CSC411 Project 1

Zikun Chen
1001117882

January 30, 2018

Note: The code for downloading and cleaning has been commented out (at the start and in part 5). The downloaded data are in data.zip file. Please put all files (txt, images and py) in the same directory while running the program and arrange the face images in a smaller directory called 'data' within the larger directory.

Part 1

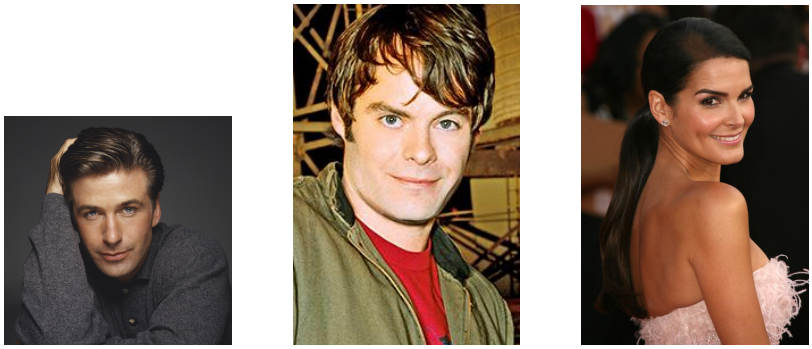


Figure 1: uncropped pictures

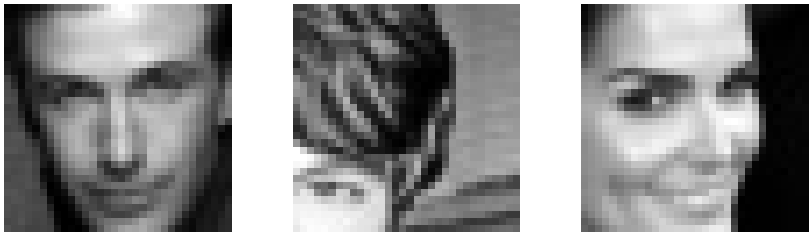


Figure 2: cropped pictures

The bounding boxes for the faces are mostly accurate. The faces in the cropped images are centered. They can be aligned with each other because their noses are in the center of the image. There are occasionally some inaccurate cropping like the second images above, so we remove them. And some images contain faces that don't face directly to the shot such as the third image, but this is okay to work with.

Part 2

The raw images are stored in the folder "uncropped" and the cropped and normalized images are stored in the folder "data". In addition, we have a text file listing the images associated with expired images and mis-cropped images "badimages.txt", which are removed from the data set.

The program first reads in actors' last names from file names of images in the "data" folder and stores them to separate lists for different actors. Then for each actor's file name list, the program sorts the list according to the order of downloading and shuffles it using np.random.shuffle(). We set the variable SEED to 2018 for any random result to be reproducible. The program then takes the first 70 images in each actor's list to be the training files for that actor, the next 10 for validation and the next 10 for the test set.

The last step is to obtain file name list for the full training set. The program concatenates respective lists (training, validation and test file names) of actors that we want to include and shuffles the full list again because we do not want the classifier to train on images of only one particular actor first before it moves on to the next one. For example, training_files in part 3 and training_files_multi for part 7 are file name lists for the respective training sets.

Part 3

The classifier minimizes the quadratic cost function (m is number of training examples):

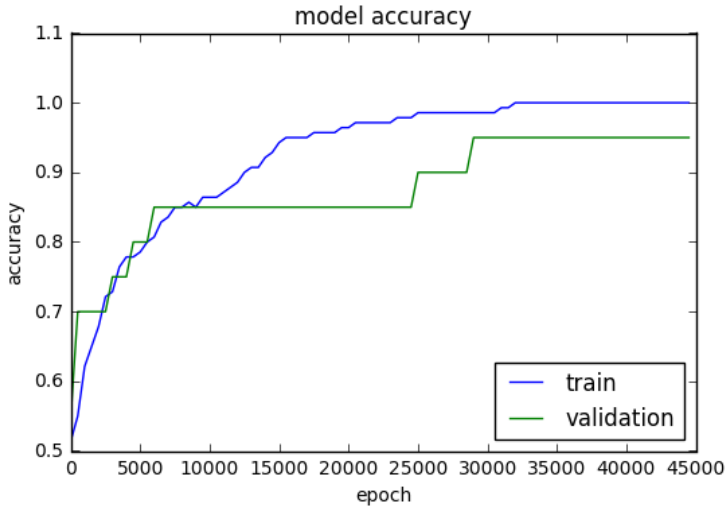
$$\sum_{i=1}^{m}(y^{(i)} - \theta^T x^{(i)})^2$$

