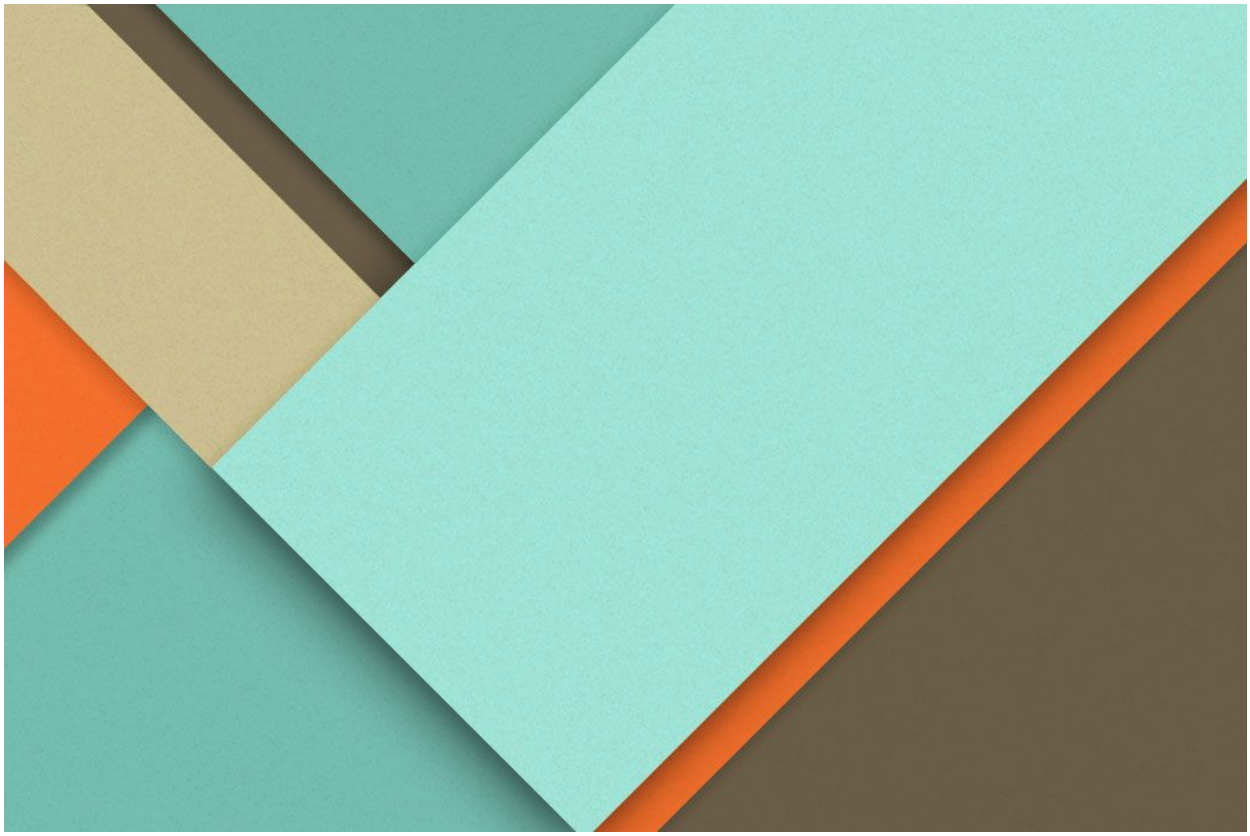


Computer Vision HW2

19.05.2017

— Face Recognition System

Andrey Leshenko & Eli Tarnarutsky
322026527, 318880911
University of Haifa

Lecturer: Hagit Hel-Or
HW Checker: David Cohn

Project Goals

In this project we implemented a **face recognition system.**

The system's input is a dataset of labeled images - "in the wild" pictures, each one of them with one person. Those are the **train images.** The system also receives a dataset of **test images** in the same format.

The system creates a model based on the train dataset, and uses this model to classify the images in the test dataset.

Running the Code

The code is written in C++11. We use OpenCV which has to be installed, and Dlib, which we compile from source.

Out code is build using CMake.

