EigenFaces

HW3 Ahron Verschleisser

Before we begin make sure you read the pdf document fully and that you did not change the order of folders or files. Keep the same order as in the zip file of HW3. For this assignment by the way, there is no need to divide the data into training and testing sets.

1 Clustering

a. Yes, the K-mediods method is more robust to outliers than the k-means algorithm in the same way that the median is more robust to outliers than the mean as long as the groups are not very small. More samples would need to be changed (be outliers) to cause the median to move greatly whereas a single very large outlier could drastically alter the mean.

b. Extrema occur where the derivative is equal to zero.

$$\frac{d}{dx}(\sum_{i=1}^{m}(x_i-\mu)^2) = \sum_{i=1}^{m}2(x_i-\mu) = 2m(\sum_{i=1}^{m}(x_i) - \sum_{i=1}^{m}(\mu)) = 2m(\sum_{i=1}^{m}(x_i) - m*\mu) = 0$$

The second derivative is positive so the extrema would be a minimum

$$\frac{d}{dx}(2m(\sum_{i=1}^{m}(x_i) - m*\mu) = \frac{d}{dx}(2m(\sum_{i=1}^{m}(x_i)) = 2m*\frac{d}{dx}(\sum_{i=1}^{m}(x_i)) = 2m*\sum_{i=1}^{m}(1) = 3m$$

If **mu** is the mean then it is equal to:

$$\mu = \frac{1}{m}\sum_{i=1}^{m}(x_i)$$

Substituting this into the expression found for the first derivative of the term above...

$$2m(\sum_{i=1}^{m}(x_i) - m*\frac{1}{m}\sum_{i=1}^{m}(x_i)) = 2m(\sum_{i=1}^{m}(x_i) - \sum_{i=1}^{m}(x_i)) = 0$$

We see that when **mu** is equal to the mean this causes the derivative to indeed equal zero and it is thus proven to be a minimum.

c. The lowest possible value for this term is 0 which would occur if all points were at mu. This term is equivilant to euclidean distance from the mean in 1D. The term is minimized when the median is used because this shortens all the distances (the smallest and largest) by at least a little and the center fully.

2 SVM

A- 2 The separation is first order, a line, and has a small margin which means a larger C (1).

B- 6 This is the result of a RBF kernel with a gamma of 1 since the area of influence is not large.

C- 3 The separation is a curve but not highly complex indicating a 2nd order polynomial was used.

D- 1 The separation is first order, aline, and has a larger margin which means a lower C (0.01).

E- 5 Since the area of influence is large this is the result of a RBF kernel with a smaller gamma of (0.2).

F- 4 The boundary is highly complex and appears to be overfitting this is the result of a high order kernel (10th order).

3 Capability of generalization

a. Regularization. Keeping the complexity of the model low to reduce overfitting.

b. 2p puts pressure towards keeping the number of parameters low while 2 ln(L) encourages parameters that meaningfully add to the model's ability to discriminate.

c. Using to many parameters would lead to overfitting the training data, and not having enough explaining power would lead to a poorly performing model.

d. We want the AIC to be low so that only meaningful parameters are included in the model. This limits the complexity by keeping only the most meaningful parameters.

4 The fun part

In [1]:
```python
import numpy as np
import itertools
from tqdm import tqdm
import pickle
import sys
import pandas as pd
import matplotlib as mpl
import seaborn as sns
import matplotlib.pyplot as plt
mpl.style.use(['ggplot'])
%matplotlib inline
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from IPython.display import display, clear_output

from PIL import Image
from sklearn.datasets import fetch_lfw_people
from sklearn.decomposition import PCA
```

We will start by loading a dataset of human faces out of `scikit-learn` datasets. If for some reason the next cell fails, you may activate the next one (which is now in comments). Make sure that the `X.npy` file is at the same location of this notebook.

In [2]:
```
# Load data
lfw_dataset = fetch_lfw_people(min_faces_per_person=0)
_, h, w = lfw_dataset.images.shape
X = lfw_dataset.data

y = lfw_dataset.target
target_names = lfw_dataset.target_names
target_names = target_names[y]
print("Dataset images are at the shape of {}X{}".format(h,w))
```

```
Dataset images are at the shape of 62X47
```

In [3]:
```
# X = np.load(''Data/X.npy')
# h = 62
# w = 47
# print("Dataset images are at the shape of {}X{}".format(h,w))
```

13,233 images were flattened and stacked in the matrix X so every row is an image and every column is a pixel. We can see that by the shape of X.

