Laboratory Experiment #6

Face Recognition: Feature Extraction and Classification

1 Introduction

Face recognition includes the following steps: face detection within the frame, face image processing such as image enhancement, feature extraction (the result is usually a feature vector that represents the face 'template'), and classification of 'templates' to the classes representing someone's identity. The classifier must be trained first on the training, also called gallery, images. The simplest classification would be 'match' or 'no-match' of the probe image 'template', against the gallery 'templates'. After classification, we evaluate the classifier performance using several measurements such as True Positive rate (TPR), True Negative Rate (TNR), False Positive Rate (FPR) and False Negative Rate (FNR), and also illustrate using a Confusion matrix.

The purpose of this lab exercise is to investigate the classical approach to **face feature extraction** based on the Principal Component Analysis (PCA), threshold-based **face matching** using Euclidean Distance (ED), and also multi-class classification using PCA + KNN.

The dataset used in this Lab, is the AT&T (also called ORL) data set, but you can add your own photos or create your own dataset. We will use Python and Jupyter Notebook (file `Lab06-FaceRec.ipynb`).

1.1 AT&T Faces Database

In this Lab, we are going to use a public dataset called "AT&T Database of Faces", which is available on the internet[1]. This dataset is composed of 400 cropped grey-scale images of 40 subjects, each subject is represented by 10 pictures, in `.pgm` format. Fig. 1 contains some images from the dataset. Note that these images are already cropped, and, thus, face detection is not required. In "the wild", the subjects are often pictured in a complex scene or background, and the face must be detected before the face recognition is attempted.

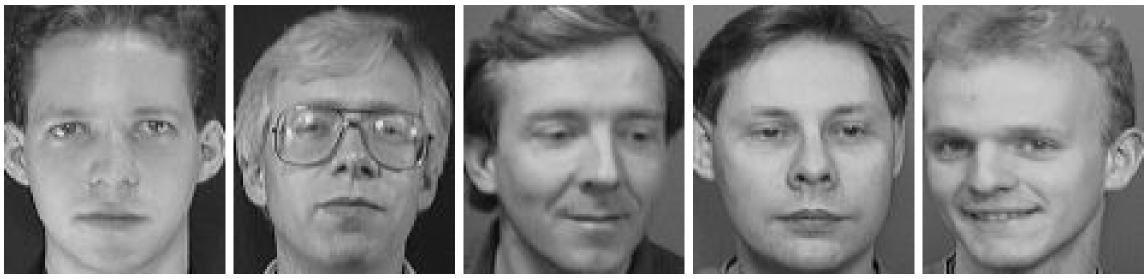


Figure 1: Samples of the AT&T Faces dataset.

The D2L section of this lab contains the AT&T dataset in a single zip-file: `ATT dataset.zip`. After the download, place it into the same directory as your Jupyter notebook `Lab06-FaceRec.ipynb`. The directory `ATT dataset/` has the following internal structure:

```
ATT dataset/
    s1/
        1.pgm, 2.pgm, ..., 10.pgm
    s2/
        1.pgm, 2.pgm, ..., 10.pgm
    ...
    s40/
        1.pgm, 2.pgm, ..., 10.pgm
```

[1] https://www.kaggle.com/kasikrit/att-database-of-faces

Folders `s1` to `s40` contain the face images of each of the 40 subjects. Inside each of these folders, there are 10 files (`1.pgm`, ..., `10.pgm`) that are different images of the same subject.

Note that you can take photos of yourself and your colleagues to create a similar dataset of two or more subjects, we recommend to have 10 images per person. Your images shall be cropped to have the same size, similarly to the provided dataset. They can be color images, and will be converted to gray-scale by the code provided. When submitting this lab notebook via D2L, **do not upload your photos**.

