CSC411: Assignment #1

Due on Monday, January 29, 2018

Chawla Dhruv

January 29, 2018

Part 1

Dataset description

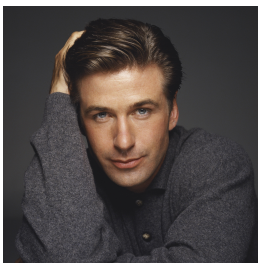
The dataset (in the `cropped` folder) is a collection of 32x32 grayscale images, which are cropped and converted from a subset of FaceScrub dataset provided on the CSC411 Project1 webpage.

The images are primarily headshots which display the actor's forehead, eyes (in some cases the actors are wearing transparent glasses), nose, mouth (can be closed, or donning a smile or a laugh) and chin.

Most of the images are shot straight on (with some exceptions where the head is turned slightly to the side) and in adequate lighting having minimal shadows on the actors' faces.

The images are labeled as `<actor_name>_<i>.jpg` where `i` is a number greater than 0 (like a counter) and `actor_name` is last name of an actor in lower case.

Some random examples from the images (cropped and uncropped) are shown



(a) Alec Baldwin Uncropped

(b) Alec Baldwin Cropped

(c) Lorraine Bracco Uncropped

(d) Lorraine Bracco Cropped

(e) Angie Harmon Uncropped

(f) Angie Harmon Cropped

Figure 1: Examples of uncropped images of 3 actors in the dataset along with their cropped images

Procedure for constructing the dataset
The two datasets from CSC411 Project1 webpage are combined into one single text file, `faces_subset.txt`. The `get_and_crop_images` function, which takes in a list of actors, finds the links for actors in `faces_subset.txt`, downloads the raw image, crops them into 32x32 using bounding boxes specified in `faces_subset.txt`, and saves them as grayscale.

A second function `remove_bad_images` can then be called to remove images with incorrect bounding boxes or poor image quality (such as blur or dark shadows) from `cropped` folder. `remove_bad_images` has a list manually chosen that removes bad images for datasets for all 12 actors in `faces_subset.txt`

Tip: Unzip cropped.zip to get all dataset images without having to bother downloading them all yourself.

Part 2

Splitting the dataset into training, validation and testing sets

The function `build_sets()` in `build_sets.py` takes in the name of an actor (example: *gilpin*) and returns three lists, `training_set`, `validation_set` and `testing_set`. `validation_set` and `testing_set` are both 10 images each while `training_set` is the remainder from the rest of the actor's dataset.

Note: All three lists are ***non-overlapping***

The lists are made using `numpy.random.shuffle()`. A seed of 5 is chosen to ensure reproducibility.