In [4]:
```
print(X.shape)
```

```
(13233, 2914)
```

Let's see some of the images of the original dataset:

In [5]:
```
def plot_gallery(images,target_names, h, w,rows=3, cols=4):
    plt.figure(figsize=(14,14))
    for i in range(rows * cols):
        plt.subplot(rows, cols, i + 1)
        plt.imshow(images[i, :].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(target_names[i])
        plt.xticks(())
        plt.yticks(())
    plt.show()
```

In [6]:
```
plot_gallery(X, target_names, h, w)
```

Before we use the `PCA` package, we have to center our data (even though `pca.fit_transform()` does it automatically, here we will do it for practicing). The center of our data (μ) is the mean of each pixel along all of our images. Thus, the center has 2,914 elements where the first element is the mean of all of the images' "first" pixels, the second is the mean of all of the images' "second" pixels etc. Notice that the images are already flattened in X which might help you.

Name your `pca` object: `pca` .

Calculate μ of X and center every image in X (row vector) according to μ. Keep X with same name `X` . Name the μ vector `mu_orig` .

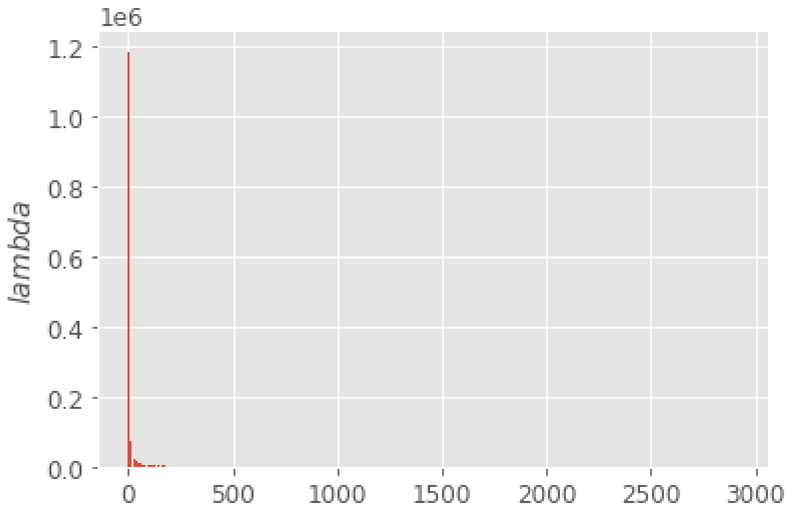
If you can, apply centering without a single loop.

In [7]:
```
#----------------------------Implement your code here-----------------------
mu_orig = np.mean(X,0)
X = X-mu_orig # centering with broadcasting
pca = PCA(n_components = 2914 , random_state = 336546)
X_pc = pca.fit_transform(X)
#---------------------------------------------------------------------------
```

Now fit a PCA model on X (**without whitening**) that would preserve 99.5% of the energy. Find out how many eigenvectors you remained with and set it as K.

In [8]:
```
#----------------------------Implement your code here-----------------------
plt.bar(range(1,len(pca.explained_variance_)+1),height= pca.explained_variance_)
plt.ylabel('$lambda$')
K = np.argmax(np.cumsum(pca.explained_variance_ratio_)>=.995) +1 # because of zer
print('I remained with ',K, 'eigenvectors.')
#---------------------------------------------------------------------------
```

```
I remained with  733 eigenvectors.
```



Question: *If we now have K eigenvectors, by how many dimensions we have reduced our data?*

Answer: n_features - K = 2914 - 733 = 2181 We have reduced our data by 2181 dimensions nearly 75% reduction.