2 The laboratory procedure

This procedure requires to enroll samples in order to create a dataset (a gallery), and the face recognition, or matching a probe face template against each of the gallery templates, one by one. The face image handling for both the enrollment and the matching is composed of two parts:

1. Face feature extraction: the face images will be presented by vectors or matrices called 'eigenfaces' which are the result of applying a PCA; this term is similar to the term 'eigenvalues' in matrix transforms.
2. Matching the probe (test) face vectors against the gallery ones; the similarity between the feature vectors for the match or non-match decision is based on the Euclidean distance between vectors, and the determined threshold (a distance value that separates the 'genuine' and 'impostor' data).
3. In this lab, besides the Euclidean distance to compare a probe image against the gallery images, we will investigate a common classifier used to classify multiple classes, called the K-Nearest Neighbors (KNN).

2.1 Acquisition and loading the sample data

For this Lab, you have the provided a Jupyter notebook (`Lab06-FaceRec.ipynb`) and a dataset of faces, the AT&T Faces dataset.

To load the images, we will use a Python library which is also included in your Anaconda installation: Scikit-Image (https://scikit-image.org/). This library provides a high level of abstraction in several task related to images, for example, loading. The following code loads an image provided in `.pgm` format, but other formats (for example, `.png` or `.jpg`) can be read in the same way:

```python
# loading  one image to get the dimensions
# you can use images .jpg and .ng as well
img = imread(path + '1.pgm', as_gray=True)
```

Note that in this exercise, all the images should be converted to the gray-scale for further feature extraction. The images in AT&T dataset are already gray-scale. If you decide to use your own photos, they can be color ones, and the argument `as_gray=True` in the `imread` command above will automatically convert your color image into a gray-scale.

To start a dataset (a gallery), we are going to use 9 images/faces of the one person, subject 1. This is a gallery of multiple images of the same person (we call it 'intra-class'). Remember that the AT&T dataset has 10 images of each subject with various head positions; we will keep on photo to be used as a probe. The code below will load the first 9 images and store them in the variable `S`:

```python
# allocation of vector that will have all images
S = np.zeros((irow*icol, M)) # img matrix

plt.figure(figsize=(8,8))
for i in range(1,M+1):
    img = imread(path + '{}.pgm'.format(i), as_gray=True)

    plt.subplot(3,3,i)
    plt.imshow(img, cmap='gray')
    plt.axis('off')

    # reshape(img',irow*icol,1);
    # creates a (N1*N2)x1 vector
    temp = np.reshape(img, (irow*icol,1))
    S[:,i-1] = temp[:,0]
```

In the image processing, it is often necessary to "normalize" the images, that is, have the pixel intensity distributed equally in all the images in the dataset. In this exercise, we will represent all images using 256 levels of gray, with the mean pixel intensity of 100 and the standard deviation of 80:

```
# normalization  parameters, mean and standard deviation
um = 100
ustd = 80

# going over all the loaded images in the dataset S to normalize
for i in range(S.shape[1]):
    temp = S[:,i]
    m = np.mean(temp)
    st = np.std(temp)
    # calculation to define the new pixels intensities
    S[:,i] = (temp - m) * ustd / st + um
```

2.2 Face Feature Extraction

For the feature extraction required for further classification, we are going to use Principal Component Analysis (PCA). This assumes a calculation of an 'average' face across the faces in the dataset, or gallery. The difference between this average face (vector) and every image (vector) in the gallery, shows the degree of how the gallery faces are different from the average face and which face conveys the most information (about how unique a face is compared to the average face). We use this information to calculate vectors called 'eigenvectors' or 'eigenfaces' for this set of training images. The vectors that vary the most are called 'principal components', hence, PCA. The 'eigenfaces' for each image in the set are used to calculate weights which describe how much each 'eigenface' contributes to that face image. These weights are 'templates' of the gallery faces (gallery's feature vectors).

At the classification phase, once a probe face is submitted, a feature vector of the probe face is calculated. Then a difference between the input vector and the mean image vector of the gallery is computed, and transformed into a weight vector. This 'probe' weight vector is compared against the gallery's 'weight' vector, using the Euclidean distance.

In our case, we have a training, or gallery set of 9 images. Those are of the same subject, but for the algorithm, there is no difference if they come from one or multiple subjects. For now, it is 9 images (of one subject). These images are a training set. In the further exercise or this lab, we will have more subjects, each represented by few images.