Here's an example of `build_sets` in action.

```
>>> training_set, validation_set, testing_set = build_sets('bracco')
>>> training_set
['bracco58.jpg', 'bracco55.jpg', 'bracco102.jpg', 'bracco109.jpg', 'bracco13.jpg',
 'bracco118.jpg', 'bracco78.jpg', 'bracco8.jpg', 'bracco31.jpg', 'bracco36.jpg',
 'bracco49.jpg', 'bracco119.jpg', 'bracco17.jpg', 'bracco63.jpg', 'bracco71.jpg',
 'bracco44.jpg', 'bracco73.jpg', 'bracco106.jpg', 'bracco96.jpg', 'bracco42.jpg',
 'bracco32.jpg', 'bracco68.jpg', 'bracco84.jpg', 'bracco113.jpg', 'bracco57.jpg',
 'bracco6.jpg', 'bracco27.jpg', 'bracco116.png', 'bracco41.JPG', 'bracco56.jpg',
 'bracco91.jpg', 'bracco33.jpg', 'bracco45.jpg', 'bracco83.jpg', 'bracco117.jpg',
 'bracco52.jpg', 'bracco47.jpg', 'bracco28.jpg', 'bracco103.jpg', 'bracco43.jpg',
 'bracco48.jpg', 'bracco60.jpg', 'bracco2.jpg', 'bracco54.jpg', 'bracco66.jpg',
 'bracco82.jpg', 'bracco74.jpg', 'bracco4.jpg', 'bracco76.jpg', 'bracco21.jpg',
 'bracco98.jpg', 'bracco112.jpg', 'bracco114.jpg', 'bracco53.jpg', 'bracco19.jpg',
 'bracco16.jpg', 'bracco9.jpg', 'bracco104.jpg', 'bracco30.jpg', 'bracco100.jpg',
 'bracco87.jpg', 'bracco69.jpg', 'bracco40.jpg', 'bracco10.jpg', 'bracco7.jpg',
 'bracco38.jpg', 'bracco12.jpg', 'bracco97.jpg', 'bracco108.jpg', 'bracco95.jpg',
 'bracco92.jpg', 'bracco81.jpg', 'bracco26.jpg', 'bracco46.jpg', 'bracco18.jpg',
 'bracco120.jpg', 'bracco23.jpg', 'bracco80.jpg', 'bracco61.jpg', 'bracco62.jpg',
 'bracco107.jpg', 'bracco70.jpg', 'bracco105.jpg', 'bracco20.jpeg', 'bracco22.jpg',
 'bracco101.jpg', 'bracco35.jpg', 'bracco86.jpg', 'bracco34.jpg', 'bracco85.jpg',
 'bracco29.jpg', 'bracco14.jpg', 'bracco115.jpg', 'bracco88.jpg', 'bracco67.jpg',
 'bracco24.jpg', 'bracco37.jpg']
>>> validation_set
['bracco3.jpg', 'bracco99.jpg', 'bracco111.jpg', 'bracco122.jpg', 'bracco25.jpg',
'bracco75.jpg', 'bracco94.jpg', 'bracco65.jpg', 'bracco39.jpg', 'bracco5.jpg']
>>> testing_set
['bracco0.jpg', 'bracco93.jpg', 'bracco15.jpg', 'bracco89.jpg', 'bracco79.jpg',
'bracco59.jpg', 'bracco72.jpg', 'bracco11.jpg', 'bracco1.jpg', 'bracco51.jpg']
```

Part 3

Binary Classfier: Steve Carell or Alec Baldwin?

Cost function minimized: Squared Error

Cost function value for training set: 4.12004741282
Cost function value for validation set: 1.53816055196

Performance on training set: 100% (on more than 200 images with a near equal distribution of Steve Carell and Alec Baldwin images)
Performance on validation set: 95% (on 20 images: 10 of Steve Carell and 10 of Alec Baldwin)

Alpha: 10^{-5}
Number of iterations: 5000

Here's my code for computing the output of the classifier (from `part3()` of `faces.py`)

```
correct, total, cost_fn = 0, 0, 0

for v_0 in validation_set_0:
    v_img = imread("cropped/"+v_0)
    v_img = rgb2gray(v_img)
    v_img = reshape(np.ndarray.flatten(v_img), [1, 1024])
    v_img = np.insert(v_img, 0, 1)

    prediction = dot(theta.T, v_img)

    cost_fn += (1 - prediction)**2

    if linalg.norm(prediction) > 0.5: correct += 1
    total += 1

for v_1 in validation_set_1:
    v_img = imread("cropped/"+v_1)
    v_img = rgb2gray(v_img)
    v_img = reshape(np.ndarray.flatten(v_img), [1, 1024])
    v_img = np.insert(v_img, 0, 1)

    prediction = dot(theta.T, v_img)

    cost_fn += (prediction)**2

    if linalg.norm(prediction) < 0.5: correct += 1
    total += 1

print("Validation Set Performance = "+str(correct)+"/"+str(total))
print("Cost Function value for validation set is "+str(cost_fn))
```

`validation_set_0` and `validation_set_1` are validation sets for Alec Baldwin and Steve Carell respectively. `theta` is θ trained from the training set.

A similar piece of code measures classifier performance on training sets.

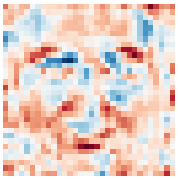
Procedure to fine-tune system performance

First I started of with 50,000 iterations and $\alpha = 10^{-7}$. I increased the number of iterations until I observed the cost function for validation sets increasing. I increased α to 10^{-6} and reduced iterations to bring down the validation set cost. I continued this process of increasing α and reducing iterations until the cost function of training set started to increase and performance started going down. This is where I stopped my process and got my presented classifier performance and cost values.

If α was made too large, the python compiler would throw an overflow error in the calculation of the dot product between θ transpose and α.