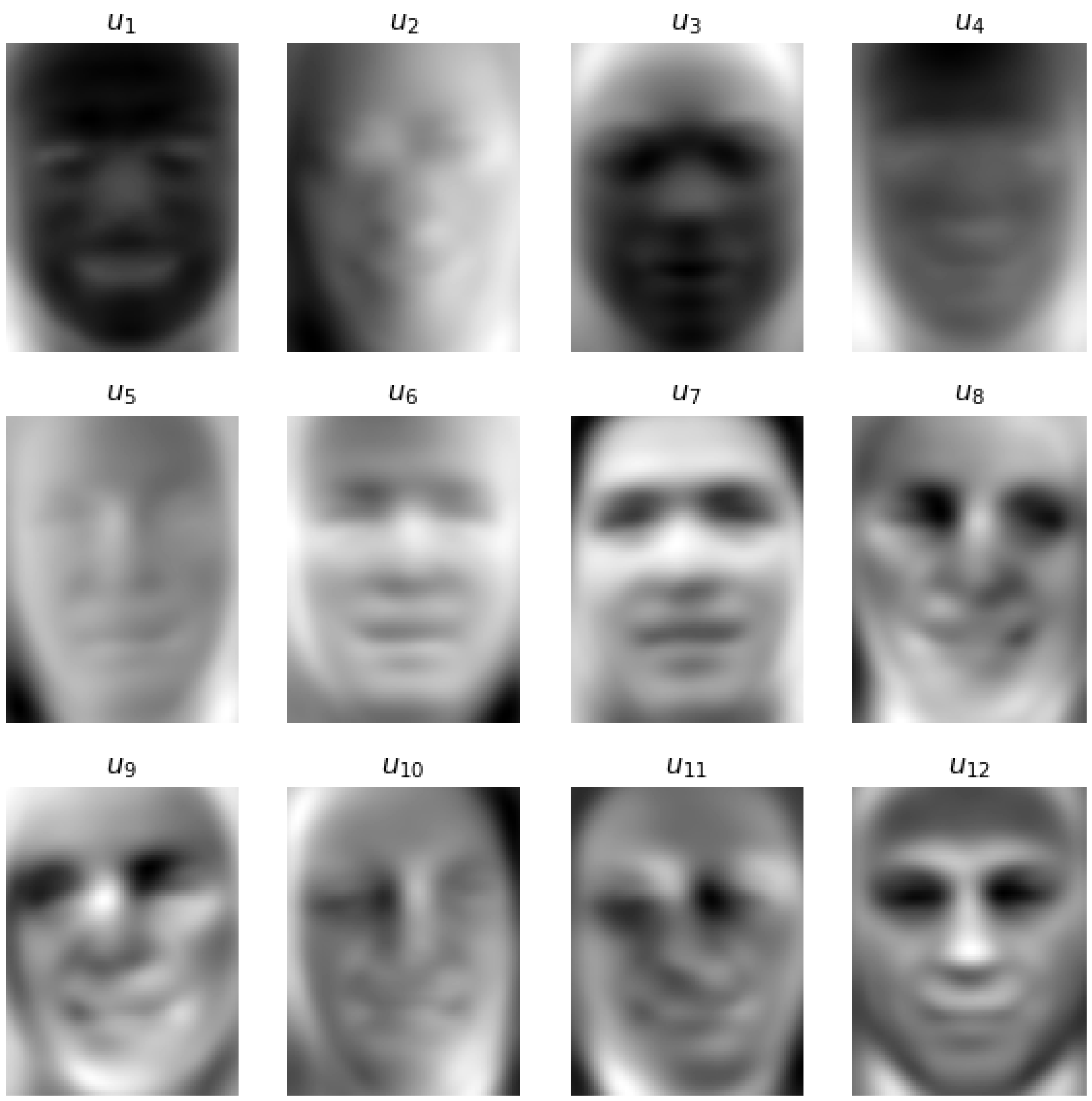
Let's see these eigenfaces!

In [9]:
```
def plot_eigenfaces(eigenvec_mat, h, w, rows=3, cols=4):
    plt.figure(figsize=(10,10))
    for i in range(rows * cols):
        plt.subplot(rows, cols, i + 1)
        plt.imshow(eigenvec_mat[i, :].reshape((h, w)), cmap=plt.cm.gray)
        plt.title("$u_{" + str(i+1) + "}$")
        plt.xticks(())
        plt.yticks(())
    plt.show()
```

`eigenvec_mat` is the matrix of the eigenvectors calculated where every row is an eigenvector and the first row has the highest eigenvalue and the second row has the next highest eigenvalue etc. Think where this matrix was calculated in the process.