To extract features on the set of 9 images, we calculate the 'average' image followed by the covariance matrix calculation, and the decomposition of the image based on the eigenvalues and eigenvector:

```
dbx = S.copy()
A = dbx.T
L = np.matmul(A, A.T)

# vv is the eigenvector for L
# dd is the eigenvalue for  L = dbx.T*dbx
dd, vv = la.eig(L)
```

The resulting set of the eigenvalues and eigenvectors is used to represent a 'weight' vectors of all 9 images.

2.3 Classification of eigenvectors using threshold

Consider a testing, or probe image, to be evaluated against the face images in the gallery. The probe image submitted for the matching, requires a normalization, and then is converted to an eigenvector. For this experiment, we are going to use as a test, one last image of subject 1. Since we used 9 images of subject 1 to create the gallery dataset, we will use the 10th image for testing. The processing of the new image is implemented by the code below:

```
# using the last image of subject #1
path = './ATT dataset/s1/'
InputImage = imread(path + '10.pgm', as_gray=True)

# Normalization
temp = InImage
me = np.mean(temp)
st = np.std(temp)
temp = (temp—me) * ustd/st + um
NormImage = temp

# Finding the Difference: NormalizedImage — MeanImage
Difference = temp — m

InImWeight = np.zeros((M,1))
for i in range(u.shape[1]):
    t = u[:,i].reshape(—1,1).T
    # scalar vector product of vectors t and Difference
    WeightOfInputImage = np.dot(t, Difference)
    InImWeight[i] = WeightOfInputImage
```

The matching procedure is then performed between the probe and each one of the gallery templates (for example, 9), and every time a similarity measure, Euclidean distance, is calculated. In the code below, the feature ('weight') vector of the probe image (`InImWeight`) is compared against the gallery of features vectors. The calculated distances are stored into the variable `eSameSubject`:

```
eSameSubject = np.zeros((M,1))

for i in range(omega.shape[1]):
    q = omega[:,i].reshape(—1,1)
    DiffWeight = InImWeight — q
    mag = np.linalg.norm(DiffWeight)
    eSameSubject[i] = mag

# max/min Euclidean distance
MaximumValue = np.max(eSameSubject)
MinimumValue = np.min(eSameSubject)
```

Figure 2 shows several comparisons with different images based on what was described before. Fig. 2a shows the probe image of the same subject that is known to dataset. Fig. 2b shows the probed face which is not known to the gallery. Fig. 2c has an input image that is not a face. For the three cases, as shown in Fig. 2, the bar plots shows how different are the Euclidean distances for each case. Clearly, the Euclidean distance is smaller (shows more similarity) for the images of the same person, and greater for the image of a different person and for a not-a-face image.

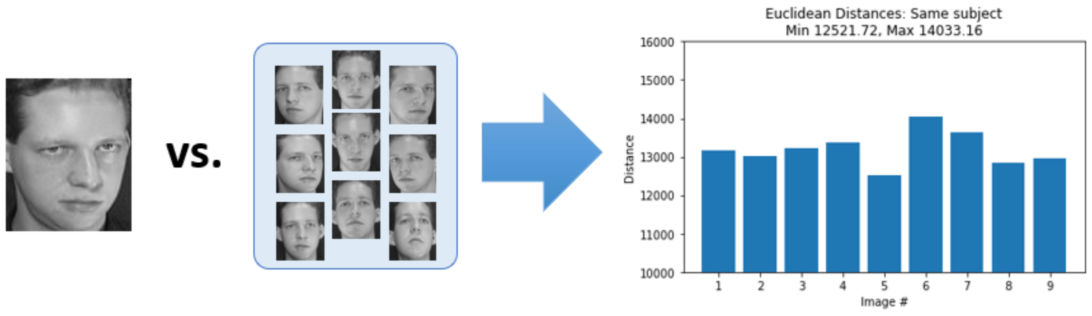
The first criterion for the best match out of many (either for 'intra-class' of the same person, or 'inter-classes', of different people) is that the Euclidean distance must be minimal.