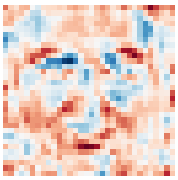
Part 4

Displaying thetas



4(a): Full training set

4(a): Reduced training set with 2 images of each actor



4(b): Gradient descent run with 5,000 iterations

4(b): Gradient descent run with only 10 iterations

Figure 2: Displaying θ using combination of full or reduced training set and number of iterations

As can be seen, training with less iterations and with less training examples results in the θs resembling actual human faces more. This can be explained as training with more iterations and more training examples goes into the territory of overfitting and spurious details start showing up and we lose regular features.

The above functionality is present in `part4()` *of* `faces.py`

Part 5

Gender Classification: Overfitting

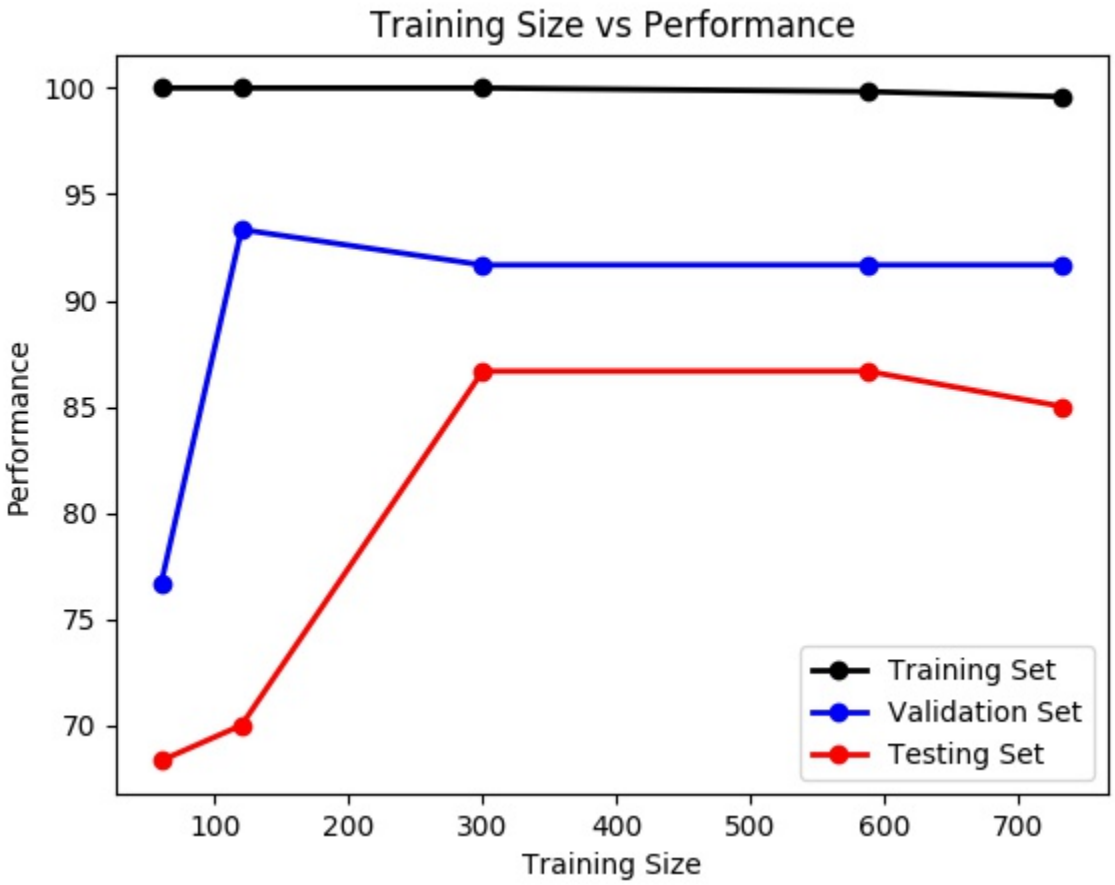


Figure 3: Performance (in %) on training, validation and test sets as size of training set increases

Parameters

Alpha: 10^{-6}
Iterations: $80,000$

The performance on the training set stays at around 100% while validation set performance reaches its maximum at around 120 training images and then drops by 2% while the performance on test set starts dropping after 600 training images.

The reduction in performance for the testing and validation set as the number of training examples increase clearly demonstrates overfitting.