Using initial random θs that has distribution $N(0, 0.1)$ and $\alpha = 10^{-6}$, we have:

```
--------final performance--------
value of quadratic cost function on training set is 4.28
value of quadratic cost funciton on validation set is 4.28

training accuracy: 100.00%
validation accuracy: 95.00%
```

Function to compute the output of the classifier:



model accuracy during training

```
def accuracy(x, y, t):
    M = float(x.shape[1])
    return divide((M-(count_nonzero((dot(t.T, x)>0.5)-y))),M)
```

In particular,

```
(dot(t.T, x)>0.5)
```

gives the classification of the classifier as a list of Boolean expressions: True (same as 1) for Steve Carell and False (same as 0) for Alec Baldwin. We then subtract it from the true label y and count the number of nonzero elements (unsuccessful classifications). We can then obtain the accuracy for the trained classifier.

What I had to do to get the system to work:

The system does not work when learning rate α is too large because even though each gradient decent step goes in the direction of the steepest decrease for the cost function locally, it takes too large of a step and results in the value of the cost function to overshoot. Consequently, the gradient descent algorithm will not converge.

On the other hand, if the learning rate is too small, the training will take very long before reaching the optimal level.

In addition, if the initial θs are not close enough to the optimal θs, then the cost function can get stuck at a local minimum with bad validation performance. For example if we use $N(0, 1)$ for the distribution of initial random parameters, we get only 60% accuracy on the validation set.

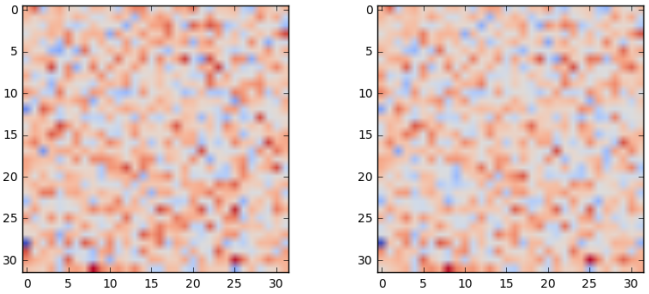
```
--------final performance--------
```

```
value of quadratic cost function on training set is 12.63
value of quadratic cost funciton on validation set is 12.63

training accuracy: 92.86%
validation accuracy: 65.00%
```

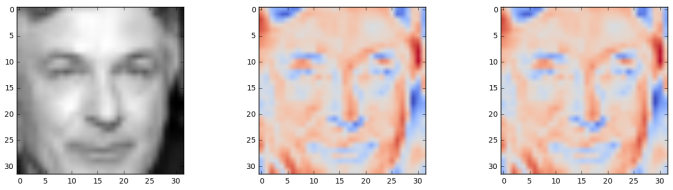
Part 4

a) Display θs as images:



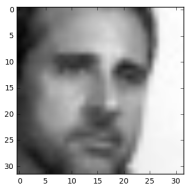
$\theta_i's$ shown as 32×32 images: full training set vs. 2-image training set

b) To obtain the visualization of θs without faces in part a), we can simply take the last 1024 elements from the output θ vector and reshape it to show a 32×32 image. We can obtain visualizations of θs that contain faces by initializing the θs to flattened images of actors with additional 1's. Then we can visualize them after training by using the same technique as in part a). The results are shown below:

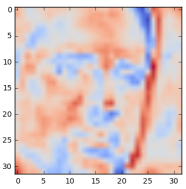


(a) original image (b) stopped at 3000 epoch (c) full training

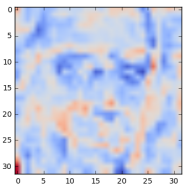
initialized with 'data/baldwin47.jpg'



(a) original image



(b) stopped at 10000 epoch

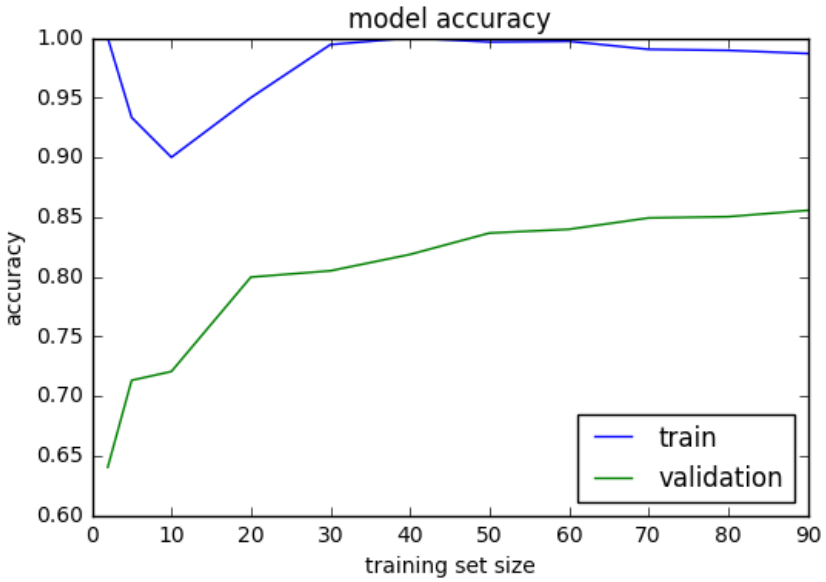


(c) full training

initialized with 'data/carell35.jpg'

