CSC411: Assignment 1

Due on Monday, January 29, 2018

Weixin Liu

January 29, 2018

Problem 0

Brief View of dataset and how to download and process them
The dataset used in the project is a subset of FaceScrub dataset. (http://vintage.winklerbros.net/facescrub.html). This dataset is provided in form of two text files: facescrub_actors.txt and facescrub_actresses.txt. They contain the information of male actors and female actors respectively. Every line in the text file contains all the information of one particular image. The following line is a example line:

Lorraine Bracco 20815 9613 http://upload.wikimedia.org/wikipedia/commons/1/1b/Lorraine_Bracco_HS_Yearbook.jpeg 22,62,135,175 d920e0af8b560d8d9df41baa422314abf24a8c729ed6bab9324e8e15f3e4d6b2

"Lorraine Bracco" is the actor's name. "http://upload.wikimedia.org/wikipedia/commons/1/1b/Lorraine_Bracco_HS_Yearbook.jpeg" is a URL where the image can be download from. "22,62,135,175" is the bounding box of the face, which is given in such format: x_1, y_1, x_2, y_2 where (x_1, y_1) is the coordinate of the upper left corner of the bounding box and (x_2, y_2) is the coordinate of the lower right corner.

The images can be either gray-scale image or color image and may have various sizes. For this project, we need to crop out the images of the faces given the coordinates of the bounding boxes, convert them to grayscale if the image is color, and then, resize them to 32×32. Note that some URLs were dead at the time when downloading them.
The code for downloading and process images is pasted below. The code will download first according to two text files and stores the original images into the sub-folders under the directory of which the python file is located. The sub-folders are facescrub_actors_cropped and facescrub_actresses_cropped. Then it will process the downloaded images in the foregoing fashion and stores them in other two sub-folders: facescrub_actors_cropped and facescrub_actresses_cropped. The code is defined as a function. To download and process images, simply call function *part*(). Note: the variable *dirpath* is pre-defined in the main block, and it is set to the directory of which the python file is located.

```
def rgb2gray(rgb):
    '''Return the grayscale version of the RGB image rgb as a 2D numpy array
    whose range is 0..1
    Arguments:
    rgb -- an RGB image, represented as a numpy array of size n x m x 3. The
    range of the values is 0..255
    '''

    r, g, b = rgb[:, :, 0], rgb[:, :, 1], rgb[:, :, 2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b

    return gray / 255.


def timeout(func, args=(), kwargs={}, timeout_duration=1, default=None):
    '''From:
    http://code.activestate.com/recipes/473878-timeout-function-using-threading/'''
    import threading
    class InterruptableThread(threading.Thread):
        def __init__(self):
            threading.Thread.__init__(self)
            self.result = None
```

```
        def run(self):
            try:
                self.result = func(*args, **kwargs)
            except:
                self.result = default

    it = InterruptableThread()
    it.start()
    it.join(timeout_duration)
    if it.isAlive():
        return False
    else:
        return it.result



def part1():
    for act_type in ['actors', 'actresses']:
        faces_subset = dirpath + "/facescrub_" + act_type + ".txt"
        uncropped_path = dirpath + "/facescrub_" + act_type + "_uncropped/"
        cropped_path = dirpath + "/facescrub_" + act_type + "_cropped/"
        if os.path.isdir(uncropped_path):
            os.unlink(uncropped_path)
        else:
            os.makedirs(uncropped_path)

        if os.path.isdir(cropped_path):
            os.unlink(cropped_path)
        else:
            os.makedirs(cropped_path)

        testfile = urllib.URLopener()

        act = list(set([a.split("\t")[0] for a in open(faces_subset).readlines()]))

        # Note: you need to create the uncropped folder first in order
        # for this to work

        for a in act:
            # name = a.split()[1].lower()
            name = a
            i = 0
            for line in open(faces_subset):
                if a in line:
                    filename = name + str(i) + '.' + line.split()[4].split('.')[-1]
                    # A version without timeout (uncomment in case you need to
                    # unsupress exceptions, which timeout() does)
                    # testfile.retrieve(line.split()[4], "uncropped/"+filename)
                    # timeout is used to stop
                    # downloading images which take too long to download
                    timeout(testfile.retrieve, (line.split()[4], uncropped_path + \
                    filename), {}, 5)
                    if not os.path.isfile(uncropped_path + filename):
```

```
                    continue

                try:
                    uncropped = imread(uncropped_path + filename)
                    crop_coor = line.split("\t")[4].split(",")
                    if (uncropped.ndim) == 3:
                        cropped = uncropped[int(crop_coor[1]):int(crop_coor[3]),
                                  int(crop_coor[0]):int(crop_coor[2]), :]
                        resized = imresize(cropped, (32, 32))
                        grayed = rgb2gray(resized)
                    elif (uncropped.ndim) == 2:
                        cropped = uncropped[int(crop_coor[1]):int(crop_coor[3]),
                                  int(crop_coor[0]):int(crop_coor[2])]
                        resized = imresize(cropped, (32, 32))
                        grayed = resized / 255.
                except:
                    continue



                imsave(cropped_path + filename, grayed)

                print filename
                i += 1
        return
```

Problem 1

Dataset Description The images vary in size, color and number of actors whom the images comprise. Most of the original images are color images, a few of them are grayscale images. And most images consist of only one actor/actress, a few images consist of multiple actors. The following figures are some example of the dataset.

