**Face Entropy**

Roy Zohar, Roy Mezan

Prof. Rita Osadchy, Dr. Orr Dunkelman

# Background

Biometric recognition is a fast-growing field, that has gained more and more popularity in the recent years. Despite common belief, we hope to show that these databases are not as safe as they seem.

Assume a system guarded by a password, of 4-15 characters (including upper cases, lower cases and digits). One can easily calculate the number of possibilities for a password matching this criterion, and it stands at around 62^16. This number gives us a sense of how safe a system can be to brute-force attacks.

However, when dealing with biometric databases, this question becomes much more difficult to answer. Does a picture of a face convey more possibilities than a 15-character password? Will we always be satisfied by the number of features in a face, or will this limit be exceeded? These are the questions we hope to answer in this project.

Our main goal is to develop a simple "brute-force" attack on a biometric database, specifically a **facial** database. We accomplish this by synthesizing a large amount of artificial feature vectors, based on a set of predefined faces.

# Definitions

We chose the Dlib vector representation, which is an 128-dimensional vector, that represents features of a certain face. Details regarding the library can be found at <http://dlib.net/>.

Two faces are considered similar, if the Euclidian **distance** between their representation is "small".

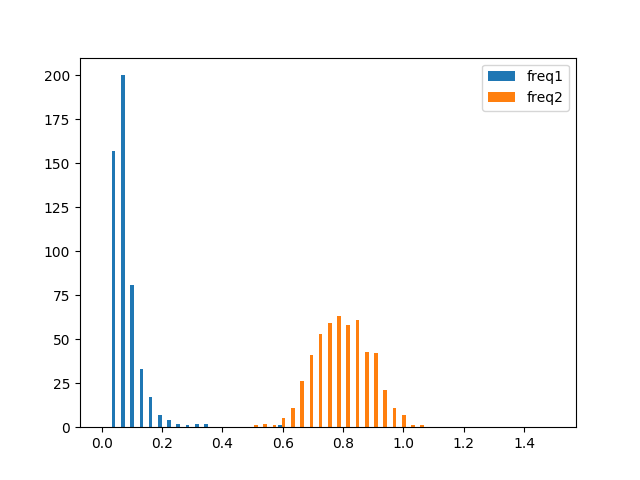
We use the terms **training database**, and **testing database** to describe the datasets of faces for which we train our model and test our model against respectively.

We also used **CSV** spreadsheets for storing features.

# Determining the threshold

One of the first questions that was asked, was what the ideal threshold is, where we can say that 2 faces represent the same person. The DLib library recommends using a threshold of about 0.6, but since the database we use is considered "easy" (in the sense that all pictures were taken in the same format), we went ahead and tried to determine the threshold ourselves.