Part 5

To demonstrate over-fitting, we use early stopping to obtain optimal validation set performance for different sizes of training sets (2, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90). The criterion that I chose is that we want to stop training if there are 5 instances of decrease for the performance of validation set after training set accuracy surpasses 85%.



model accuracy vs. training set size

We can see from the above image that when we train each classifier until the best performance on the validation set, the validation set has higher performance when the training set is larger. This implies that for smaller training set, it is more likely to over-fit and not generalize well to unseen data.

In addition, the six actors in the validation set are different from those in the training set. So there could be some bias in choosing the data set as well, which can influence the performance of the classifier on the model.

Part 6

a)

$$J(\theta) = \sum_{i=1}^{m}(\sum_{j=1}^{k}(\theta_j^T x^{(i)} - y^{(i)})^2) = \sum_{i=1}^{m}(\sum_{j=1}^{k}(\sum_{s=1}^{n}\theta_{sj}^T x_s^{(i)} - y_j^{(i)})^2)$$

p is a particular index in (1, ... n)
q is a particular index in (1, ... k)

$$\frac{\partial J}{\partial \theta_{pq}} = \partial \sum_{i=1}^{m}(\sum_{s=1}^{n}\theta_{sq}^T x_s^{(i)} - y_q^{(i)})^2 / \partial \theta_{pq}$$

$$= \sum_{i=1}^{m} 2(\sum_{s=1}^{n}\theta_{sq}^T x_s^{(i)} - y_q^{(i)}) x_p^{(i)}$$

$$= 2\sum_{i=1}^{m} x_p^{(i)}(\sum_{s=1}^{n}\theta_{sq}^T x_s^{(i)} - y_q^{(i)})$$

b) $n = 32 \times 32 + 1 = 1025$ is the length of a flattened 32×32 image with an addition 1
m is the number of training examples
$k = 6$ is the number of actors/actresses to classify in the output

$X_{n\times k}$ is the matrix that contains all the input training data and additional 1's
$Y_{k\times m}$ is the matrix that contains all the correct k-dimension labels for the m training examples
$\theta_{n\times k}$ is the matrix that contains the all the parameters that get updated during training.

From part a):

$$\frac{\partial J}{\partial \theta} = \begin{bmatrix} 2\sum_{i=1}^{m} x_1^{(i)}(\sum_{s=1}^{n}\theta_{s1}^T x_s^{(i)} - y_1^{(i)}) & \cdots & 2\sum_{i=1}^{m} x_1^{(i)}(\sum_{s=1}^{n}\theta_{sk}^T x_s^{(i)} - y_k^{(i)}) \\ \vdots & \ddots & \vdots \\ 2\sum_{i=1}^{m} x_n^{(i)}(\sum_{s=1}^{n}\theta_{s1}^T x_s^{(i)} - y_1^{(i)}) & \cdots & 2\sum_{i=1}^{m} x_n^{(i)}(\sum_{s=1}^{n}\theta_{sk}^T x_s^{(i)} - y_k^{(i)}) \end{bmatrix}$$

$$\theta^T = \begin{bmatrix} \theta_{11} & \cdots & \theta_{n1} \\ \vdots & \ddots & \vdots \\ \theta_{1k} & \cdots & \theta_{nk} \end{bmatrix} X = \begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(m)} \\ \vdots & \ddots & \vdots \\ x_n^{(1)} & \cdots & x_n^{(m)} \end{bmatrix}, Y = \begin{bmatrix} y_1^{(1)} & \cdots & y_1^{(m)} \\ \vdots & \ddots & \vdots \\ y_k^{(1)} & \cdots & y_k^{(m)} \end{bmatrix}$$

$$\theta^T X - Y = \begin{bmatrix} \sum_{s=1}^{n} \theta_{s1} x_s^{(1)} & \cdots & \sum_{s=1}^{n} \theta_{s1} x_s^{(m)} \\ \vdots & \ddots & \vdots \\ \sum_{s=1}^{n} \theta_{sk} x_s^{(1)} & \cdots & \sum_{s=1}^{n} \theta_{sk} x_s^{(m)} \end{bmatrix} - \begin{bmatrix} y_1^{(1)} & \cdots & y_1^{(m)} \\ \vdots & \ddots & \vdots \\ y_k^{(1)} & \cdots & y_k^{(m)} \end{bmatrix}$$