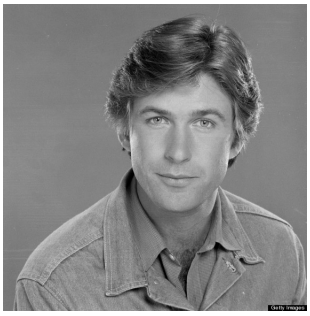

Figure 1: Grayscale image

Figure 2: Color image

Figure 3: image contain multiple actors

However, there are some images that are auto-generated due to dead URLs. Some examples are shown below:

Figure 4: Dead URL1

Figure 5: Dead URL2

Most bounding box are accurate and the cropped-out faces do not necessarily align to each other as some are side face and some are frontal face. Some examples are shown below:

Figure 6: Side Face1

Figure 7: Side Face2

Figure 8: Frontal Face1

Figure 9: Frontal Face2

However, few images' bounding box are incorrect. Those incorrect bounding box may cause disturbance on our training. Some examples are shown below:

Figure 10: uncropped image 1 with incorrect box

Figure 11: cropped image 1 with incorrect box

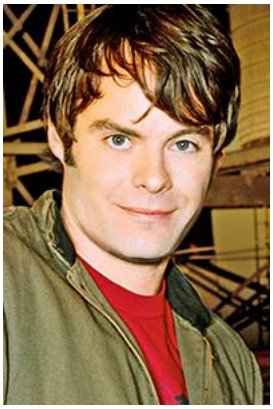

Figure 12: uncropped image 2 with incorrect box

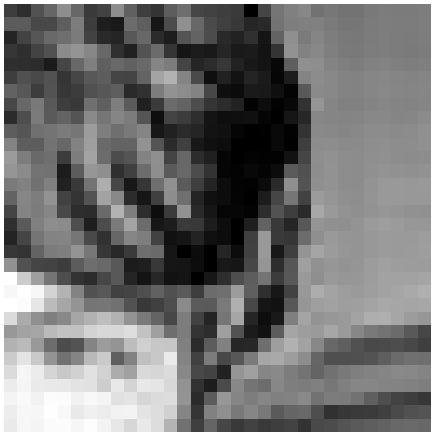

Figure 13: cropped image 2 with incorrect box

Problem 2

Randomly Choose Training Sets, Validation Sets, and Test Sets The code for this part is defined as function *part2*(). This function return a dictionary named *act_data* whose keys are the actor's name, and the corresponding value is a list of array with length of 3, where the first element is a array that contains the training set, the second element is the array that contains the validation set and the third element is the array that contains the test set.

This function first calls the function *get_names*() which return a list that contains the actor's name appeared in the dataset. Then, the function load all the cropped images for each actor into a dictionary named *act_rawdata*, and the dictionary has a following structure: the key is the actor's name and the corresponding value is a array that contains all the cropped images for that specific actor. Note that when loading the cropped images, the cropped images is flatten into a column vector. After loading the cropped images, it shuffles the columns of images array for each actors. For the sake of reproducibility, *numpy.random.seed*() is implemented to shuffle the columns. Since there may be the cases where the total number of images available for one actor is less than 90, we let the the size of training data be $min(70, numberofimagesavilable-20)$. So for each actor, the corresponding training set which is stored in $act\_data[< actor'sname >][0]$ contains the 0th to $min(70, numberofimagesavilable - 20)$th columns of the shuffled array $act\_rawdata[< actor'sname >]$. The validation set contains the -20th to -10th columns of the shuffled array, and the test set contains the -10th to the -1st columns of the shuffled array.

The code is pasted below.

```
def get_names():
    NameList = []

    for i in ['actors', 'actresses']:
        act_type = i
        dataset_path = dirpath + "/facescrub_" + act_type + "_cropped/"
        for root, dirs, files in os.walk(dataset_path):
            for filename in files:
                temp = filename.rfind('.') - 1
                while filename[temp].isdigit():
                    temp -= 1
                if filename[:temp + 1] not in NameList:
                    NameList.append(filename[:temp + 1])

    return NameList


def part2():
    act = get_names()
    act_rawdata = {}
    act_data = {}
    for i in act:
        act_rawdata[i] = np.empty(1024)
        act_data[i] = [0, 0, 0]

    for j in ['actors', 'actresses']:
        act_type = j
        dataset_path = dirpath + "/facescrub_" + act_type + "_cropped/"
        for root, dirs, files in os.walk(dataset_path):
```

```
            for filename in files:
                im = imread(dataset_path + filename)
                im = im / 255.
                im = im.flatten()
                for i in act:
                    if i in filename:
                        act_rawdata[i] = np.vstack((act_rawdata[i], im))

    # randomly shuffle act_rawdata
    np.random.seed(0)
    for i in act:
        temp = act_rawdata[i]
        np.random.shuffle(temp)
        act_rawdata[i] = temp.T

    for i in act:
        act_data[i][0] = act_rawdata[i][:, :min(70, act_rawdata[i].shape[1] - 20)]
        act_data[i][1] = act_rawdata[i][:, -20:-10]
        act_data[i][2] = act_rawdata[i][:, -10:]
    return act_data
```

Problem 3

Linear Regression Classifier to distinguish Alec Baldwin and Steve Carell
In this part, linear regression is used to classify whether a image is Alec Baldwin or Steve Carell. So quadratic loss is used as the cost function:

$$\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

The label of Alec Baldwin is set to 1 and the label of Steve Carell is set to -1. The prediction of images is obtained as:

- if hypothesis $h_\theta(x)$ is greater than 0, then predict x having label of 1, which mean this image is Alec Baldwin in this case.
- if hypothesis $h_\theta(x)$ is less than 0, then predict x having label of -1, which mean this image is Steve Carell in this case.

