CSC411: Project 1

Due on Monday, January 29, 2018

Anas Al-Raheem

January 30, 2018

Part 1

The data-set consists of about 1420 images, 690 for the 6 actors and 730 for the 6 actresses. The images are cropped to 32 × 32-pixel images using the provided bounding boxes to include just the faces and saved in grey-scale. For each actor there is a variety of faces; different angles as well as different ages, impressions, and lighting conditions, thus not all images can be aligned together for a given actor. Most of the images are cropped correctly by the given bounding boxes, those that were not cropped correctly, were duplicated, or were missing features (such as covered mouths) were added to a discard list (the list can be found in faces.py, I created this list as I noticed lower performance using some of those images), thus not used in the program. A few examples are below:

Figure 1: original images, displaying different face angles and different ages

Figure 2: grey-scale cropped images of faces

Part 2

Using *get_images* function, all the images for a given actor or actress are uploaded into a list then the list of images is shuffled three times. The shuffled list is split into three parts, the first is the test set, the second is the validation set, and the third is the training set. The size of each set is an argument provided when calling the function.

Part 3

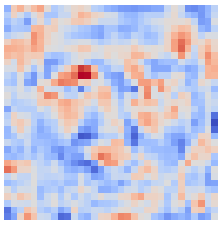
Cost function: $\sum_{i=1}^{m}((\theta^T x) - y)^2$.
The cost function value on the training set: 0.481838082244 and on the validation set: 1.75691882777
Performance on training set, validation set: 100.0% and 90.0%.
For each image i, the following code was used to determine the label of the image, 1 for baldwin and 0 for carell.

```
np.dot(theta1, i.T) > 0.5 // baldwin images, expected output is True
np.dot(theta1, i.T) < 0.5 // carell images, expected output is True
```

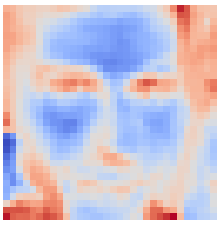
In order for the program to work, I tried different alpha values, for alpha values that are too large I would get an over flow error. Using lower values with too many iterations, I noticed the cost function value starts to increase after some point. And when trying alpha values that are too small, the program would have poor performance if the number of iterations is not increased, which increased the run time greatly. Trying different alpha values I selected the current one based on the time it takes to minimize the function as well as increase the performance of the program with the resulted theta. I also noticed starting theta0 with all values as 0.1 improved the performance faster, i.e with less iterations.

Part 4

A):
The theta images obtained by running the program on the full training set and on only 2 images per actor:



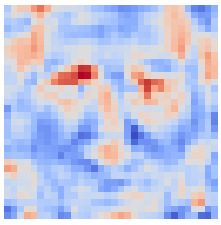
(a) running on full training set



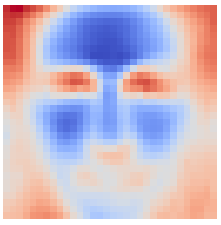
(b) running on 2 images per actor

Figure 3: theta images

B):
The theta images representing a face and random dots by running the program on the full training set:
For this part I noticed that stopping after only 5000 iteration the resulted theta is not a recognizable face, and stopping after 10 iterations would result in a face as seen below:



(a) running on full training set, stopping after 5000 iteration



(b) running on full training set, stopping after 10 iteration

Figure 4: theta images

Part 5

For this part, I ran the program, repeatedly on a changing size training set. Starting from 1 image per actor, to the minimum number of training images any actor has (the minimum is gilpin which has only 68 images for training). For each size I plot the performance of the resulted theta on the training set, validation set (of size 10), and a validation set for the 6 actors not in the given list act (50 images each). We can see in the graph below, how the performance on the training set is always 100% representing overfitting, and how it is always less on the validation sets.