The above functionality is present in `part5()` *of* `faces.py`

Part 6

Multi Classification

Part A

$$J(\theta) = \sum_i (\sum_j (\theta^T x^{(i)} - y^{(i)})_j^2)$$

$$\begin{aligned}\frac{\partial J}{\partial \theta_{pq}} &= \sum_i \frac{\partial}{\partial \theta_{pq}} (\theta_0^{(i)} x_0^{(i)} + ... + \theta_q^{(i)} x_q^{(i)} + ... + \theta_k^{(i)} x_k^{(i)} - y_0^{(i)} - ... - y_q^{(i)} - ... - y_k^{(i)})^2 \\ &= \frac{\partial}{\partial \theta_{pq}} (\theta_0^{(p)} x_0^{(p)} + ... + \theta_q^{(p)} x_q^{(p)} + ... + \theta_k^{(p)} x_k^{(p)} - y_0^{(p)} - ... - y_q^{(p)} - ... - y_k^{(p)})^2 \\ &= 2x_q^{(p)} (\theta_q^{(p)} x_q^{(p)} - y_q^{(p)})\end{aligned} \tag{1}$$

Part B

$$\frac{\partial J}{\partial \theta} = \begin{bmatrix} \frac{\partial J}{\partial \theta_{11}} & \frac{\partial J}{\partial \theta_{12}} & \cdots & \frac{\partial J}{\partial \theta_{1k}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial \theta_{n1}} & \frac{\partial J}{\partial \theta_{n2}} & \cdots & \frac{\partial J}{\partial \theta_{nk}} \end{bmatrix} \tag{2}$$

$$\frac{\partial J}{\partial \theta} = \begin{bmatrix} 2x_1(\theta^T x - y)_1 & 2x_1(\theta^T x - y)_2 & \dots & 2x_1(\theta^T x - y)_k \\ \vdots & \vdots & \ddots & \vdots \\ 2x_n(\theta^T x - y)_1 & 2x_n(\theta^T x - y)_2 & \dots & 2x_n(\theta^T x - y)_k \end{bmatrix} \tag{3}$$

$$\frac{\partial J}{\partial \theta} = 2 \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} (\theta^T x - y)_1 & (\theta^T x - y)_2 & (\theta^T x - y)_k \end{bmatrix} \tag{4}$$

$$\frac{\partial J}{\partial \theta} = 2X(\theta^T X - Y)^T \tag{5}$$

Part C

The cost function from 6(a) is as below:

```
def f_multiclass(x, y, theta):
    x = np.transpose(x)
    x = vstack((ones((1, x.shape[1])), x))
    return sum( (y - dot(x.T, theta.T)) ** 2)
```

The code for vectorized gradient function is:

```
def df_multiclass(x, y, theta):
    x = np.transpose(x)
    x = vstack( (ones((1, x.shape[1])), x))
    return 2*(dot(x, (dot(theta, x)).T-y)).T

def grad_descent_multiclass(f, df, x, y, init_t, alpha, max_iter = 20000):
    EPS = 1e-10
    prev_t = init_t-10*EPS
```

```
    t = init_t.copy()
    ite  = 0

    while norm(t - prev_t) >  EPS and ite < max_iter:
        prev_t = t.copy()
        t -= alpha*df(x, y, t)
        if ite % 5000 == 0 or ite == max_iter-1:
            print "Iter", ite
            print "Gradient: ", df(x, y, t), "\n"
        ite += 1
    return t
```

The above functions are present in `calculus.py`

Part D

Finite Difference Approximation

The code for this part is in `part6()` in `faces.py` and is reproduced here.

```
# To build x and y for training set, let's pick a picture first
actor = 'Lorraine Bracco'
a_name = actor.split()[1].lower()
training_set, _1, _2 = build_sets(a_name)

x_img = imread("cropped/"+training_set[0])
x_img = rgb2gray(x_img)

x = reshape(np.ndarray.flatten(x_img), [1, 1024])
y = np.zeros((1, 6))
y[0][1] = 1

np.random.seed(5)
theta = np.random.rand(6, 1025)

# h value
h = 0.000001

total_elements = theta.shape[0] * theta.shape[1]
difference = 0

for i in range(theta.shape[0]):
    for j in range(theta.shape[1]):
        theta_h = np.zeros((6, 1025))
        theta_h[i][j] = h

        difference += abs(((f_multiclass(x, y, theta + theta_h) - f_multiclass(x, y, theta))/h) -
                                                        df_multiclass(x, y, theta)[i][j])

print("Average difference in approximation: " + str(difference/total_elements))
```

The value of `h` was chosen to be 10^{-6} as it was very comparable to the value of α which ensures that the interval for calculating the derivative's approximation was finite enough to achieve a good accuracy (the target in this case was a 5 significant figure accuracy).