The performance is reported by dividing the number of images being correctly classified by the number of total images being classified.

In order to get the linear regression to work, the learning rate α need to be no too large, otherwise the algorithm would not converge. However, too small learning rate will lead to slow convergence. Through the function *part3_alpha*(), we can obtain the largest alpha that will lead to divergence. This function tries several different learning rates and plot cost function value vs. number of iterations using different learning rates. The result is shown below in Figure 14. From the plot, alpha of 0.000022 leads to divergence. So selecting alpha as 0.00001 which is the largest number that will not lead to divergence to faster the gradient descent.

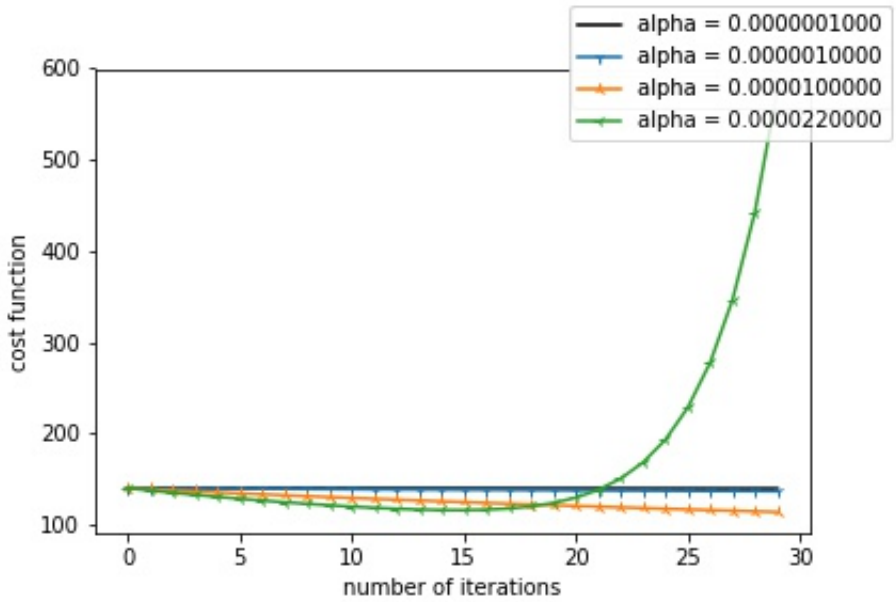

Figure 14: cost function value vs. number of iterations using different learning rates

With initial thetas being all zeros, learning rate *alpha* being 0.00001, maximum number of iteration being 80000 which is chosen such that there is no early stop occurred, and stopping condition EPS being 0.00001

which is given in the code provided in CSC411, the result is as following:

This is the result for part3
('Cost function on training set:', 0.20306443574704594, 'Normalized:', 0.0014504602553360424)
('Cost function on validation set:', 11.490984362135935, 'Normalized:', 0.5745492181067967)
('Performance on training set', 1.0)
('Performance on validation set', 0.85)

The code used in this part is pasted below:
Note: *f* is to calculate the cost function, *df* is to calculate the gradient, *grad_descent* is to function for gradient descent. Those three functions is modified from the code provided in class. *performance* is to calculate the performance of the theta in one particular dataset. *part3* consists of setting up training set, validation set, and test set, initializing theta, calling function *grad_descent* and calculating the performance.

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


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


def performance(X, Y, theta):
    X = vstack((np.ones((1, X.shape[1])), X))
    h = dot(theta.T, X)
    cor = 0
    for i in range(len(Y)):
        if Y[i] == 1 and h[i] > 0:
            cor += 1
        elif Y[i] == -1 and h[i] < 0:
            cor += 1
    return float(cor) / len(Y)


def grad_descent(f, df, x, y, init_t, alpha, EPS=1e-5, max_iter=80000):
    prev_t = init_t - 10 * EPS
    t = init_t.copy()
    iter = 0
    cost_func_ = []
    while norm(t - prev_t) > EPS and iter < max_iter:
        cost_func_.append(f(x,y,t))
        prev_t = t.copy()
        t -= alpha * df(x, y, t)
        if iter % 500 == 0:
            print "Iter", iter
            print "Cost", f(x, y, t)
            print "Gradient: ", df(x, y, t), "\n"
        iter += 1
    return t, cost_func_

def part3(alpha= 0.000010, st_devi = 0, max_iteration = 80000):
```

```
    X_train = np.hstack((act_data['Alec Baldwin'][0], act_data['Steve Carell'][0]))
    Y_train = np.append(np.ones(act_data['Alec Baldwin'][0].shape[1]),
                        np.full(act_data['Steve Carell'][0].shape[1], -1))
    X_vali = np.hstack((act_data['Alec Baldwin'][1], act_data['Steve Carell'][1]))
    Y_vali = np.append(np.ones(10), np.full(10, -1))
    X_test = np.hstack((act_data['Alec Baldwin'][2], act_data['Steve Carell'][2]))
    Y_test = np.append(np.ones(10), np.full(10, -1))