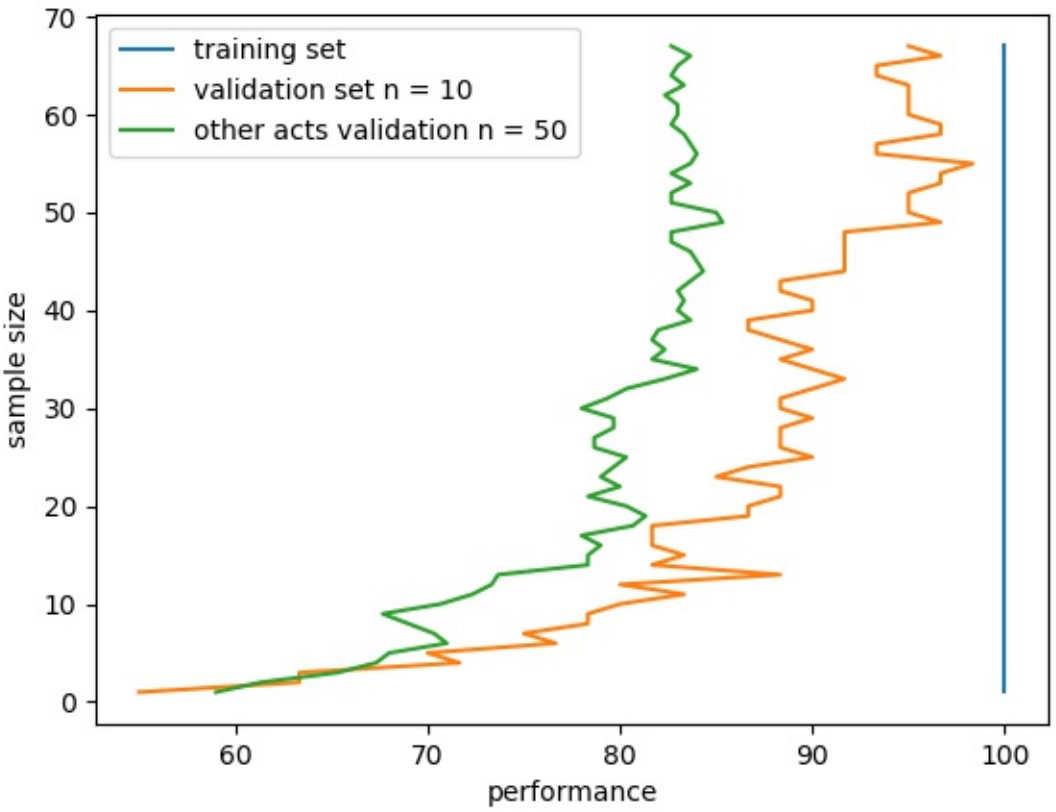


Figure 5: performance of classifiers

Part 6

A):
By looking at the cost function j, we see that for any given i we have the following:
$\sum_{j=1}^{k}((\theta^T x^i) - y^i)^2 = ((\theta_0^T x^i) - y_0^i)^2 + ((\theta_1^T x^i) - y_1^i)^2 + ... + ((\theta_k^T x^i) - y_k^i)^2$
Thus for $i = p$ and $j = q$, we have:
$((\theta_0^T x^p) - y_0^p)^2 + ((\theta_2^T x^p) - y_1^p)^2 + ... + ((\theta_q^T x^p) - y_q^p)^2 + ... + ((\theta_k^T x^p) - y_k^p)^2$
So the derivative of θ_{pq}:
$((\theta_q^T x^p) - y_q^p)^2 \frac{dj}{d\theta_{pq}} = 2((\theta_q^T x^p) - y_q^p)(((\theta_q^T x^p) - y_q^p)\frac{dj}{d\theta_{pq}}) = 2x^p((\theta_q^T x^p) - y_q^p)$.
We see that $\frac{dj}{d\theta_{pq}}$ for all other components at i = p will be 0 and similarly for all the components of sums at

other i values.
Thus $\frac{dj}{d\theta_{pq}} = 2((\theta_q^T x^p) - y_1^i)x^p$

B):
As shown in part A, for i = p we have:
$((\theta_0^T x^p) - y_0^p)^2 + ((\theta_1^T x^p) - y_1^p)^2 + ... + ((\theta_k^T x^p) - y_k^p)^2$
if we take the derivative of each component of the sum, we will have:
$(2x^p((\theta_0^T x^p) - y_0^p)) + (2x^p((\theta_1^T x^p) - y_1^p)) + ... + (2x^p((\theta_k^T x^p) - y_k^p))$
we see that each component starts with $2x^p$ thus we can take it out of the sum:
$2x^p(((\theta_0^T x^p) - y_0^p) + ((\theta_1^T x^p) - y_1^p) + ... + ((\theta_k^T x^p) - y_k^p))$
We notice that the sum above can be written as a matrix multiplication:
$2x^p(\theta^T x^p - Y^p)$, where θ here is a $n \times k$ matrix and x^p is $n \times 1$, Y^p is a $k \times 1$ matrix.
We can generalize this result for all i, the sum over all values of i is:
$(2x^0(\theta^T x^0 - Y^0) + 2x^1(\theta^T x^1 - Y^1) + ... + 2x^m(\theta^T x^m - Y^m))$
We can see that the above sum can be re written as matrix multiplication:
$2X(\theta^T X - Y)^T$, where X is a $n \times m$ matrix, θ is a $n \times k$ matrix, and Y is a $k \times m$ matrix.
m is the number of training examples, n is the number of pixels + 1, k is the number of labels.