In [10]:
```
print(X_pc.shape)
eigenvec_mat = pca.components_
plot_eigenfaces(eigenvec_mat, h, w)
```

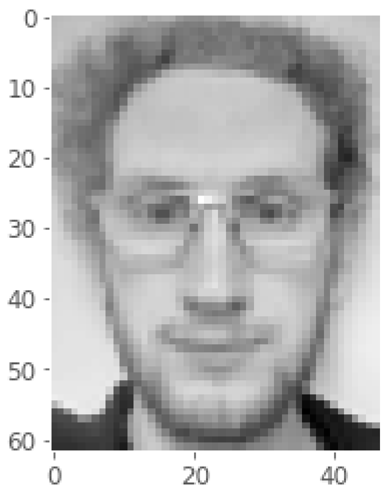
```
(13233, 2914)
```



If you got it right, you should see those "ghosts" we talked about.

And now, let's move on towards the more interesting part! We will start by showing your resized gray level face image. It might be a bit distorted due to resizing. Change the format of `.jpg` as needed.

In [11]:
```
img_format = '.jpg' # change the format according to your image. Do not delete th
image = Image.open('Data/Orig' + img_format)
#image = Image.open('Data/Tilt1' + img_format)
#image = Image.open('Data/Offcenter' + img_format)
gray = image.convert('L')
g = gray.resize((w, h))
orig = np.asarray(g).astype('float32')
plt.imshow(orig, cmap=plt.cm.gray)
plt.grid(False)
```



Now we should flatten our image and center it by the same μ vector you calculated before.

In [12]:
```
flattened_img = np.asarray(g).astype('float32').flatten()
flattened_img -= mu_orig
```

Define `U` to be the matrix containing the first K eigenvectors (rows) extracted from `eigenvec_mat`. Now, calculate the projections c_i so you would have a vector with K elements where the first element is c_1 and the second is c_2 etc. Relate to the pdf if you are not sure. **Note:**

you can only use `U` , your flattened imgae and `numpy` for this section. **Do not apply `pca` methods for calculating the projections.**

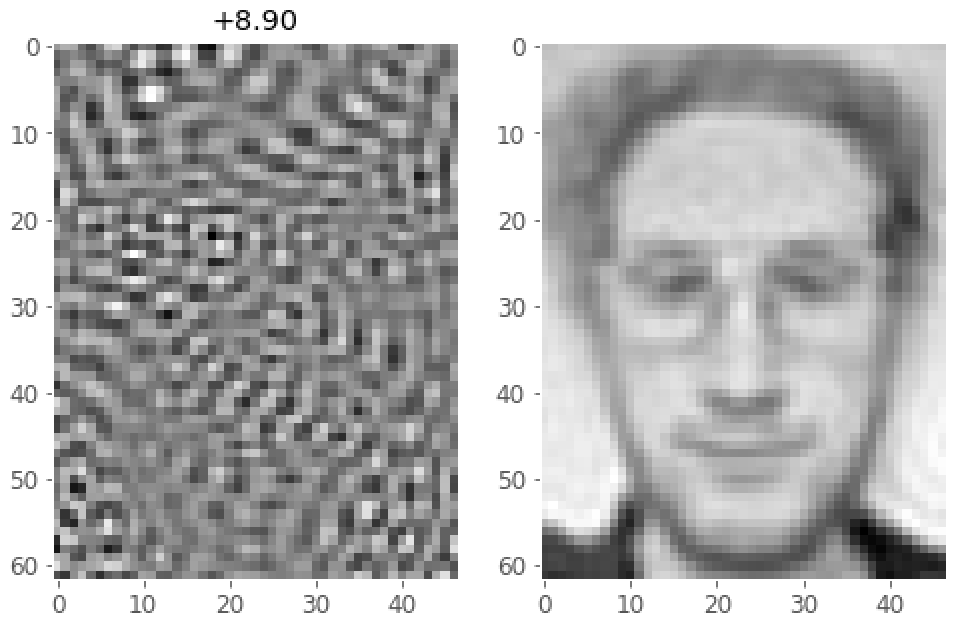
In [13]:
```
#---------------------------Implement your code here-----------------------
U = eigenvec_mat[0:K,:]
c = np.dot(U,flattened_img) # projection is simply the dot product because the ei
print(c.shape)
#------------------------------------------------------------------------
```

```
(733,)
```