Windows

1. Install CMake from the official site.
2. If it is not yet installed, install OpenCV from the offical site. The installer will ask you where to extract the files. Remember this location. (On the Jacobs building computer it is C:/opencv3)
3. Open the CMake program.
4. Set "Where is the source code" to the correct path (for example E:/Andrey/cvhw2).
5. Set where to build the binaries to the came path + "/build" (for example E:/Andrey/cvhw2/build).
6. Press the Generate button below.
7. CMake will ask for a visual studio compiler version. Select **V14 2015**.
8. CMake will say "**Error in configuration process**", because it can't find OpenCV. Press OK.
9. In the middle of the window there is a field "**OpenCV_Dir**". Set this field to the "build" directory inside your OpenCV installation. On the Jacobs computer this is "C:/opencv3/opencv/build"
10. Press Generate again. You should see "Generating done".
11. There is now a ready Visual Studio project inside the "build" folder. Open it by pressing the Open Project button in the middle of the CMake window.
12. You can compile and debug using Visual Studio!

Important points: you can choose whether to compile with Dlib by setting the USE_DLIB define. When it is set, do not compile in Debug mode, as the Dlib face detection in **debug mode** is extremely slow.

Linux

1. Install cmake using the package manager
2. Install Opencv. On fedora from the package manager, on Ubuntu you have to compile from source (as of May 2017).
3. Navigate to the source directory
4. Execute the following commands in the terminal:
    a. mkdir build
    b. cd build
    c. cmake ..
    d. Make
5. Run the program by executing "./cvhw2"

Theory of Eigenfaces

Let's say we have a small 100*100 pixel image. This image is essentially a 10,000 element vector in a 10,000 dimensional space. However, each of the pixels (or dimensions) doesn't tell us anything meaningful about the image, as there is much more noise than data - most of the images aren't images of faces.

Using algorithms like Principal Component Analysis we can create a much smaller "Face Space" (for example a 32 dimensional space). In this space each dimension holds a lot more information. We can compare how close two faces are using simple Euclidean distance inside the face space.

Now the problem of recognizing faces turns into a much simpler problem: for each person we have a set of points in the face space. Given a test point, we have to decide to which of the sets it belongs. There are a few options we implemented:

- Nearest Neighbor: take the label of the closest train point.
- K Nearest Neighbors: take the label that most of the K nearest neighbors have.
- Mahalanobis Distance: Calculate the distance to the mean of each set. Normalize each dimension according to the standard deviation along this dimension. Use this distance to find the closest set.

What we described here is the algorithm we implemented in this project. The complete algorithm is:

1. Align and normalize train images

2. Create a PCA model (face space) of the train images
3. Project the train images into the face space
4. Align and normalize the test images
5. Project the test images into the faces space
6. For each test image, find the person class that is closest to this image. All distances are measured in face space.
7. Display the results

Architecture

Command Line Interface

Using the program is similar to using Terminal or CMD. You type commands together with arguments, and the program executes them. Available Commands:

train PATH

Train the face recognition on the dataset found at PATH (Relative to the "images" folder of the project). The dataset must be organized as a set of directories, one for each person, each directory containing the photos of this person. When you append "#even" or "#odd" to the path, then only the even or odd images of the dataset are loaded. The program will create a PCA model for this dataset

train+ PATH

Same as train, only the dataset at PATH is added to the images used in the previous training.

test PATH

Loads the dataset found at PATH (in the same format as in train), tries to recognize the faces in the dataset, and displays the result. When you append "#even" or "#odd" to the path, then only the even or odd images of the dataset are loaded.

dist_type TYPE

Sets the distance type used in face recognition. The options are "knn" or "mahalanobis" ("m" works too).

dlib BOOL

Enable or disable Dlib face detection and alignment. 0 disables, and 1 enables. Note that to be able to set dlib to 1, you have to compile with USE_DLIB defined.

show_eigenfaces

After training is done, shows that mean face and the eigenfaces created using PCA.

show_train

Shows the train images and their projections into the eigenfaces space.

show_test

Shows the test images and their projections into the eigenfaces space.

exit

Exits the program

Here is an example of a command line session with the program:

```
?> train faces96#even

?> show_train

?> test faces96#odd

?> dist_type mahalanobis

?> test faces96#odd

?> exit
```

ImageDB

To make the program run faster we implemented a special solution - the ImageDB. An ImageDB is a data structure that holds all the processed images. Each image is uniquely identified by its path, so it is never loaded twice. Moreover, the ImageDB is saved to disk (in a simple binary format), which means that once an image is loaded or created, it will be available in the next runs of the program.