There must be two thresholds established by experimental way. If the Euclidean distance is below a certain 'insider' threshold, the subject is known to the gallery, and the lowest Euclidean distance is the best match. If the Euclidean distance is above the 'insider' threshold but below a determined 'outsider' threshold, the face is unknown to the gallery. If the Euclidean distance is above the 'outsider' threshold, the image is not a face. Below we show some sample thresholds based on the previous analysis:

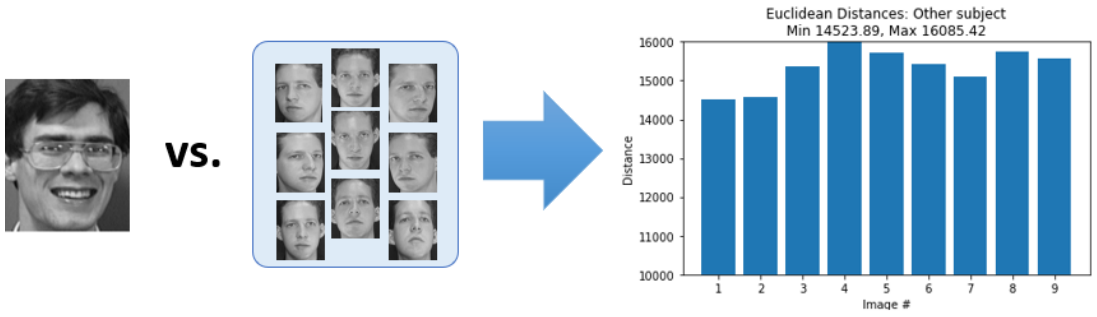
```
# Set Threshold Values
threshold1 = 14000
threshold2 = 15500

mean_of_distances = np.mean(e);
```

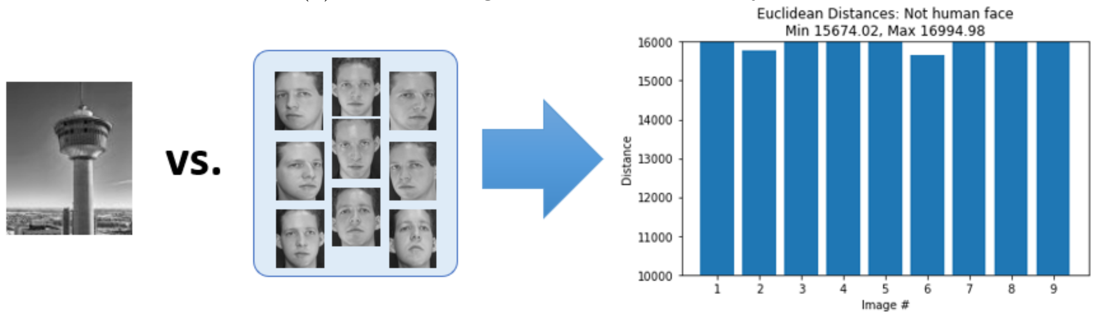
```
if (mean_of_distances <= threshold1):
    print('Image is in the database')
elif (mean_of_distances > threshold1 and mean_of_distances <= threshold2):
    print('Image is a face but not in the database')
elif (mean_of_distances > threshold2):
    print('Image is not a face')
```



(a) The new image is of the same subject in the dataset.



(b) The new image is from a different subject.



(c) The new image is not a human face.

Figure 2: Comparisons of different probe images against the template dataset of 9 images. The Euclidean distances between the probe image's feature vector and the vectors in the dataset are shown using the bar plots.

In this Lab, the two thresholds must be found experimentally. To find the 'insider' threshold, for example, analyze the maximum and minimum Euclidean distances for the probe that is known to belong to the database.

2.4 Multi-class classification using KNN

In this lab, we will use the features extracted and reduced using PCA, and then apply a classifier called K-Nearest Neighbors (KNN), instead of Euclidean distance such as in a simple matching.

For classification, the following parameters need to be defined:

- `n_subjects`: this variable corresponds to the quantity of subjects that will be used for the following tests. In the provided Jupyter notebook we set it at 20. Given the AT&T dataset, this number could be up to 40.