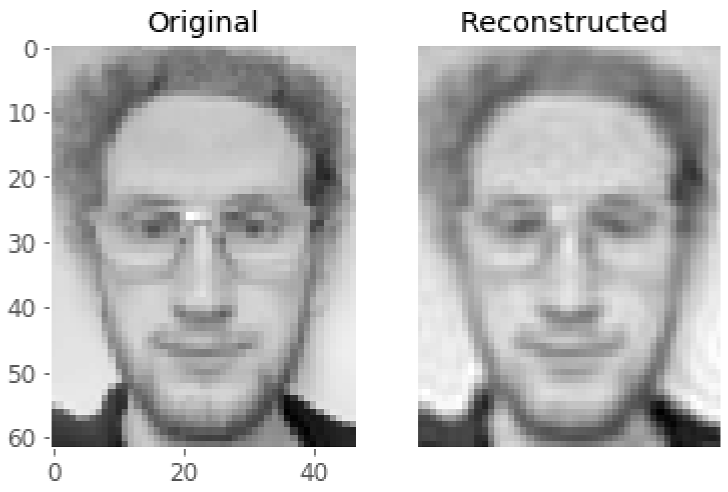
If you got it all correct, then the cell below will show you how your face is reconstructed as a linear combination of the eigenfaces which actually means that your face is a linear combination of other people's faces! The coefficients will appear with their adequate sign in the title in the left and the image constructed will appear slowly on the right.

Notice the "correction" of the centering made in the loop for visualization.

In [14]:
```
s = np.zeros((h*w,))
fig, axes = plt.subplots(1, 2,figsize=(8,8))
for j in range(K):
    s += c[j]*U[j, :]
    if np.mod(j, 10) == 0:
        axes[0].imshow(U[j, :].reshape((h, w)), cmap=plt.cm.gray)
        axes[0].grid(False)
        corrected_image = s + mu_orig
        axes[1].imshow(corrected_image.reshape((h, w)), cmap=plt.cm.gray)
        axes[1].grid(False)
        if c[j] < 0:
            axes[0].set_title('{:.2f}'.format(c[j]))
        else:
            axes[0].set_title('+{:.2f}'.format(c[j]))
        display(fig)
        clear_output(wait = True)
        plt.pause(0.3)
        if c[j] > 0:
            p = '+' + str(c[j])
        else:
            p = str(c[j])
```



In [15]:
```
fig, axes = plt.subplots(1, 2)
axes[0].imshow(orig, cmap=plt.cm.gray)
axes[0].title.set_text('Original')
axes[0].grid(False)
plt.xticks(())
plt.yticks(())
axes[1].imshow(corrected_image.reshape((h, w)), cmap=plt.cm.gray)
axes[1].title.set_text('Reconstructed ')
axes[1].grid(False)
plt.xticks(())
plt.yticks(())
plt.show()
```



Hopefully, you got a nice reconstruction from those "ghosts". Now, we will continue to the face recgonition part.

Facebook has also learned the natural basis of human faces using a database of different faces much like you did here. Now, some of your images are tagged (labeled) in a larger (mostly different) database with your name. It then projects the images on the PCA axes and gets an "id" (c coefficients) for each and every image in the tagged database. When someone uploades a picture without tagging, Facebook uses the same learned PCA in order to create this new image an id . and then looks through the tagged id database for comparison. The closest match (user) will get a notification whether or not he\she would like to be tagged according to the match.

We will make a simple simulation. First we will create a tagged database, which in our case will simply be `X` concatenated with your "Orig" flattened and centered image. We will call it `X_new` .

In [16]:
```
X_new = X.copy()
X_new = np.vstack([X_new, flattened_img])
X_new.shape
```

Out[16]:
```
(13234, 2914)
```

Notice that the number of rows was increased by 1 accordingly. Now we will add your name. Fill your name as a string below:

In [17]:
```
name = 'Ahron V'
target_new = target_names.copy()
target_new = np.append(target_names, name)
target_new.shape
```

Out[17]:
```
(13234,)
```

Let's shuffle the databse:

In [18]:
```
from sklearn.utils import shuffle
X_new, y_new = shuffle(X_new, target_new, random_state=0)
```

Now let's simulate a new uploaded untagged (out of the database) image and show it. You can use either "tilt1" or "tilt2". Then we will also center it.