The achieved accuracy was $3.822\,978\,570\,34 \times 10^{-5}$ per element of θ

Part 7

Multi face recognition

Performance on training set: 94%
Performance on validation set: 71.67%

Alpha: 10^{-7}
Number of iterations: 150,000

A small value of α was chosen to obtain more accurate classifier performance and number of iterations were chosen to the same effect while trying to avoid overfitting the model.

Obtaining label from output of model

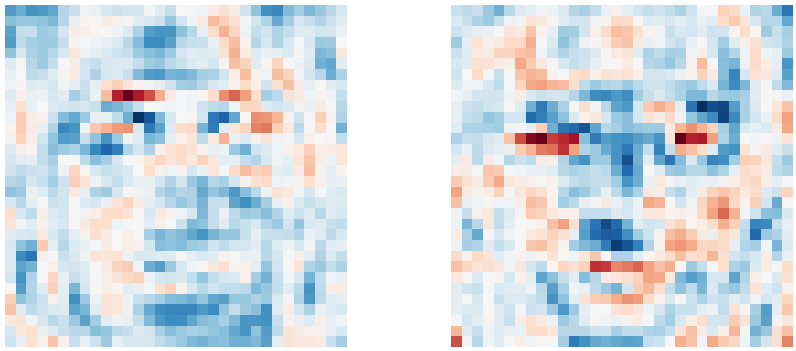
To obtain label from the output of the model, we find the index of maximum element in `numpy.dot(theta, x)` and find the label that correspronds to that index in the training set.

For example, if `numpy.dot(theta, x)` comes out to be `[0.1, 0.2, 0.38, 0.83, 0.56, 0.17]`, the 3rd index has the highest element which corresponds to Alec Baldwin.

The above functionality is present in `part7()` *of* `faces.py`

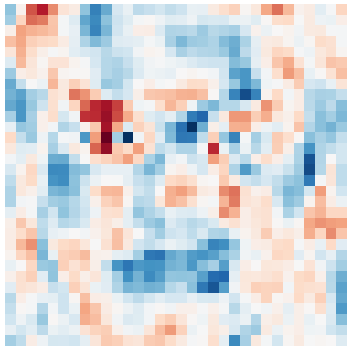
Part 8

Visualising thetas for different actors

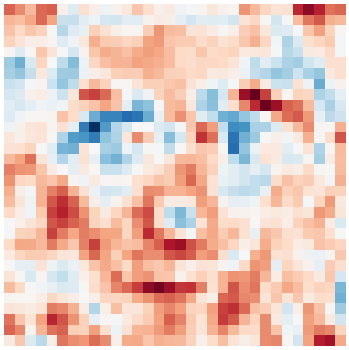


(a) Lorraine Bracco

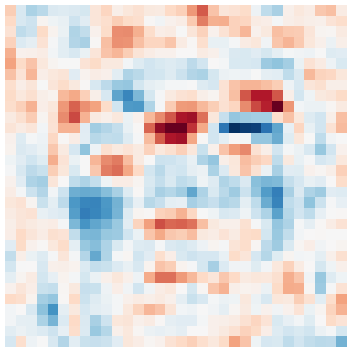
(b) Peri Gilpin



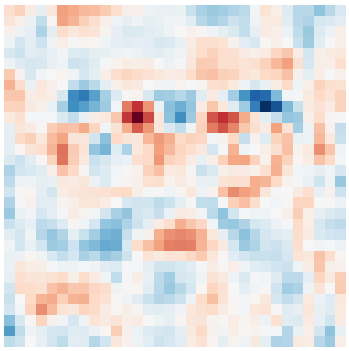
(c) Angie Harmon



(d) Alec Baldwin



(e) Bill Hader



(f) Steve Carell

Figure 4: Visualising θs for different actors in the training set

The above functionality is present in `part8()` *of* `faces.py`