- `n_training_images` and `n_test_images`: those variables are complementary. To train the KNN classifier, the first `n_training_images` of each subject will be used. Later on, after the training, the remaining `n_test_images` are used to evaluate the classifier. Since the AT&T dataset has 10 images per subject, the sum of those two variables should not exceed 10.
- `knn_neighbors`: to make the decision about assigning a class to a probe sample, the KNN classifier analyzes the first `K` neighbors of such sample. Here we define it as 3 but some other odd values (such as 5, 7, 9, 11) are commonly used as well.

2.4.1 Principal Component Analysis (PCA)

In this exercise, we will use the PCA already implemented in Scikit-Learn library[2]. This PCA implementation requires a parameter called `n_components`, which represents how many principal components will be used. The code below shows how to call the PCA method in Python considering the 100 principal components:

```
# n_components: number of principal components
pca = PCA(n_components=100)

# fit the model, e.g., creating the covariance matrix... as done manually in Lab 4
pca.fit(trainingFaces);
```

The method `fit(...)` needs to be called in order to allow the PCA to *learn* from your data. In this example, the "data" are the training faces stored in the variable `trainingFaces`. The explanation of how `trainingFaces` is included in the next section.

After calling `fit(...)`, the PCA has already learnt the data, however the data used for training was not yet mapped into the Eigenfaces domain. To do this "mapping", we call the `transform(...)` function, as shown below:

```
train_pca = pca.transform(trainingFaces)
```

Next, we will use the remaining images (not used for training) as testing images. For illustration, we can plot the Eigenfaces. For example, the first 30 Eignfaces are shown in Fig. 3:

```
fig = plt.figure(figsize=(16, 6))
for i in range(30):
    ax = fig.add_subplot(3, 10, i + 1, xticks=[], yticks=[])
    ax.imshow(pca.components_[i].reshape(img.shape))
```

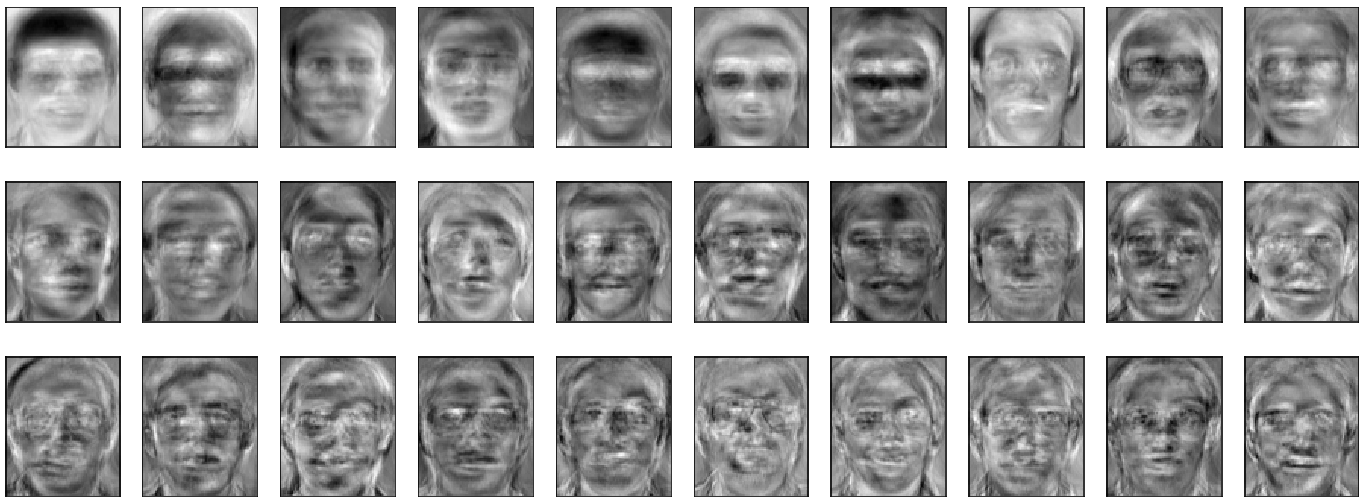


Figure 3: The first 30 Eigenfaces calculated by the PCA.

