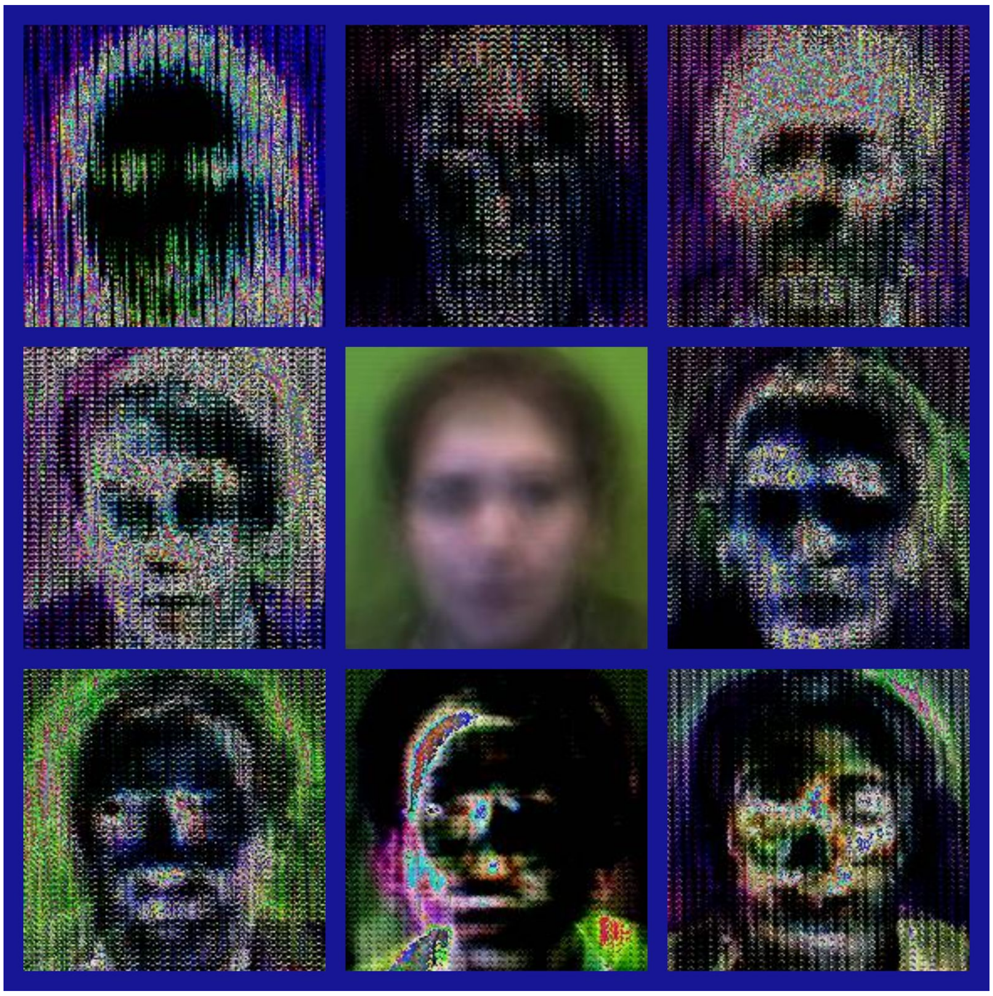
uc3m | Universidad Carlos III de Madrid

Assignment 1: EigenFaces

Statistical Learning

Group 96-97

Aya Ben Hriz and Alejandro Bermejo Gordillo



Introduction

Face recognition may be one of the most mainstream uses of computer vision. Its rapid development has to be attributed to law enforcement agencies all over the world to help identify criminal suspects. From a more general user perspective, it is used to group photos by who is on them or as a security measure to unlock our devices [1].
More formally, we can formulate face recognition as a classification task, where the inputs are images and the outputs are people's names. We're going to discuss a popular technique for face recognition called **eigenfaces** [2]. And at the heart of eigenfaces is an unsupervised dimensionality reduction technique called **principal component analysis (PCA)**, and we will see how we can apply this general technique to our specific task of face recognition.

A - The PCA function

We will discover through the whole process that our PCA function is of an important use.This will help speed up our computations and be robust to noise and variation.
We use pca as a technique of dimensionality reduction and notice we're not performing PCA on the entire dataset, only the training data. This is so we can better generalize to unseen data.
Our function takes as an input data whose sample covariance is to be calculated and returns the mean which is known as mean face and the variance which we want to keep, since they will later be used when projecting new data into the existing eigenspace.
also it returns the eigenvectors which will be used to use in the eigenface projection and to decide the number of principal components that we have to choose to optimize our algorithm

BC - Building a k-NN face classifier via the eigenfaces approach

Preliminaries

When reading an image using the `OpenImageR` package, it is stored as a 3D array containing matrices for each of the color channels of our images of number of rows and columns according to the images' height and width. In order to apply PCA, we vectorize the images so we can describe a whole image as one row in our matrix.
The database we get is composed of 150 RGB images of 180*200 pixels; therefore, each observation is placed as a vector in a space of 108,000 dimensions.
The aim of our work is to identify a specific person in the database with an image. So, even though an image's filename contains more information, we are going to only utilize the first number which can be considered as the person's ID. We'll store our labels in a character vector for later use.
Note that the classifier will use all the images we have as the training data. We will disclose why we choose certain parameters later in this document.