    np.random.seed(0)
    theta0 = np.random.normal(scale=st_devi, size=1025)
    theta = grad_descent(f, df, X_train, Y_train, theta0,\
     alpha, max_iter = max_iteration)

    print("This is the result for part3")
    print("Cost function on training set:", f(X_train, Y_train, theta),\
     "Normalized:", f(X_train, Y_train, theta)/(Y_train.shape[0]))
    print("Cost function on validation set:", f(X_vali, Y_vali, theta),\
     "Normalized:", f(X_vali, Y_vali, theta)/(Y_vali.shape[0]))
    print("Performance on training set", performance(X_train, Y_train, theta))
    print("Performance on validation set", performance(X_vali, Y_vali, theta))

    return theta

def part3_alpha():
    X_train = np.hstack((act_data['Alec Baldwin'][0], act_data['Steve Carell'][0]))
    Y_train = np.append(np.ones(act_data['Alec Baldwin'][0].shape[1]),
                        np.full(act_data['Steve Carell'][0].shape[1], -1))
    X_vali = np.hstack((act_data['Alec Baldwin'][1], act_data['Steve Carell'][1]))
    Y_vali = np.append(np.ones(10), np.full(10, -1))
    X_test = np.hstack((act_data['Alec Baldwin'][2], act_data['Steve Carell'][2]))
    Y_test = np.append(np.ones(10), np.full(10, -1))

    fig = plt.figure(31)
    alpha_ = [1e-7, 1e-6, 1e-5, 2.2e-5]
    for j, i in zip(range(len(alpha_)), alpha_):
        np.random.seed(0)
        theta0 = np.random.normal(scale=0, size=1025)
        theta, cost_func_ = grad_descent(f, df, X_train, Y_train,\
         theta0, i, max_iter = 30)
        plt.plot(range(0,30), cost_func_, "-%i"%j, label = "alpha = %010.10f"%i)
        plt.xlabel("number of iterations")
        plt.ylabel('cost function')
        fig.legend(loc = "upper right")
    plt.show()
    fig.savefig(dirpath + '/part3_1.jpg')