[2]https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

2.4.2 Training and testing sets

In this section, we describe how to load all the images necessary for the training and testing sets using PCA. All the images are rearranged into a vector and added to the corresponding lists:

```
# use the first 5 images from subjectID 1—n_subjects to train the classifier
trainingFaces = []
trainingLabels = []
for subjectId in range(1, n_subjects+1):
    for imageId in range(1, n_training_images+1):
        img = imread("ATT dataset/s%d/%d.pgm" %(subjectId, imageId), as_gray=True)
        trainingFaces.append(np.reshape(img, (img.size,)))
        trainingLabels.append(subjectId)

# use the last 5 images from subjectID 1—20 to test the classifier
testingFaces = []
testingLabels = []
for subjectId in range(1, n_subjects+1):
    for imageId in range(n_training_images+1, n_training_images+n_test_images+1):
        img = imread("ATT dataset/s%d/%d.pgm" %(subjectId, imageId), as_gray=True)
        testingFaces.append(np.reshape(img, (img.size,)))
        testingLabels.append(subjectId)
```

Now, we “train” the PCA using the variable `trainingFaces` :

```
pca = PCA(n_components=100)
pca.fit(trainingFaces);
```

The next step is to apply the PCA tuned to the sets of images for training or testing:

```
train_pca = pca.transform(trainingFaces)
test_pca = pca.transform(testingFaces)
```

After this procedure, the data is ready to be used to train the classifier.

NOTE: For the classifier training and for its testing, we have divided the dataset into the training and testing sets. To avoid any bias in splitting the sets, a technique called *K-Fold cross-validation* is used. It includes:

- Separate the database into K groups;
- Train the recognition system using $K - 1$ groups;
- Test the recognition system using the remaining group;
- Record the performance (accuracy) of the test results;
- Iterate through all K groups such that every group has been tested;
- Obtain the final result by averaging all K performances.

In this Lab, we will not use this technique, but you may need to use it in your project.

2.4.3 Classification with KNN

The KNN implementation is available in Scikit-Learn library[3]. The parameters include the number of neighbors (defined previously in the variable `knn_neighbors`), the variable containing the samples used for training (for example `train_pca`) and the corresponding labels (stored in the variable `trainingLabels`).

```
# n_neighbors: number of neighbors to use
knn = KNeighborsClassifier(n_neighbors=knn_neighbors).fit(train_pca,
                                                           trainingLabels)
```

[3]https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

2.4.4 Classifier evaluation

After the classifier is trained, we now can test the classifier using the probe face images. We will use the *testing* image. The classifier shall produce, or 'predict' a class (label):

```
predictedLabels = knn.predict(test_pca)
```

Note that the only argument of the `predict(...)` function is the list of "faces" in the *test* set. Here, we do not use the labels. The predicted labels are stored in the variable `predictedLabels`. Using these predicted labels (variable `predictedLabels`) and the original ones (variable `testingLabels`), we can analyze how many were predicted correctly.

To evaluate the classifier performance, we count how many true/false matches or non-matches were produced by the classifier. In this Lab's notebook, we have created a Python function called `prediction_evaluation`, which compares both labels (predicted and original) and show the results based on several metrics. You can call it using the command below, and pass both the label vectors and the ID of to subject for whom you want to evaluate the classifier performance. Let us consider, for example, subject with ID=1:

```
prediction_evaluation(predictedLabels, testingLabels, subject_id=1)
```