Eigenfaces

Our classify function takes an image path in the working directory or a 2D/3D image array as a parameter. The image is first vectorized.
Next, we run our `pca` function on the training data, getting a list containing the mean of its columns, the 150 eigenvectors and the proportion of variance each eigenvector represents.
To implement PCA we first need to scale our data, thus, we remove the mean from our training data.
We compute the vector of the cumulative proportion of variance because the adding complexity of utilizing more eigenvectors for a more accurate representation is not justified in our case. We decided on retaining those eigenvectors which represents more than 95% of the variance, which resulted to be 25. We can now multiply our scaled data transposed by the matrix of 25 eigenvectors to obtain our 108,000*25 eigenfaces matrix.
Each column in the eigenfaces matrix represents patterns of faces. The image in the middle of our cover is the mean image we get from our training dataset. The ones surrounding it however, are the first 8 eigenfaces from the matrix. It can be seen that there is a clear distinction in all images of the nose, eyes and lips and also the shape of the head.
We can now project our training data on the space of the eigenfaces multiplying our scaled training data by the eigenfaces getting a 150*25 matrix. Returning to the image we wanted to predict, we scale it by subtracting the mean and projecting it into the same space by multiplying the row vector by the eigenfaces.
We now `rbind` both our projected image with our projected training dataset to use k-NN.

k-Nearest Neighbors

When using k-NN we have to compute the distance of the observation that has to be labeled to that of all of the data points in our training set. We then sort k of those points based on the distance. The unlabeled observation will be labeled as the label that appears most frequently on the list we selected. In case of a tie, we have chosen to set the label as the one of those labels tied closest to the unlabeled one.
We use our projected data to compute the distances to every point and pick the distances of our image with the training data using the `manhattan` distance method. We sort the observations and use the rules aforementioned to select a label. We have just assigned our unlabeled image a label.

Validation Strategies

We used validation strategies to optimize certain parameters of our model:

- How much variance we are retaining
- The number of k-nearest neighbours
- The distance method used
- The threshold to determine when an image belongs to the database

In order to do this, we broke down our code into smaller functions:

`findK(obs, train.data, method, maxK)`

Using a projected image and projected training data, including labels, it runs the knn algorithm using the method specified for every `k` from 1 up to `maxK`. We decided on using a `maxK` value because it did not add much computational complexity to the program. This function returns a vector containing the label predicted for each of the k's tested whose last two elements are the real label of the test observation and the farthest observation which has the same label as the test..

`testMethod(test.data, train.data, method, maxK)`

Given projected and labeled test and training data, it runs `findK` on all elements of the test set using `method` and `maxK`. It returns the accuracy for each k on each element of the test set as a vector and the maximum distance of those obtained using `findK`.

`plotGenerator(th, maxK)`

This function generates a bar plot for a specific threshold `th` of percentage of variance that displays the accuracy of the classifier for each `k` and the three distance methods we were recommended to try, `manhattan`, `cosine` and `euclidean`.

Repeated stratified k-fold cross-validation

Inside the `plotGenerator` function is where we implemented our validation strategies. We first added a column with the index of our dataset so when we used `str` from the `splitstackshape` package we just stratify 2 columns which is less computationally expensive than using the full 108,000. We use `str` inside our loop for the repetitions and end-up with 6 folds of data, containing each 25 images from different persons. We loop through them calling `testMethod()` and obtain the data necessary for creating the accuracy plots.

Optimization Results

Here we present the plots (Figure 1) we obtained for out-of-sample accuracy.
It is obvious that the accuracy for `k=1` and `k=2` is always the same because with `k=2` in case of a tie the closest label would be given.