For example: say we load and image “dany.jpg”. This image will be stored in the ImageDB as “dany.jpg”. We then want to find the face and normalize this image. The new image will be called created and stored as “dany.jpg#NORMALIZED_VJ”. The next time we want to use “dany.jpg” or “dany.jpg#NORMALIZED_VJ” the image will just be taken from the ImageDB with no more work to do.

In addition to regular images, the PCA eigenvectors are also stored in ImageDB to speed up the program.

Face Alignment

Before faces can be recognized, they have to be found and aligned to the same form. There are two options available: using OpenCV’s CascadeClassifier (based on Viola Jones), or using the Dlib machine learning library.

With Viola Jones

First, we tried running the system with the built in VJ function in OpenCV (*CascadeClassifier* with the matching "haarcasdade_frontalface_default.xml" file).

The Viola Jones algorithm is good at detecting faces in low-resolution images. In our experiments we found an almost 100% recall, but also a noticeable false-positive rate.

However, the Viola Jones eye and mouth detection algorithms were very unreliable in our experiments. Sometimes the eyes were detected alongside with other parts of the face like nostrils, sometimes only one eye was detected or neither of them. Thus, we couldn't rely on eye detection to align the images robustly.

At the end, when using Viola Jones, we align the faces according to the detected rectangle OpenCV gives us.

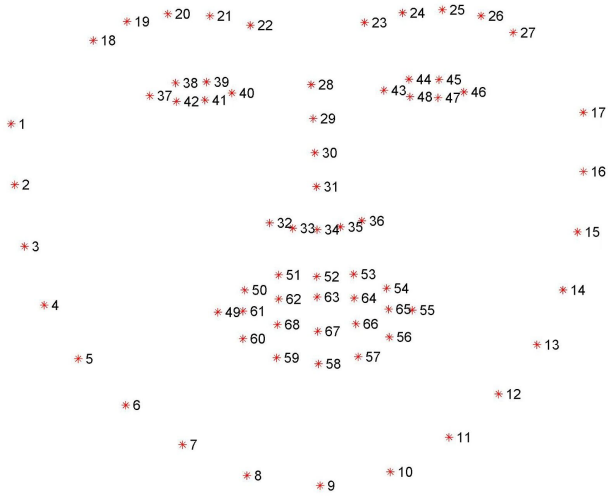
With Dlib

Dlib is a free, open source C++ machine learning and image processing library.

We used Dlib's frontal_face_detector, alongside with the 68 landmarks face model, which is available at the following link.

After some experiments, we found out the Dlib's detection to be very accurate, precise and reliable. The only downside was not very good performance on very low-resolution images, but this isn't a problem is most datasets.

We had to write a few functions to translate Dlib's output into OpenCV's datatypes. Eventually, we got 68 points representing the face by the model shown here.



To align the faces, we use an affine transformation: we calculate the centroids of the eyes and mouth and create an affine transformation that maps them to fixed points in the

image. We create the transformation using OpenCV's getAffineTransform function. Finally, we apply the transformation to the image and crop it to the desired size, using warpAffine.

No matter if we use Viola Jones or Dlib alignment, we resize the image and apply histogram equalization to remove lighting. After this part we have small, equally sized images of the faces that go to the next step.

PCA

The PCA algorithm tries to closely represent a high dimensional space using a small set of basis vectors. In our case, the images contain more than 10,000 dimensions (pixels), and we reduce them using PCA to less than 32.

PCA stands of Principal Component Analysis - a statistical method that takes in account the variance of the samples on the different axis, and can output a set of n vectors that can represent the samples as a new base with satisfyingly high accuracy.

An interactive explanation is available at this link.

We use PCA in the normalized train images during the train step. We used OpenCV's implementation cv::PCA.

Recognitions and Distance Calculation

After we create the face space using PCA, both the train and the test data is projected into it. For each person in the train dataset we have a set of points in face space that describe him. Given a test image (in face space) we find which set is closest to the test image - this will be the result of the recognition.