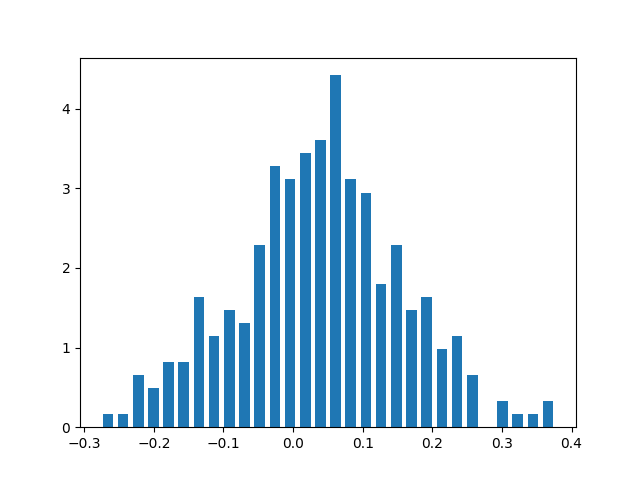
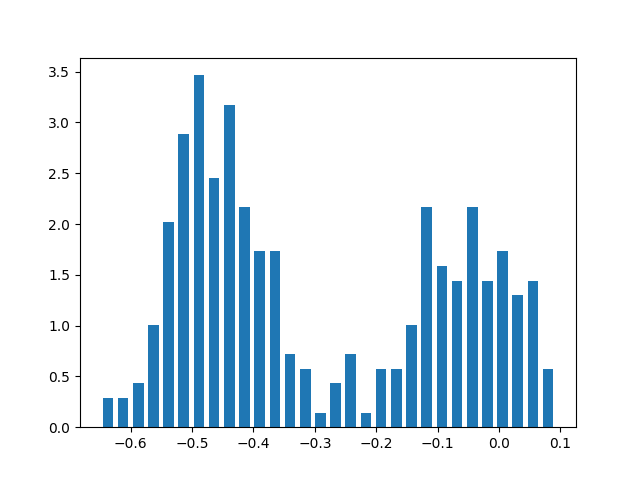
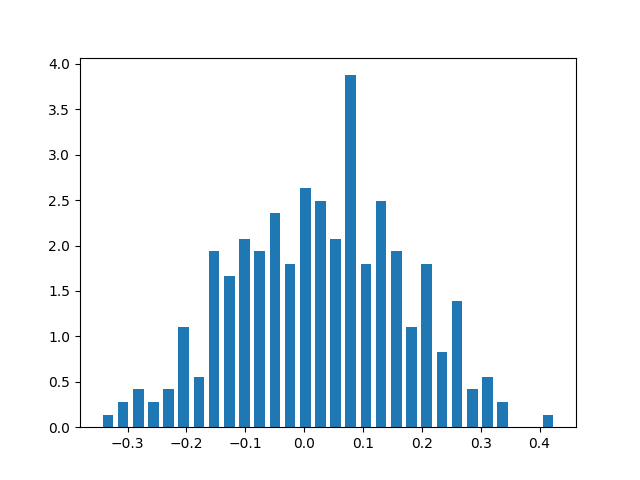
To determine this threshold, we plotted the distances between two pictures of the same person (Shown in blue as freq1), and two pictures of different people (Shown in orange as freq2):

[](https://github.com/Royz2123/Biometric-Attack/blob/master/figures/threshhold500.png)

As can be seen here, the ideal threshold to minimize false-positives and true negatives lies around the 0.4 mark, since it sits in the middle of the 2 distributions.

# Synthesizing Faces

After setting the threshold at 0.4, we were able to synthesize feature vectors of faces, and check them against features of a different database. To get a sense of how these features are distributed, we plotted them out. Here are for instance the first 3 features in 128-D feature vector:



Our plan was to find 2 datasets of faces, train our model by one database, and test against the other. Synthesizing faces was achieved by sampling features from the training database.

Sampling features can be broken down into 2 steps. First off, we remove the correlations between features using PCA, since we want to sample each feature independently from the rest. This stage is necessary since there are strong correlations between features. For example, if people with large eyes often have a large nose, sampling a face with large eyes and a small nose isn't accurate. Now that we have independent features, we use the "inverse sampling" method to sample each feature independently.

Conceptually, the entire sampling process can be broken down as follows

**Part 1: Creating the model to sample from:**

1. V <- Feature vectors from a database
2. V <- V – mean( V )
3. E <- PCA ( V )
4. V' <- Project V onto E
5. For feature in V'
   1. Create histogram for this feature based on V'
   2. Create a CDF based on the histogram and save

**Part 2: Sampling from the model:**

1. For face in batch\_size:
   1. For feature in face:
      1. Sample randomly from CDF and save
2. Faces <- Unproject faces from E
3. Faces <- Faces – mean( V )
4. Return Faces

Explanation:

Stage 1 refers to using the DLib library to compute the feature vectors. This step was done once (very time consuming) and was saved in CSV files under "databases/". Stage 2 is just normalization. Stages 3 and 4 are necessary in order to remove the correlations between the features so that we can sample each feature independently. Step 5 is an implementation of "inverse sampling". Stage 6 samples from the model we create in Part 1. Stages 7 and 8 undo the projection from Step 1.