    return
```

Problem 4

Display Theta

a)

The figures shown below are generated by calling the function *part4a*(). This function removes the off-set θ_0 and resizes the theta back to 32×32. The left figure shows the thetas obtained by using the full training set, the right figure shows the thetas obtained by using only two images of each actors. The right figure contains a obvious face, which indicates overfitting problem. Since the size of training set is 4 and size of features is 1025 in this case, overfitting can easily occur. On the other hands, as the size of training set increases, the overfitting problem is mitigated as there is no obvious face in the left figure.

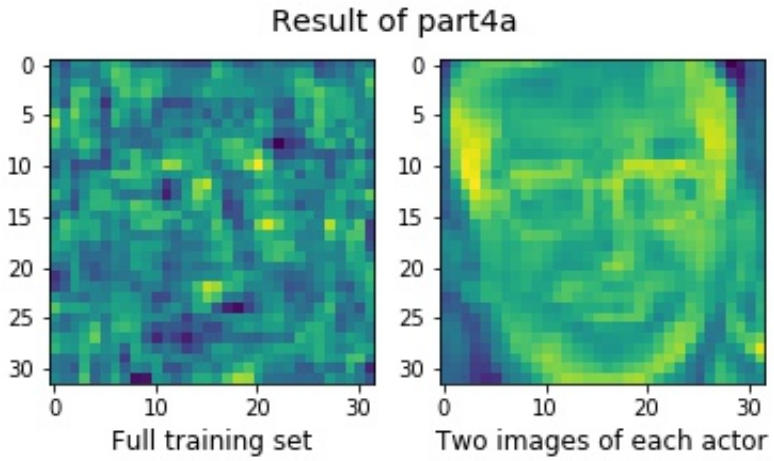

Figure 15: result of part4a)

b)

Note: The images in this section show the theta generated by initializing thetas by drawing it from a normal distribution with mean of 0 and standard deviation of std_deviation (stated in the title of each image), and the maximum iteration being max_itera (stated in the title of each image).

In order to generate a face-like image, we need to initialize theta closed to all zeros and set the maximum iteration to be small. In this case, the initial thetas are closed to the optimal point and early stop occurs. However, if the initial thetas are not closed to zeros and early stop occurs, the image of thetas does not contain any obvious images. Figure 16 shows the image with obvious face and Figure 17 shows the image without obvious face.

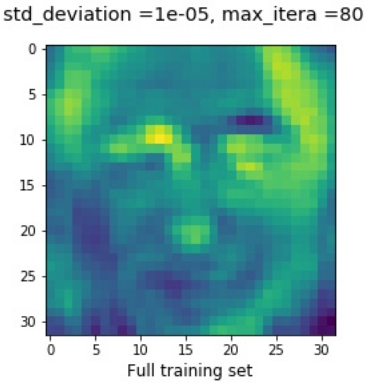

Figure 16: image with obvious face

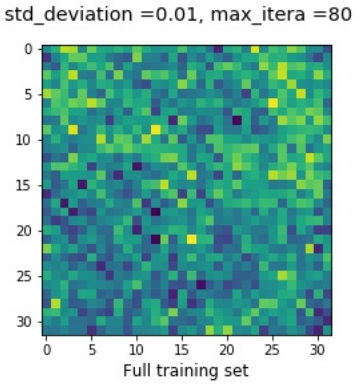

Figure 17: image without obvious face

To obtain back the left figure in Figure 15, no matter what initial theta is, as long as early stop does not occur and learning rate is small such that the algorithm converges, because the cost function, in this case, is a convex function. Convex function indicates that the local minimum is the global minimum. The images of theta generated by different initial thetas are shown below:

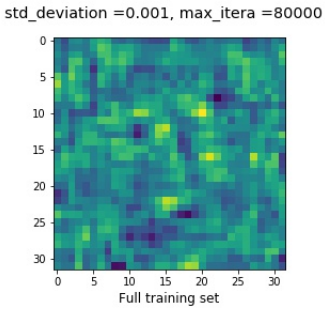

Figure 18

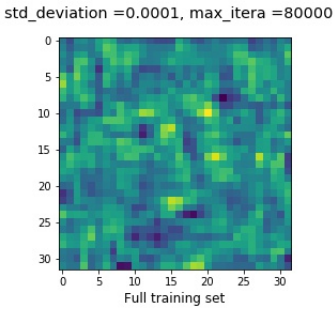

Figure 19

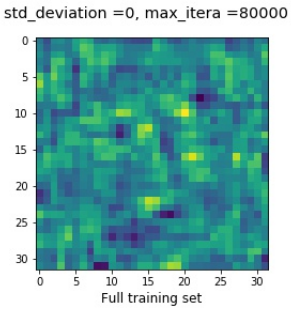

Figure 20

Problem 5

Classify Gender

In this question, we need to investigate how the size of the training affects the performance on the training and validation sets for the 6 actors in *act* and on the validation sets for the 6 actors who are not in *act*. The size of training set ranges from 10% to 100% of the full training set. The parameter used in this problem such as learning rate, initial theta value, etc., are the same as those parameters used in part3. The plot of the performance on n the training and validation sets for the 6 actors in *act* and on the validation sets for the 6 actors who are not in *act* vs. the size of the training set is shown below.

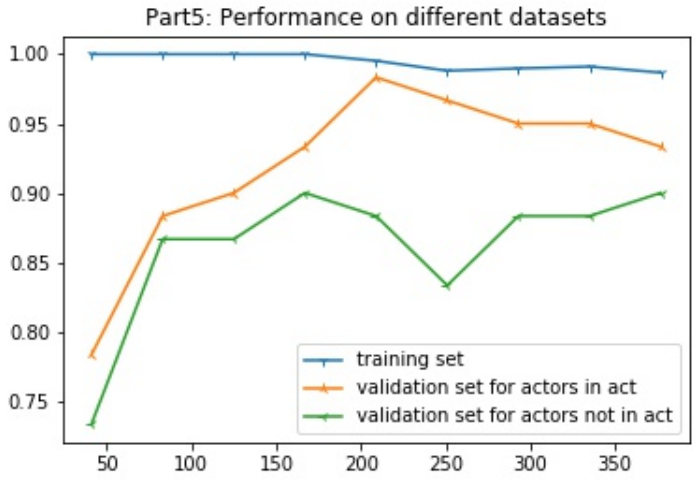

Figure 21

The result shows that the performance on training set remains really high (about 100%), although the performance slightly decrease as the size of the training set increases. The performance on the validation set of the acts who are in *act* is always higher than that of the acts who are not in *act*. Performance on the validation of both the actors who are in *act* and who are not in *act* increases as the size of the training set increases. When the size of training set is small, overfitting problem occurs: performance on training set is much higher than the performance on both validation sets. As the size of training set increases, the difference between the performances on training set and validation sets decreases, which indicates it is less overfitting.

Problem 6

a)

Compute $\frac{\partial J}{\partial \theta_{pq}}$