We implemented two function for distance based classification:

classifyKNN implements the K Nearest Neighbors algorithm, described in the theory section.

classifyMahalanobis uses the Mahalanobis distance to find the closest label to each train image. It calculates the mean and standard deviation along each axis, and calculates distances to the mean normalized using the standard deviation.

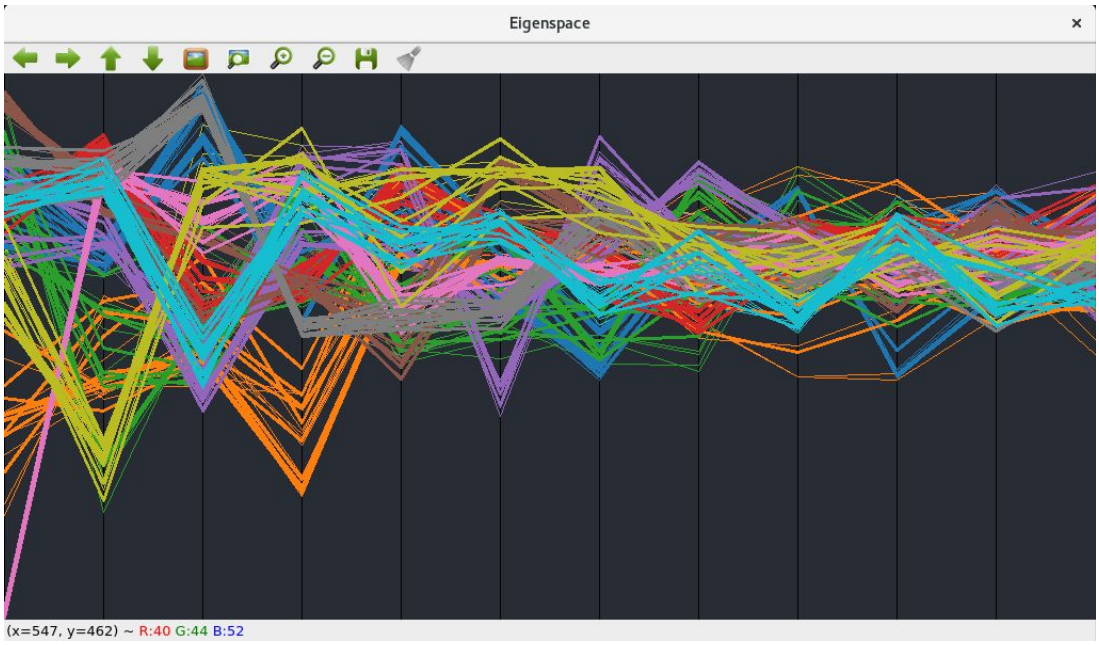
Note: One option is to choose a distance threshold, so if a test image is far from any of the train images it is marked as unrecognized. We decided not to do that, because we didn't want a hard-coded threshold that will not work in all cases.

FaceRecSystem

This class contains all the parts necessary to train and test the algorithm. The functions of FaceRecSystem are called from the main function, which parses the command line input.

Visualizations

When executing the **test** command, a visualization window opens:



The X axis displays the coordinates in the facespace, and the Y axis displays the values of these coordinates. Each line is an image. Different labels (people) are displayed in different colors. You can use the 'J' and 'K' keys to look at specific labels:

For each label, the white lines show the train images. The green lines show that test images of the label that were recognized correctly. The red lines show the test images of that label that were recognized incorrectly. The black lines shows the test images of other labels that were incorrectly recognized as this label.

Press 'Q' to exit this window.

This window is useful for understanding which problems that algorithm is facing.