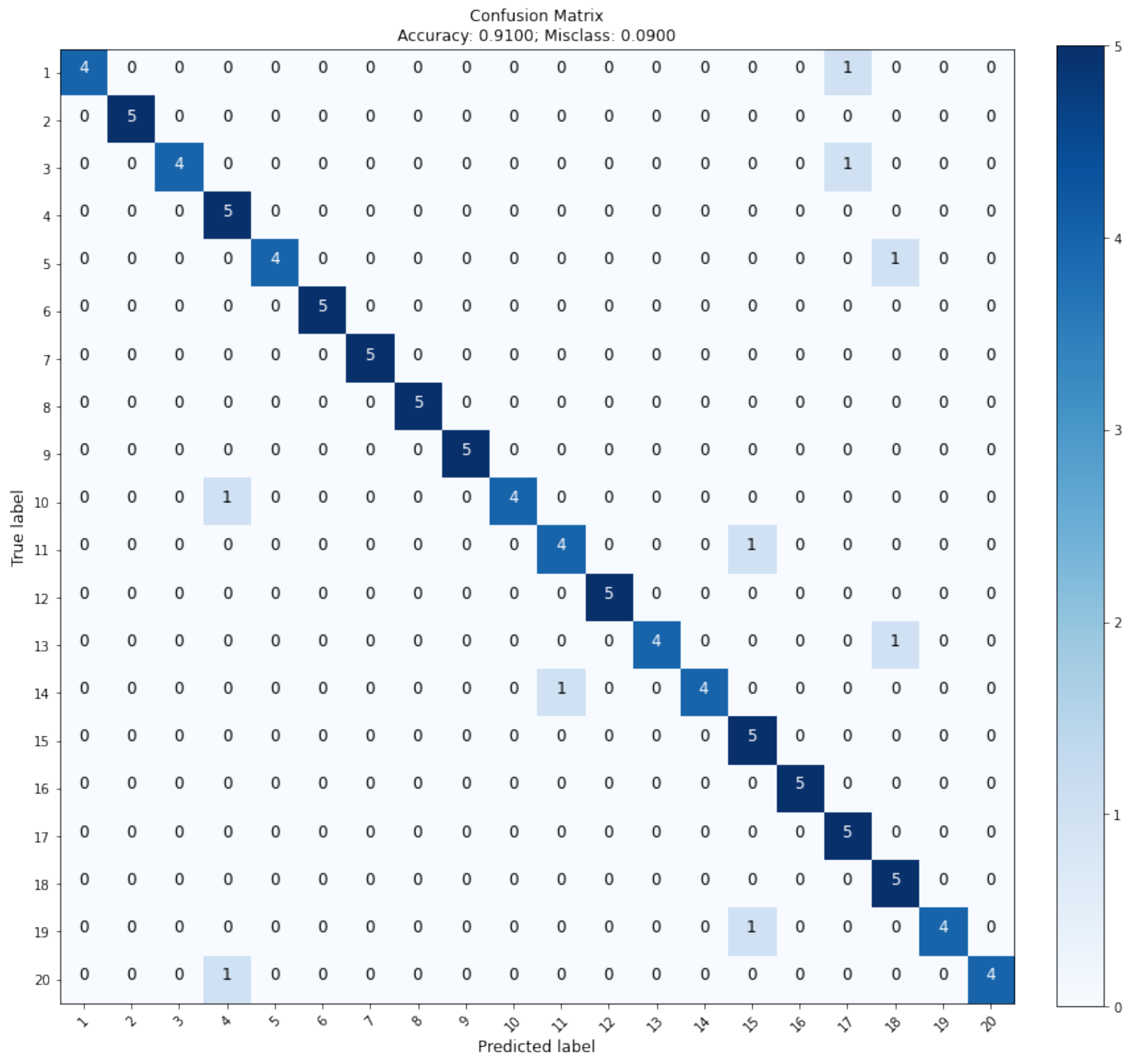


Figure 4: Confusion Matrix for the KNN classifier using PCA features.

The outcome of the classifier performance for this subject is shown below:

```
Overall Accuracy: 91%
Subject #1:
    TP: 4, FP: 0, TN: 95, FN: 1
    TPR: 80.00%, TNR: 100.00%, FPR: 0.00%, FNR: 20.00%
```

Given the fist subject, i.e. `subject_id = 1`, the measurements such as TP (True Positive), FP (False Positive), TN (True Negative), FN (False Negative), TPR (True Positive Rate), TNR (True Negative Rate), FPR (False Positive Rate) and FNR (False Negative Rate), are calculated for this subject. All these metrics can be derived from the Confusion Matrix shown in Fig. 4. For example, the overall accuracy (as shown by the code) may reach 91% (0.9100).