C):
The cost function j, and it's derivative:

```
def j(x, y, theta):
    return np.sum(np.sum((np.dot(theta.T, x) - y)**2, 0))

def dj(x, y, theta):
    return 2. * np.dot(x, (np.dot(theta.T, x) - y).T)
```

D):
The code below, calculates the finite difference when changing an entry in theta. If the difference is less than 10^{-2} we consider the 2 numbers identical, and thus the correctness of the function and the derivative.

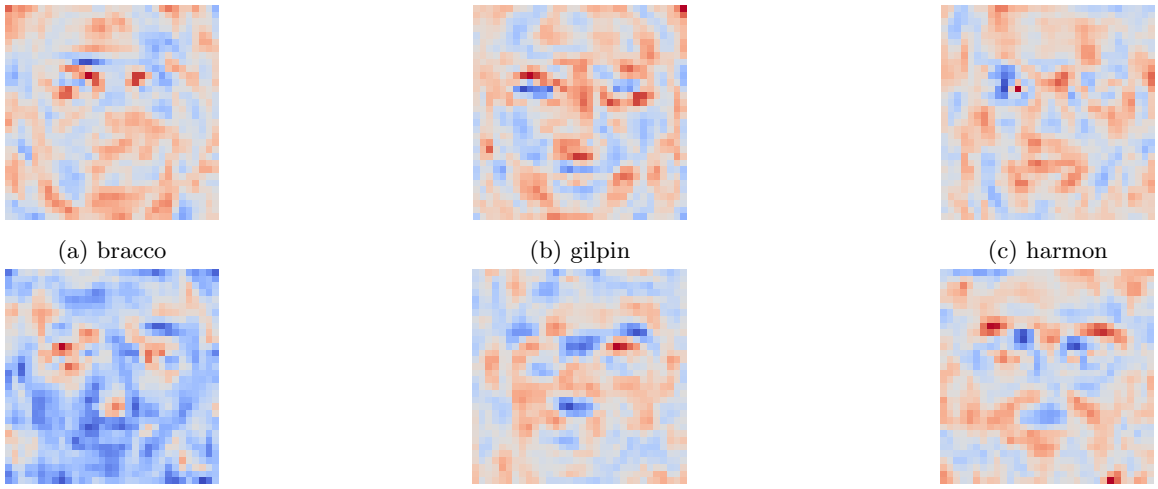
```
def test_functions(x, y, theta):

    results = []
    h = 1e-5
    for i in range(5):
        theta1 = theta.copy()
        theta2 = theta.copy()
        theta2[i, i] = theta1[i, i] + h

        result1 = (j(x, y, theta2) - j(x, y, theta1)) / (h)
        result2 = dj(x, y, theta1)
        results.append(abs(result1 == result2[i, i]) < 1e-2)
        # results.append((result1, result2[i, i]))

    return results
```

I tried using different h values and noticed that larger values cause the finite difference to increase up to 10^2, and for smaller h values the finite difference between f(x + h) and f (x) will be unnoticeable which results in f(x + h) - f(x) = 0. Thus I selected the smallest possible h value that produced actual results.

Part 7

Performance on training set: 94.0%, and validation set: 81.6666666667%.
For this part I tried different values for alpha and also different number of iterations. For a larger alpha or simply increasing iteration limit (up to 300,000) the cost function result was further reduced but the performance for certain actors would decrease as well while improve greatly for others. Thus by trying different alpha values I decided on the one I am currently using as it seemed to produce the best overall performance and with a short running time. Also initiating theta with 0.001 seemed to improve performance further, rather than starting with a greater value or randomly setting the values.
For each image the result is an array of k = 6 elements, by getting the index i of the highest value in the array we can get the label, where the label is act[i].

Part 8



(a) bracco (b) gilpin (c) harmon

(d) baldwin (e) hader (f) carell

Figure 6: theta images