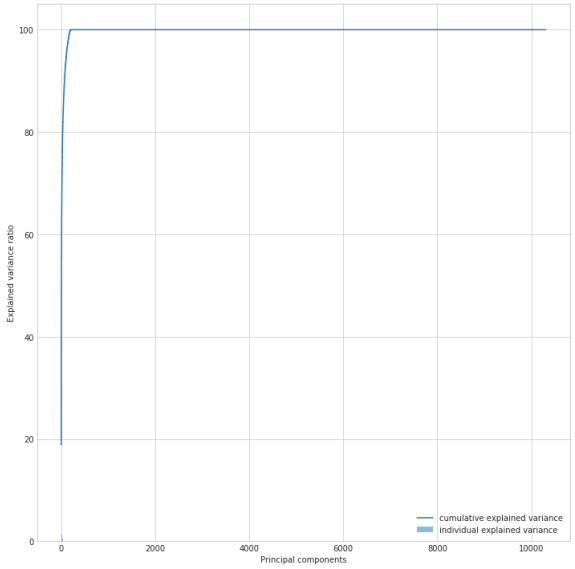

Fig. 2. Explained Variance Ratio vs. Number of Principal Components

B. CLASSIFICATION USING THE 1-NN CLASSIFIER

Classifying the face recognition dataset using the first nearest neighbor classifier resulted in 80% accuracy.

C. CLASSIFIER TUNING

Similarly to the LDA case, to find the optimal value of the hyperparameter k of the kNN classifier, we evaluated the classification at different values of k and plotted the results below. It is obvious that as the number of neighbours is increased, the error of the classification is increased.

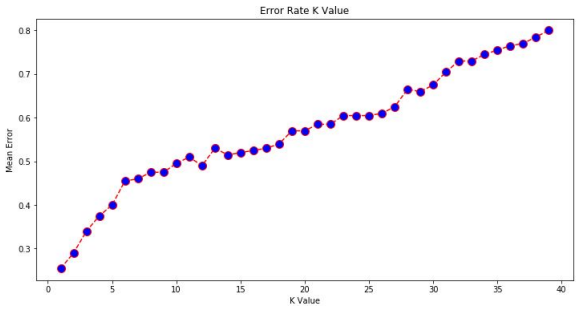

Fig. 3. Mean Error vs. k Value

REFERENCES

[1] Raschka, Sebastian. Python Machine Learning, 1st ed.: Packt Publishing, 2015.