$$J(\theta) = \sum_{i=1}^{m} \left(\sum_{j=1}^{k} (\theta^T x^{(i)} - y^{(i)})_j^2 \right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{pq}} = \sum_{i=1}^{m} \frac{\partial}{\partial \theta_{pq}} \left[\sum_{j=1}^{k} (\theta^T x^{(i)} - y^{(i)})_j^2 \right]$$

$$= \sum_{i=1}^{m} \frac{\partial}{\partial \theta_{pq}} \left[(\theta^T x^{(i)} - y^{(i)})_1^2 + (\theta^T x^{(i)} - y^{(i)})_2^2 + \cdots + (\theta^T x^{(i)} - y^{(i)})_q^2 + \cdots + (\theta^T x^{(i)} - y^{(i)})_k^2 \right]$$

$$= \sum_{i=1}^{m} \frac{\partial}{\partial \theta_{pq}} (\theta^T x^{(i)} - y^{(i)})_q^2$$

$$= \sum_{i=1}^{m} \left[2 \cdot \frac{\partial}{\partial \theta_{pq}} (\theta_q^T x^{(i)}) \cdot (\theta^T x^{(i)} - y^{(i)})_q \right] \quad -①$$

$$\frac{\partial}{\partial \theta_{pq}} (\theta_q^T x^{(i)}) = \frac{\partial}{\partial \theta_{pq}} \left[\sum_{l=1}^{n} \theta_{ql}^T x_l^{(i)} \right]$$

$$= \frac{\partial}{\partial \theta_{pq}} \left[\theta_{1q} x_1^{(i)} + \theta_{2q} x_2^{(i)} + \cdots + \theta_{pq} x_p^{(i)} + \cdots + \theta_{nq} x_n^{(i)} \right]$$

$$= x_p^{(i)}$$

$$① : \frac{\partial J(\theta)}{\partial \theta_{pq}} = \sum_{i=1}^{m} \left[2 \cdot x_p^{(i)} \cdot \left(\sum_{l=1}^{n} \theta_{ql}^T x_l^{(i)} - y_q^{(i)} \right) \right]$$

$$= 2 \sum_{i=1}^{m} \left[x_p^{(i)} \cdot \left(\sum_{l=1}^{n} \theta_{ql}^T x_l^{(i)} \right) \right]$$

b)

$$(2X(\theta^T X - Y)^T)_{pq} = 2 \cdot \sum_{i=1}^{m} X_{pi}(\theta^T X - Y)^T_{iq}$$

$$= 2 \cdot \sum_{i=1}^{m} X_{pi}(\theta^T X - Y)_{qi}$$

$$= 2 \cdot \sum_{i=1}^{m} X_{pi}((\theta^T X)_{qi} - Y_{qi})$$

$$= 2 \cdot \sum_{i=1}^{m} X_{pi}\left[\left(\sum_{l}^{n} \theta_{ql}^T x_h^{(i)}\right) - y_q^{(i)}\right]$$

$$= \frac{\partial J}{\partial \theta_{pq}}.$$

c)

f_multi is a modified function to calculate the cost function, *df_multi* is a modified function to calculate the gradient.

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

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

d)

Comparison between gradient obtained by vectorized gradient function and gradient obtained by finite-difference approximations is used to verify that the vectorized gradient function stated in part 6c functions properly. The finite-difference approximation is given below:

$$\frac{\partial f}{\partial \theta_i} \approx \frac{f(\theta_1, \theta_2, ..., \theta_i + h, ..., \theta_n) - f(\theta_1, \theta_2, ..., \theta_i, ..., \theta_n)}{h}$$

In this case, the f is the cost function which is calculated by function f_multi and the variable is $theta$. The training set contains all the images in training sets(act_data[actor's name][0] obtained at part2) for all actors in act. The verification initializes the theta to be all zeros, run 50 iterations of gradient descent, then calculate the gradient obtained by vectorized gradient function(through function df_multi) and by finite-difference approximations(through function $finite\_diff$). The selection of h is quite critical because mathematically, smaller h leads to more accuracy approximation; however, too small h will lead to underflow problem when using computer, which can significantly increases inaccuracy. So in order to select the optimal h, I calculated the average difference over 5 coordinates with different h. A plot of average difference over 5 coordinates vs. h is shown below.

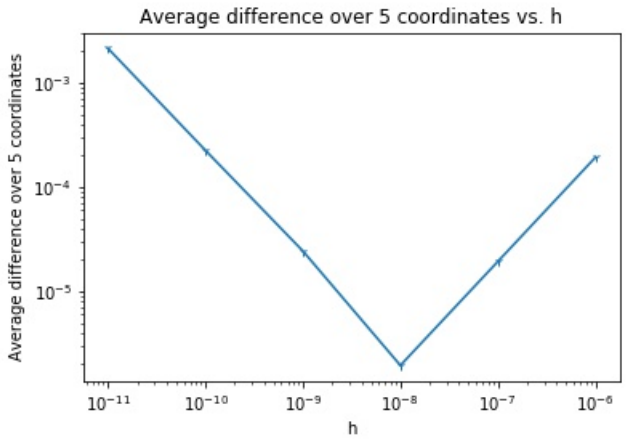

Figure 22

As shown in the figure, selecting h as 10^{-8} gives us a accurate approximation. The difference along 5 coordinates with h as 10^{-8} are:
Difference in gradient[235, 0] is 0.0000027125
Difference in gradient[905, 1] is 0.0000026027
Difference in gradient[715, 4] is 0.0000004888
Difference in gradient[847, 5] is 0.0000026068
Difference in gradient[960, 4] is 0.0000013362

The code used is pasted below:

```
def finite_diff(f, x, y, theta, row, col, h):
     #function for calculating one component of gradient
     #using finite-difference approximation
    theta_h = np.copy(theta)
    theta_h[row,col] = theta_h[row,col] + h
    return (f(x, y ,theta_h) - f(x, y, theta))/h
```

```
def part6d():
    #construct training set
    act1 = ['Lorraine Bracco', 'Peri Gilpin', 'Angie Harmon', 'Alec Baldwin', \
    'Bill Hader', 'Steve Carell']
    X_train = np.empty((1024, 0))
    Y_train = np.empty((len(act1), 0))
    for i in range(len(act1)):
        temp = np.zeros((len(act1), act_data[act1[i]][0].shape[1]))
        temp[i, :] = 1
        X_train = np.hstack((X_train, act_data[act1[i]][0]))
        Y_train = np.hstack((Y_train, temp))

    #run 50 iterations of gradient descent
    np.random.seed(1)
    theta0 = np.random.normal(scale=0, size=(1025, 6))
    theta = grad_descent(f_multi, df_multi, X_train, Y_train, theta0, 0.000001,\
     max_iter=50)
    gradient = df_multi(X_train, Y_train, theta)

    #pick 5 random coordinates
    np.random.seed(1)
    row_ = np.random.randint(0,1026,5)
    col_ = np.random.randint(0,7,5)
    h_ = (10**exp for exp in range(-11,-5))
    error_ = []
    hh =  []
    for h in h_:
        hh.append(h)
        temp = 0
        for i in range(5):
            temp += abs(finite_diff(f_multi, X_train, Y_train, theta, row_[i],\
                                    col_[i],h) - gradient[row_[i], col_[i]])
        error_.append(temp/5.)