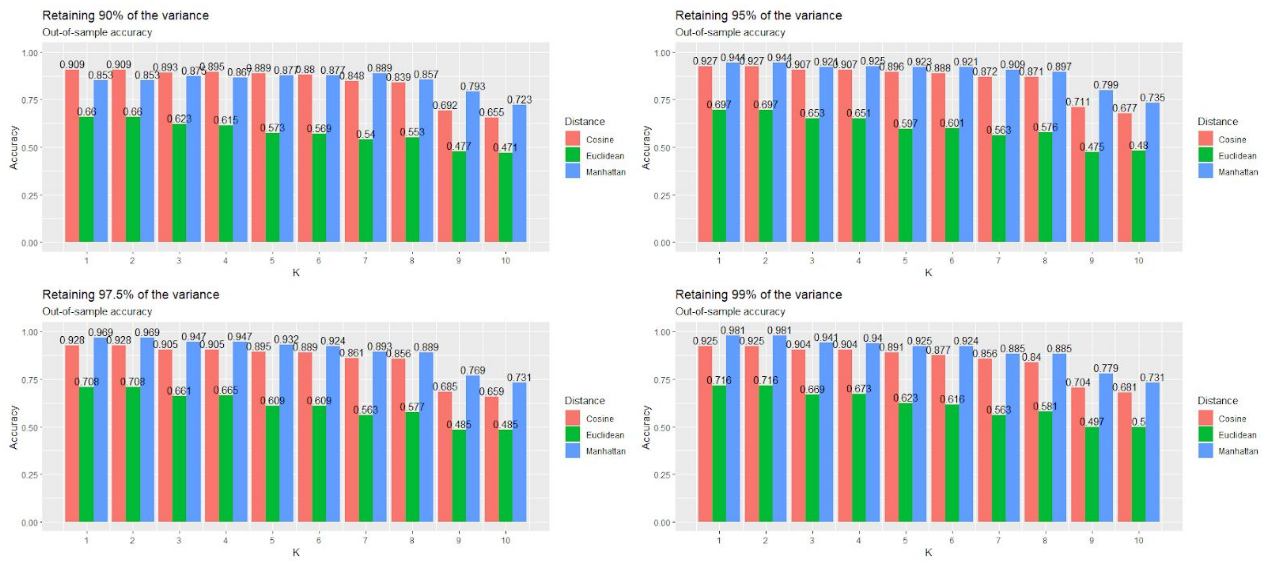


Figure 1. *Plots of the result of cross-validation*

We achieved our highest accuracy using k=1, but this will lead to overfitting since all training images seem to be frames from a video and are similar. With k>3 we see that there is a greater drop in accuracy in comparison to the previous k's from k=6 to k=7 in all plots. Therefore, we choose k=6.
We chose the manhattan distance because it was the one yielding higher accuracies.

90%	95%	97.5%	99%
2.94	2.97	3.34	3.94

Table 1. *Time spent to label an observation*

Variance	90%	95%	97.5%	99%
90%	-	1.01	1.13	1.34
95%	0.98	-	1.13	1.33
97.5%	0.88	0.89	-	1.18
99%	0.75	0.75	0.85	-

Table 2. *Fraction of time for increase of variance*

The difference in computation time from 90% and 95% is unnoticeable. The increase in accuracy from 95% to 97.5% is around 2.5% according to the graph, we do not deem it an important matter since the increase in computational time is 13%. So we decide on using 95% of the variance.
We stored the distance of every test observation to the image in the training set with the correct label that was the farthest. Then, we computed the mean of those for all of our repetitions and ended up with a value of 53827.58. If the closest image we want to classify

has a distance greater than this one, we reject it and conclude that the person is not in our database.

D - Building a k-NN face classifier without reducing dimensionality

Our hypothesis here is that this classifier will take longer to run since the k-NN is based on the distance and computing that in a 108,000 dimensional space is not an easy task. So we used the `tictoc` package (Table 3) and confirmed that our hypothesis was correct. Both classifiers were using the same parameters of `k=2` and using the manhattan method, but the no-PCA classifier took more than 10x more time than the one using PCA.

Classifier	Time to run (s)
PCA	2.78
No-PCA	35.23

Table 3. *PCA vs no-PCA run-time in seconds*

We were already achieving very high accuracy rates on our previous classifier, so we don't think that would be a very relevant attribute to discern between the two. But of course, it would depend on the purpose of the model.
One example we can imagine for this dataset is that all of them are coworkers and there is a restricted entry room in the office. To make it more convenient they decide to install a facial recognition security system on the door. Waiting 3 seconds for it to open might be faster than putting the key into the keyhole, but waiting 30 seconds for that would be bothersome, tiring and unsafe since a lot can happen in that timeframe that the door is still unlocking.
We feel that most of the uses of a classifier like this would choose speed over a little more accuracy, so we would choose the PCA classifier.

Conclusion

We can then conclude that our highest accuracy obtained was 0.944 when retaining 95% of our data and using the parameters k=1 ,2 and Manhattan distance.
Although with retaining 97,5% of our data we get a higher accuracy of 97,5% we may observe that it is not worth it since the accuracy difference is only 2% for an increase of 13% for the computational time.
Also we were able to realize the importance of dimensionality reduction techniques since as we demonstrated the no-PCA classifier was 10x slower than the one using PCA and in real life that would be extremely difficult to achieve with a massive dataset besides costs would be high.But we've also noticed that some downsides to this approach. if we had a l database with more images and faces to recognize, then executing this comparison for each face one by one will take a long time! Imagine

that we’re building a face recognition system for real-time use! The larger our dataset, the slower our algorithm. But more faces will also produce better results

References

[1] T. Klosowski, “Facial Recognition Is Everywhere. Here’s What We Can Do About It.,” *Wirecutter: Reviews for the Real World*, Jul. 15, 2020. https://www.nytimes.com/wirecutter/blog/how-facial-recognition-works/ (accessed Nov. 01, 2020).

[2] “Eigenface,” *Wikipedia*. Oct. 01, 2020, Accessed: Nov. 01, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Eigenface&oldid=981330287.