**Note**: The entire sampling process is implemented in the python module face\_transform.py

# Visualizing sampled faces

Since the outputted sampled faces aren't pictures, but merely 128-D vectors, it isn't easy to visualize the face that they represent (one would have to inverse DLib's feature extraction network).

However, to get a feel for the faces that we sampled, we averaged pictures of people that were at a small distance from the sampled feature vector. Assuming the face space is considerably smooth, the features of these averages should be close the sampled features:

A screenshot of a person

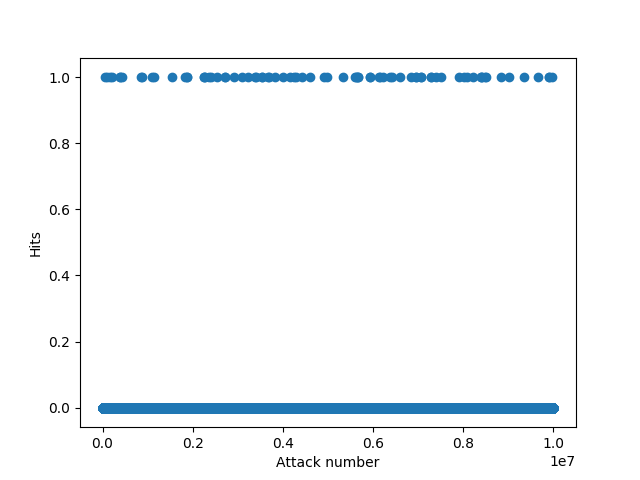
Description generated with very high confidenceA screenshot of a social media post

Description generated with very high confidence

# Results

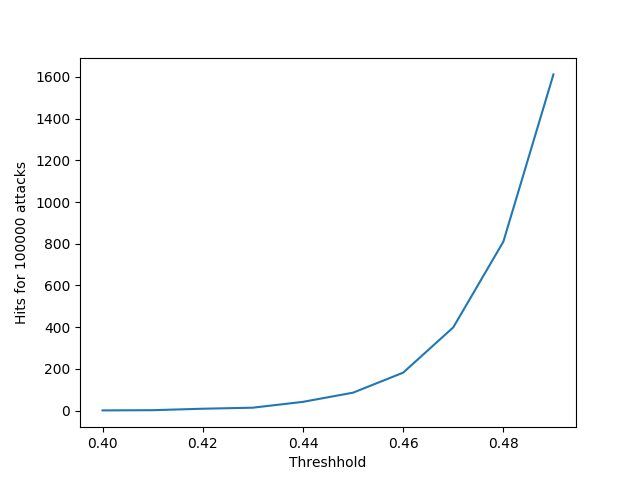
Once we were able to sample faces from the model, we were able to run several attacks on the testing database.

We found that for around 10 million synthesized faces, we had 80 hits (Around 0.001%, or 1 in 100,000 hit ratio for 0.4 threshold):

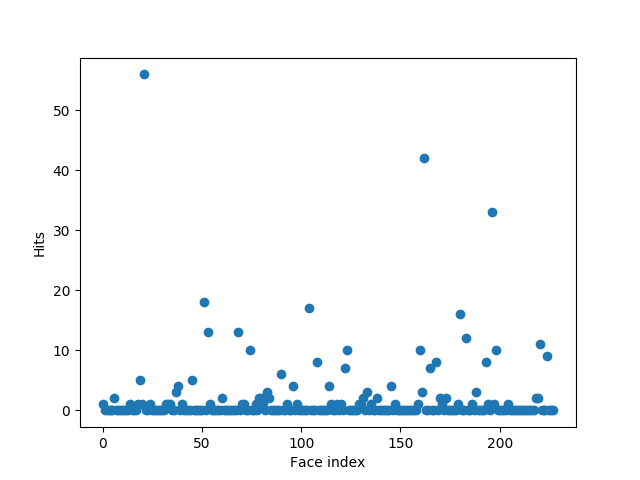
[](https://github.com/Royz2123/Biometric-Attack/blob/master/figures/diff_bases_10_mil.png)

This hit ratio was consistent also for 50 million synthesized faces, where we had 417 total hits (Details for this run can be found at general log file, from **05:06:2018\_18:41:57**). This was the largest attack that we ran.