Results

A - Training

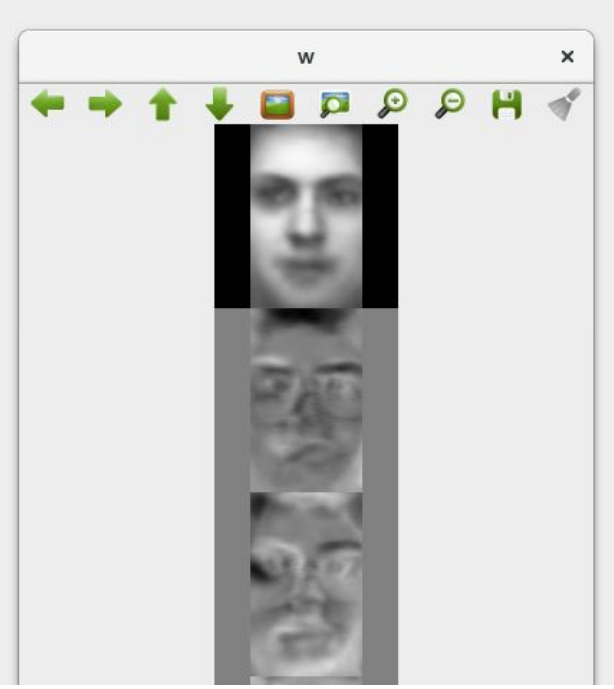
We are going to use the faces96 dataset. We separated it into "faces96_no_makari", which contains 9 people but not Makari, and "faces96_makari" which contains only Makari. We will use it for testing.

First we train on the even images without Makari:

?> train faces96_no_makari#even

Now we look at the eigenfaces

?> show_eigenfaces

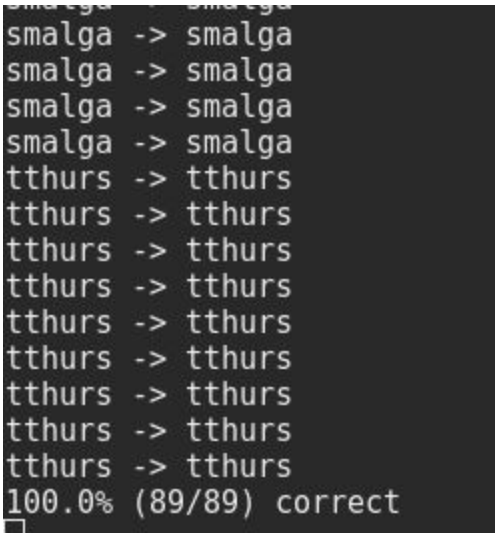


B - Testing on the train images

We now test on the same images we trained on:

?> test faces96_no_makari#even

We get everything 100% correct!



B - Testing on unseen images

We will now test on the odd images from faces96_no_makari (we didn't train on them):

?> test faces96_no_makari#odd

Again 100% correct!

```
smalga -> smalga
smalga -> smalga
smalga -> smalga
tthurs -> tthurs
tthurs -> tthurs
tthurs -> tthurs
tthurs -> tthurs
tthurs -> tthurs
tthurs -> tthurs
tthurs -> tthurs
tthurs -> tthurs
tthurs -> tthurs
tthurs -> tthurs
100.0% (90/90) correct
```