    #save figure
    fig = plt.figure(6)
    plt.title('Average difference over 5 coordinates vs. h')
    plt.xlabel('h')
    plt.ylabel("Average difference over 5 coordinates")
    plt.loglog(hh,error_,'-1')
    plt.show()
    fig.savefig(dirpath + '\part6d_1.jpg')

    #print out result
    for i in range(5):
        print("Difference in gradient[%i, %i] is %010.10f" %(row_[i], col_[i] \
              ,abs(finite_diff(f_multi, X_train, Y_train, theta, row_[i],\
                               col_[i],10**(-8)) - gradient[row_[i], col_[i]])))

    return
```

Problem 7

Face Recognition on Six Actors In terms of prediction, $h_\theta(x)$ in this part is a vector and the original label is also a vector that only one element being 1 and other elements are zeros. So we can transform our prediction in the same format as the original label: the maximum value among $h_\theta(x)$ is considered as 1 and other values are considered as 0. For example, say $h_\theta(x)$ is [0.1,0.7,0.3,0.5,0.1,0.4]; then, the transformed prediction is [0,1,0,0,0,0].

In this part, we need to tune parameters in order to get this algorithm functions properly. Similar to part3, we initialize thetas to be all zeros. The selection of learning rate is done in the same fashion as how it is chosen in part3. The result is shown below in Figure 23. From the plot, alpha of 0.0000072 leads to divergence. So selecting alpha as 0.000004 which is the largest number that will not lead to divergence to faster the gradient descent.

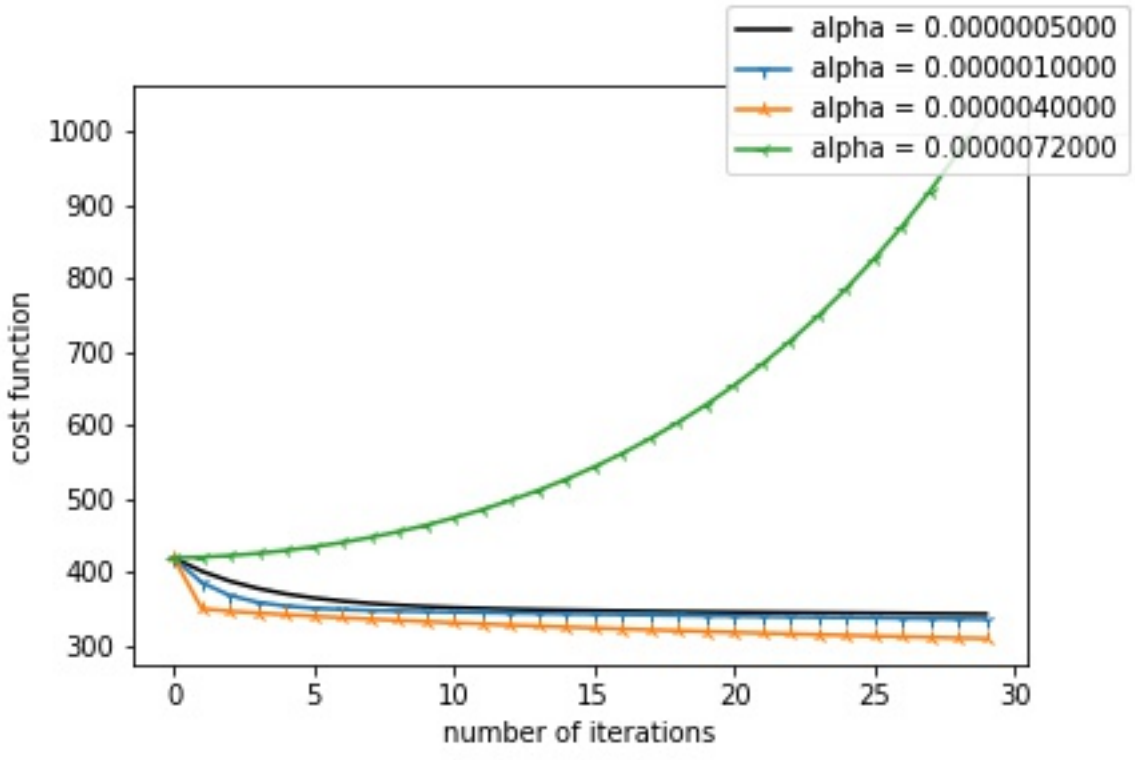

Figure 23

We also need to tune maximum number of iteration as overfitting might occur in this problem. So the gradient descent should stop when the it has best performance on validation set rather than at time when the cost function is minimized. A plot shown in Figure 24 below shows the relation between the maximum number of iterations and the performance on the validation set. From the figure, the model performs the best on validation set when maximum number of iterations is around 10000.

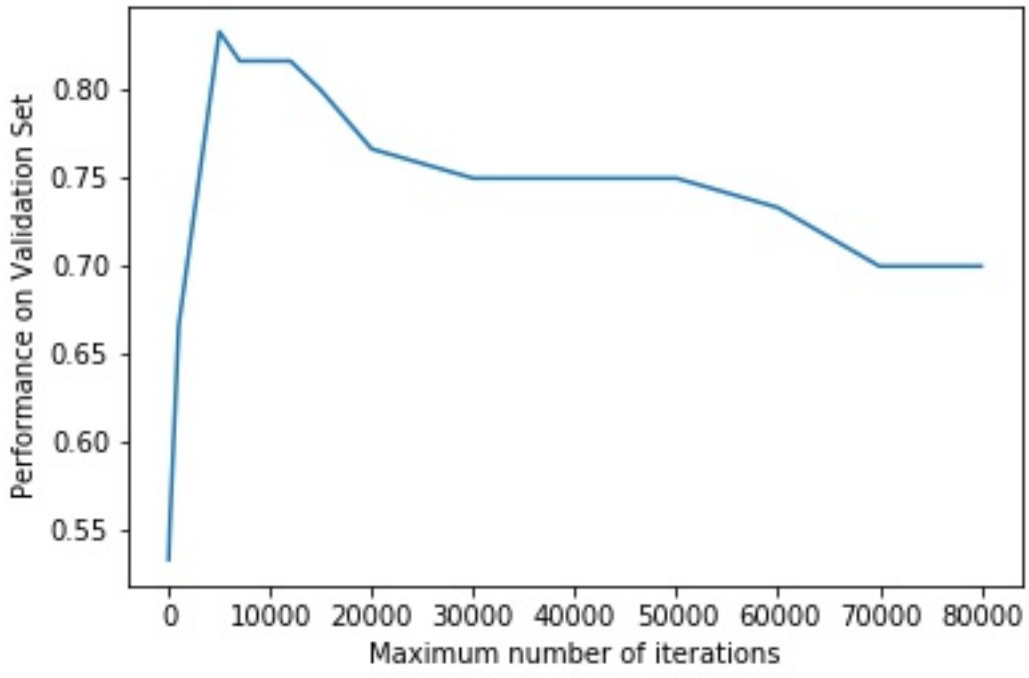

Figure 24

With initial thetas being all zeros, learning rate *alpha* being 0.000004, maximum number of iteration being 10000, and stopping condition EPS being 0.00001 which is given in the code provided in CSC411, the result is as following:

this is the result for part7
('Cost function on training set:', 83.57263081491993, 'Normalized:', 13.928771802486656)
('Cost function on validation set:', 29.586979152868857, 'Normalized:', 4.931163192144809)
('Cost function on validation set:', 28.024608743663755, 'Normalized:', 4.670768123943959)
('Performance on training set', 0.9618138424821002)
('Performance on validation set', 0.8166666666666667)
('Performance on test set', 0.85)

Problem 8

Visualization of thetas

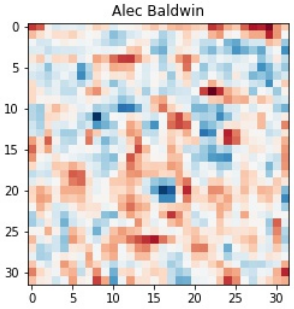

Figure 25: Alec Baldwin

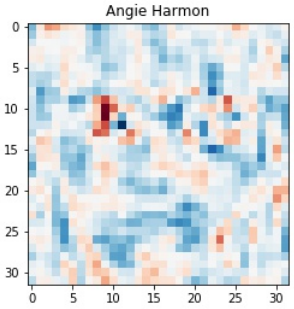

Figure 26: Angie Harmon

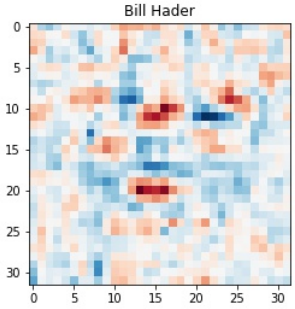

Figure 27: Bill Hader

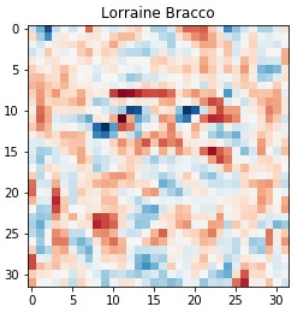

Figure 28: Lorraine Bracco

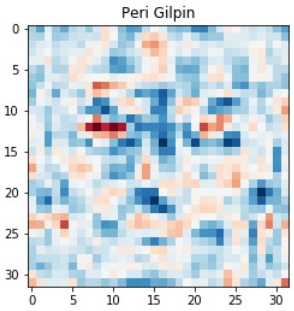

Figure 29: Peri Gilpin

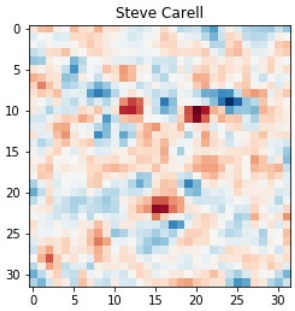

Figure 30: Steve Carells