To create this visualization of the confusion matrix, you have to run the following commands:

```
# Generate the confusion matrix
confusionMatrix = confusion_matrix(testingLabels, predictedLabels)

# Plot the confusion matrix
plot_confusion_matrix(cm=confusionMatrix,
                      target_names=[i for i in range(1, n_subjects+1)])
```

The code above calls the Scikit-Learn's `confusion_matrix(...)` function to generate the numerical matrix. The second command (`plot_confusion_matrix`) shows the confusion matrix.

The confusion matrix has the size which represents the number of classes (subjects) used in classification; in our case, it is 20×20. Each row corresponds to the *true labels* while the columns correspond to the *predicted labels*. The numbers in each cell show how many samples were predicted according to the true and predicted label. For example, in the first row, there are 3 samples of true label 1 (subject 1) correctly predicted as label 1. However, there are two labels 1 wrongly predicted as label 4, and another one predicted as label 19. The ideal case is a *diagonal matrix*, where each true label is predicted correctly.

3 Lab Report

Your report in the form of a Jupyter Notebook/Python (file extension `.ipynb`) shall include the following graded components (10 marks total):

- Introduction (a paragraph about the purpose of the lab).
- Description of the result on each exercise with illustrations/graphs and analysis of the results (marks are distributed as shown in the Exercise section) (10 marks).
- Conclusion (a paragraph on what is the main take-out of the lab).

Save your Notebook using menu "Download As" as `.ipynb`, and submit to D2L dropbox for Lab 6, by the next lab session (next Friday by 9am).

4 Lab Exercise in Jupyter Notebook with Python

For the following exercises, use the subset of images from AT&T dataset, or your own images, and some images of non-faces (remember to use the same size, i.e. the size must be normalized prior to processing).

- **Exercise 1** (3 marks): For this exercise, you are going to build a gallery and perform the classification (matching) for the three cases': face known to the gallery, face unknown to the gallery and not a face, as shown in Fig. 2. However, you need to choose two or three different subjects (for example, subjects 2, 3 and 4) from the dataset (gallery), or create your own. Use the first 9 images of each subject to build the gallery. For example, if you choose 3 subjects, then you have a gallery of 3 x 9 = 27 faces, and for algorithm each image is a sample, or input (no separate model is built per subject).

  Now, choose the probe images as follows:

  1. The 10th image of each of the three subjects in the gallery.

2. 2-3 faces of the subjects not known to the gallery.
3. 1-2 not-a-face image.

For each of the comparisons, save the Euclidean distances calculated, and mark the minimum and maximum value in each case.

- **Exercise 2** (3 marks): Determine the first and the second thresholds; hint: analyze the maximum of the Euclidean distances for the probe images known to the dataset.

  To find the second threshold, use unknown faces as the input faces and gather all the Euclidean distance. Hint: consider the maximum as the second threshold, but also analyze other values.

- **Exercise 3** (2 marks): In the description of the face recognition using PCA and KNN above, the PCA was used with 100 principal components. Now repeat the same procedure using 50 and 200 principal components. Evaluate the resulting classifier accuracy, using Confusion matrix. Visualize the comparison using either a table or a bar plot (comparing the results for the three number of components: 50, 100 - already done - and 200).

- **Exercise 4** (2 marks): Evaluate the impact of the number of neighbors (K) defined for the KNN. Using the same data separation (20 subjects, 8 images per subject are used for training and 2 for testing), conduct the two experiments considering, for example: 1) 5 neighbors and, 2) 11 neighbors. Compare the KNN classifier results, using Confusion matrices, and draw the conclusions.

5 Acknowledgments

We would like to acknowledge the TA I. Yankovyi for testing this lab exercise.

Dr. S. Yanushkevich, Dr. H. C. R. Oliveira.
February 16, 2022