Another interesting phenomenon is the rise in hits as a factor of the threshold. Turns out that if we increase the threshold by just a little, we suddenly reach many more hits:

[](https://github.com/Royz2123/Biometric-Attack/blob/master/figures/thresh_test4.png)

Another interesting figure that we created, was what faces were hit the most. Does an "average face" exist, that keeps getting matched all the time? We plotted the number of hits for an attack size of 50 million synthesized faces:



Here we can see that one face got over 50 hits, substantially more than the rest.

# Conclusions

In conclusion, we found that even with a harsh threshold (stricter than the one specified in the library), we still managed to synthesize faces that were close enough to ones in the database.

We conclude that if one was to use our technique as a brute force attack, his success rate would be **roughly** the same as brute-forcing a 6-7 digit code, which can be easily reached in seconds-minutes by any modern computer.

Our work helps get a better feel for the entropy of face spaces, and how faces in them are distributed.

We also found that if we would have used DLib's recommended 0.6 threshold, we would have gotten a much higher hit ratio. From this we conclude that tuning the threshold is crucial and is dependent on the database.

# System Setup

**System Organization**

Databases/ includes CSV databases that we saved throughout the project. Each CSV file contains a spreadsheet, where each row represents a face. The first column is the file location, and the next 128 columns are the faces' computed features

Dlib\_models/ includes dlib requirements

Docs/ includes our progress reports, this final document and some presentations

Faces/ Are the faces we used in this project. Note that we didn't upload our database to Github.

Figures/ contains all of the images from this final report and more.

Logs/ contains running logs from out runs. Note that this folder is split into "general/" and "seeds/". "general/" contains logs with general information, regarding, times, databases, hits and so on. "Seeds/ contains the random seed for each run, so that you can recreate a previous run (See running examples, example 4 below). Seeds are generated using the Mersenne-Twister as the core generator: <https://docs.python.org/2/library/random.html>

Src/ Contains some code samples that we used.

**Prerequisites**

You will need to download the DLib library. We chose to install Dlib using a virtual environment, for reference <https://www.learnopencv.com/install-dlib-on-ubuntu/>. Our code has been fitted to work on a virtualenv, on Ubuntu 16.04.

**Running examples**

Note: All examples assume that the setup has been set with a virtual environment as described in "prerequisites". If you have obtained Dlib in some other way, or perhaps are just running from the default csv files provided, remove the "workon" and "deactivate" lines from the bash examples.

**Example 1 - default run**

Running the code with all the default parameters is done in the following way:

workon facecourse-py3

python3 attack\_database.py

deactivate

Note that this examples has been provided as a bash file (run.sh):

**Example 2 - specifying an attack size**

workon facecourse-py3

python3 attack\_database.py --attack-size 1000000

deactivate

**Example 3 - specifying a new threshold**

For testing purposes, playing around with the threshold can be done as follows:

workon facecourse-py3

python3 attack\_database.py --threshhold 0.6

deactivate

**Example 4 - using an existing seed, from a previous run**

We make recovering a previous run super simple. Find the run that you wish to obtain it's random seed, and copy its recovery time (Either copy from filename or take first field of log file)

For Example, for the timestamp "05:06:2018\_17:12:09", the code can be run as such:

workon facecourse-py3

python3 attack\_database.py --recover-time 05:06:2018\_17:12:09

deactivate