In [19]:
```
img_format = '.jpg' # change the format according to your image. Do not delete th
image = Image.open('Data/tilt1' + img_format)  #change the name if needed
gray = image.convert('L')
g = gray.resize((w, h))
new_input = np.asarray(g).astype('float32')
plt.imshow(new_input, cmap=plt.cm.gray)
plt.grid(False)
```

In [20]:
```
flattened_img_new = np.asarray(g).astype('float32').flatten()
flattened_img_new -= mu_orig
```

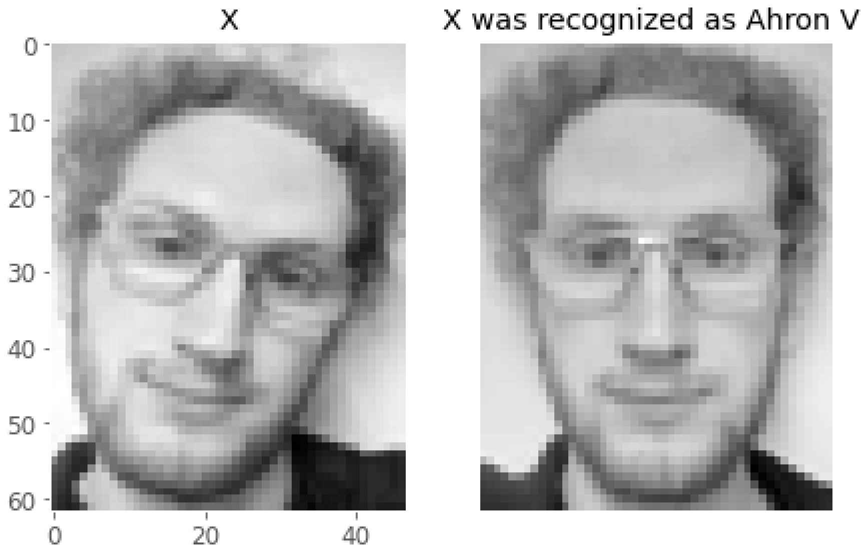
And now for the recognition part: First, project the new database onto the **already fitted** PCA basis. You can use `pca` methods. Then calculate the projections (`c` vector) for the new input similarly to what you did before. Then run in a for loop and measure the Euclidean distance between the new output projection to each and everyone of the projections of the new database. Finally return the index (row number) of the best matching image in the database (minimal distance) and name it as `idx_match` .

In [21]:
```
#----------------------------Implement your code here-----------------------
c_newdb = pca.transform(X_new)
c_newim = pca.transform(flattened_img_new.reshape(1, -1))

for idx, c in enumerate(c_newdb):
    edist = np.linalg.norm(c_newim - c)
    if idx == 0:
        closest = edist
        idx_match = idx
    elif edist<closest:
        closest = edist
        idx_match = idx
#---------------------------------------------------------------------------
```

Let's see if you got it right. If so, on left image you would see your tilted untagged image and on the right the best matching image of the database.

In [22]:
```
fig, axes = plt.subplots(1, 2, figsize=(7,7))
axes[0].imshow(new_input, cmap=plt.cm.gray)
axes[0].title.set_text('X')
axes[0].grid(False)
plt.xticks(())
plt.yticks(())
axes[1].imshow((X_new[idx_match, :] + mu_orig).reshape((62, 47)), cmap=plt.cm.gra
axes[1].title.set_text('X was recognized as {} '.format(y_new[idx_match]))
axes[1].grid(False)
plt.xticks(())
plt.yticks(())
plt.show()
```



The power of PCA is the ability to find your image across 13,324 images with using only Euclidien distance of K elements vectors rather using image matching techniques involving higher dimensional data, computational power and mostly morphlogical priors and filters! Try it also for the second image and check if it worked.

Notice that even if it failed to recognize you but the code was correct, you will get the full credit points for this section.

Hope you enjoyed :)

Note: Basically, once you run all of the Jupyter cells and save the file, then it would be saved with all of your outputs. Thus, only the pdf with your answers for the theoretical part and the notebook with the outputs should be uploaded to GitHub without your personal jpg images.