B - Testing on the missing person

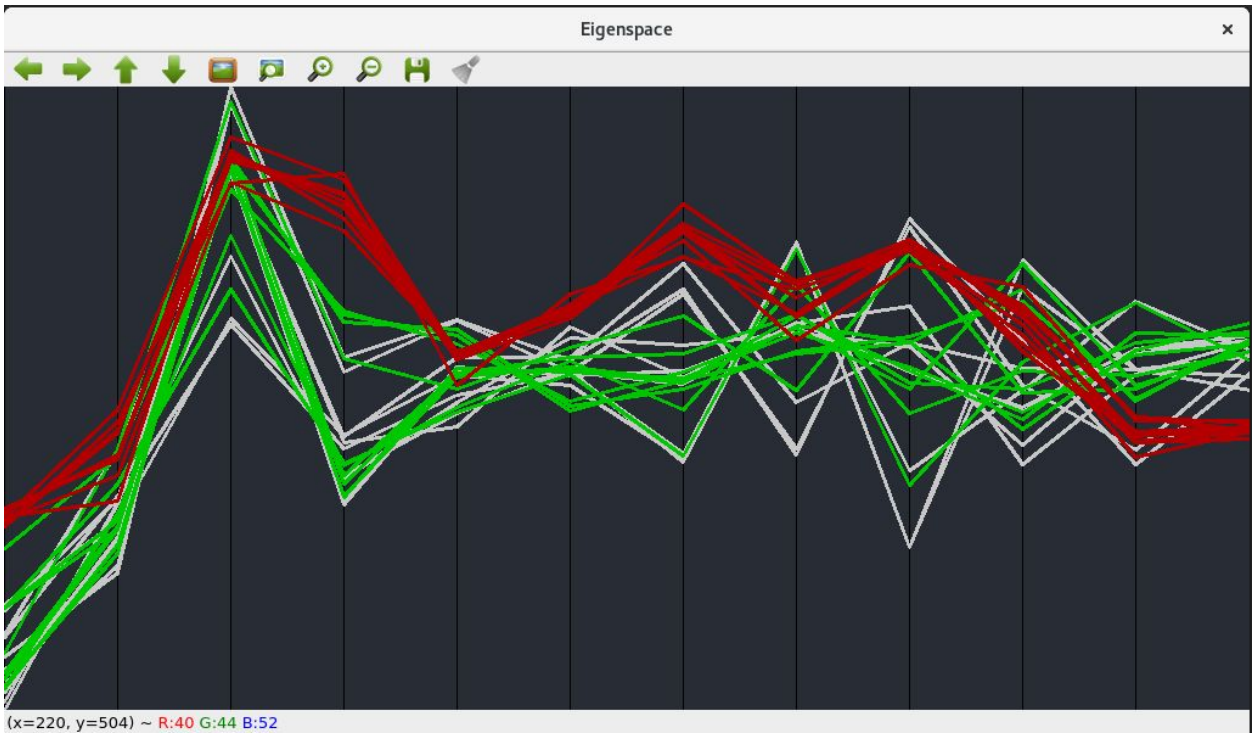
Now we will train to recognize all the pictures, even the ones containing Mr. Makari:

?> test faces96#odd

We can see that while everyone else was recognized correctly, Makari not recognized correctly:

```
tthurs -> tthurs
tthurs -> tthurs
tthurs -> tthurs
tthurs -> tthurs
tthurs -> tthurs
nmakri -> arwebb X
nmakri -> arwebb X
nmakri -> arwebb X
nmakri -> arwebb X
nmakri -> arwebb X
nmakri -> arwebb X
nmakri -> arwebb X
nmakri -> arwebb X
nmakri -> arwebb X
nmakri -> arwebb X
90.0% (90/100) correct
```

Here is how this looks in the facespace:



Now we train the system again to include Makari this time:

?> train+ faces96_makari#even

And test again:

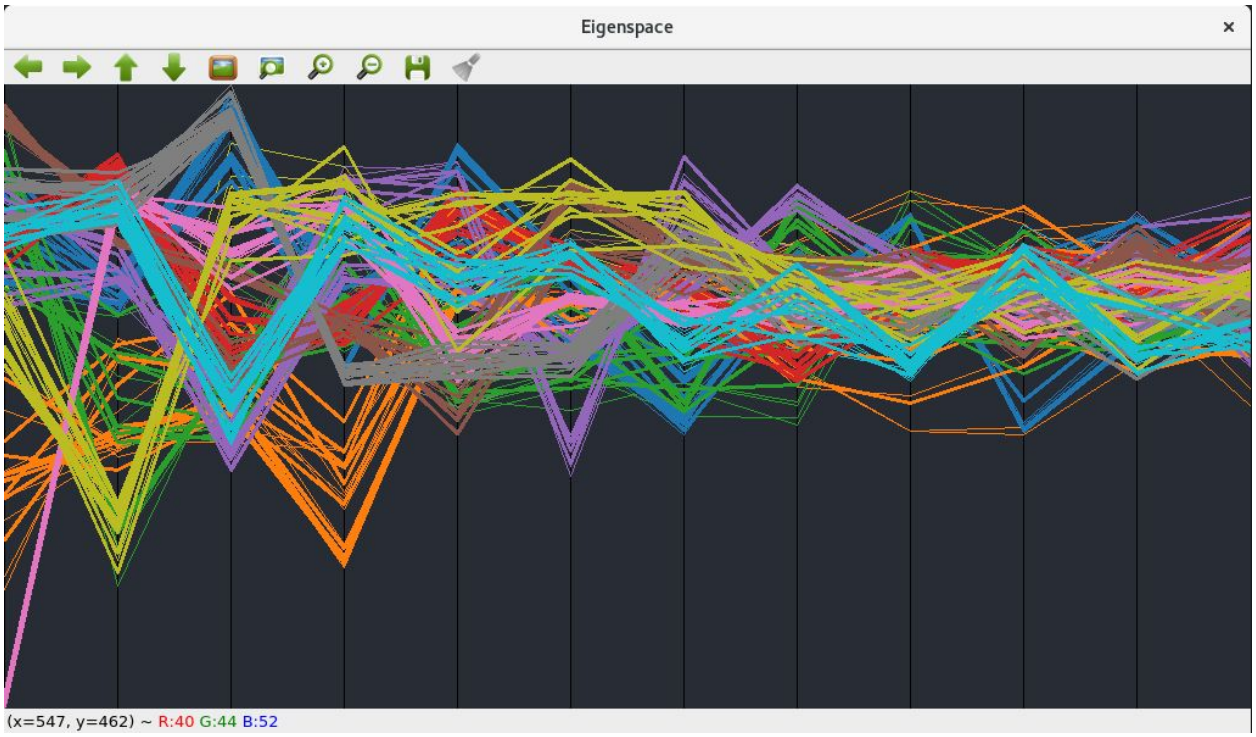
?> test faces96#odd

This time the system did recognize Makari!

```
mclarkd -> mclarkd
mclarkd -> mclarkd
mclarkd -> mclarkd
nmakri -> nmakri
nmakri -> nmakri
nmakri -> nmakri
nmakri -> nmakri
nmakri -> nmakri
nmakri -> nmakri
nmakri -> nmakri
nmakri -> nmakri
nmakri -> nmakri
nmakri -> nmakri
nmakri -> nmakri
rbrown -> rbrown
rbrown -> rbrown
```

```
tthurs -> tthurs
tthurs -> tthurs
100.0% (100/100) correct
```

In the face space:



E - Where the system fails

Let's try another dataset - gt_db. This dataset is harder because of different lighting conditions, and because the subjects aren't looking straight forward.

?> train gt_db#even

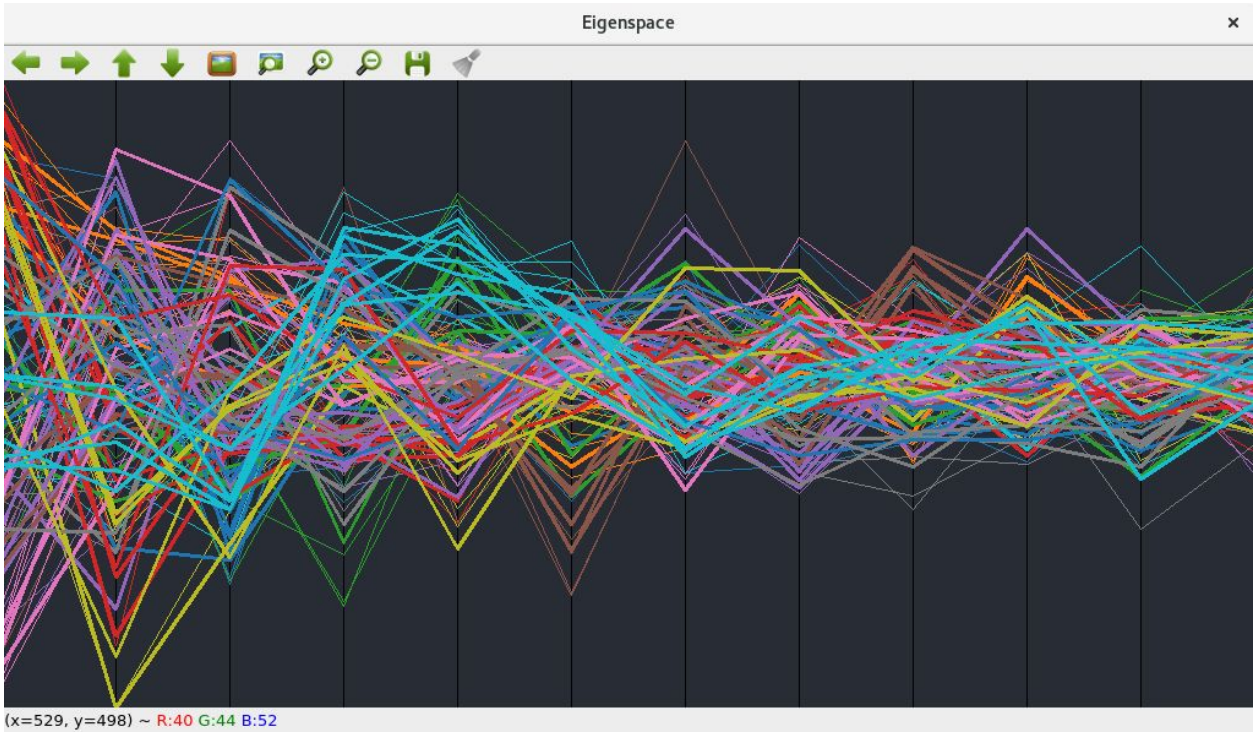
Now let's test:

?> test gt_db#odd

We can see that now the detection ration is only 77.6%

```
s45 -> s45
s48 -> s48
s48 -> s48
s48 -> s48
s48 -> s48
s48 -> s48
s48 -> s48
s48 -> s48
77.6% (52/67) correct
```

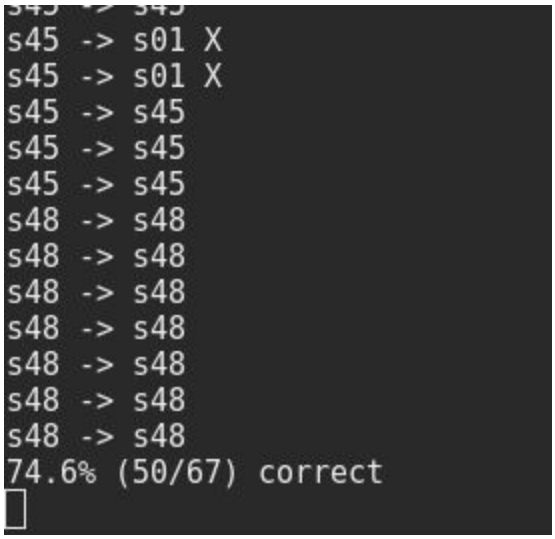
And the points in the eigenspace look like a total mess.



Let's try using the Mahalanobis distance instead of K Nearest Neighbors:

?> dist_type mahalanobis

?> test gt_db#odd



Not better. Let's using Dlib alignment instead: (We have to define USE_DLIB for this to work)

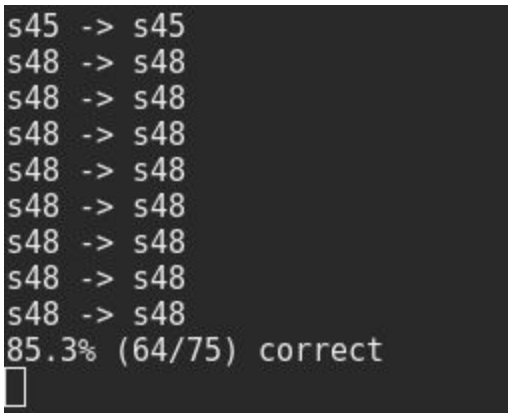
?> dist_type knn

?> dlib 1

?> train gt_db#even

?> test gt_db#odd

Yay! 85%. Dlib detection does help in this case.



If we look at the mistakes we made in the face space:

We can see that all the curves are very close to each other, so it is understandable that the system fails in such cases.

F - The number of eigenfaces

From tests we made, the more eigenfaces there are, the better the detection worked for us. Here is a table that shows the detection accuracy as we increase the number of eigenfaces.

1	16.4%
2	44.8%
3	46.3%
4	58.2%
5	64.2%
6	71.6%
7	70.1%
8	71.6%
9	74.6%
10	76.1%
11	79.1%
12	77.6%
14	76.1%
16	77.6%
18	80.6%
20	79.1%
25	77.6%
30	79.1%
35	82.1%