$$(\theta^T X - Y)^T = \begin{bmatrix} \sum_{s=1}^{n} \theta_{s1} x_s^{(1)} - y_1^{(1)} & \cdots & \sum_{s=1}^{n} \theta_{sk} x_s^{(1)} - y_k^{(1)} \\ \vdots & \ddots & \vdots \\ \sum_{s=1}^{n} \theta_{s1} x_s^{(m)} - y_1^{(m)} & \cdots & \sum_{s=1}^{n} \theta_{sk} x_s^{(m)} - y_k^{(m)} \end{bmatrix}$$

$$2X(\theta^T X - Y)^T = 2 \begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(m)} \\ \vdots & \ddots & \vdots \\ x_n^{(1)} & \cdots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} \sum_{s=1}^{n} \theta_{s1} x_s^{(1)} - y_1^{(1)} & \cdots & \sum_{s=1}^{n} \theta_{sk} x_s^{(1)} - y_k^{(1)} \\ \vdots & \ddots & \vdots \\ \sum_{s=1}^{n} \theta_{s1} x_s^{(m)} - y_1^{(m)} & \cdots & \sum_{s=1}^{n} \theta_{sk} x_s^{(m)} - y_k^{(m)} \end{bmatrix}$$

$$= 2 \begin{bmatrix} \sum_{i=1}^{m} x_1^{(i)} (\sum_{s=1}^{n} \theta_{s1}^T x_s^{(i)} - y_1^{(i)}) & \cdots & \sum_{i=1}^{m} x_1^{(i)} (\sum_{s=1}^{n} \theta_{sk}^T x_s^{(i)} - y_k^{(i)}) \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{m} x_n^{(i)} (\sum_{s=1}^{n} \theta_{s1}^T x_s^{(i)} - y_1^{(i)}) & \cdots & \sum_{i=1}^{m} x_n^{(i)} (\sum_{s=1}^{n} \theta_{sk}^T x_s^{(i)} - y_k^{(i)}) \end{bmatrix} = \frac{\partial J}{\partial \theta}$$

c) **Cost function and Vectorized Gradient function**

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

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

d) **Code:**

```
# 6 d)
def check_gradient_multi(theta, x, y, i, j, h=1e-7):
    deltaij = np.zeros(theta.shape)
    deltaij[i, j] = h
    fd = (J(x, y, theta + deltaij) - J(x, y, theta))/h
    dj = dJ(x, y, theta)[i, j]
    print 'theta [%1d, %1d]'%(i, j)
    print '-------------------------------------'
    print 'finite-difference approximation: %.7f\ndJ/dthetaij: %.7f' % (fd, dj)
```

```
    print 'absolute difference: %.7f' % (abs(fd - dj))

# contract labels and data for calculation
y_train = np.empty([len(training_files_multi), 6])
for i in range(len(training_files_multi)):
    z = np.zeros(6)
    if lastname(training_files_multi[i]) == 'bracco':
        z[0] = 1
        y_train[i] = z
    elif lastname(training_files_multi[i]) == 'gilpin':
        z[1] = 1
        y_train[i] = z
    elif lastname(training_files_multi[i]) == 'harmon':
        z[2] = 1
        y_train[i] = z
    elif lastname(training_files_multi[i]) == 'baldwin':
        z[3] = 1
        y_train[i] = z
    elif lastname(training_files_multi[i]) == 'hader':
        z[4] = 1
        y_train[i] = z
    elif lastname(training_files_multi[i]) == 'carell':
        z[5] = 1
        y_train[i] = z
y_train = y_train.T

X_train = imread('data/' +training_files_multi[0])[:,:,0].flatten()
for i in range(1, len(training_files_multi)):
    im = imread('data/' + training_files_multi[i])
    X_train = np.vstack((X_train, im[:,:,0].flatten()))
X_train = X_train/255.
X_train = X_train.T

# check gradient at different components of theta
np.random.seed(SEED)
init_t_multi = np.random.normal(0, 0.1, size=(32*32+1, 6))
check_gradient_multi(init_t_multi, X_train, y_train, 2, 4, h=1e-7)
check_gradient_multi(init_t_multi, X_train, y_train, 456, 2, h=1e-7)
check_gradient_multi(init_t_multi, X_train, y_train, 304, 0, h=1e-7)
check_gradient_multi(init_t_multi, X_train, y_train, 100, 5, h=1e-7)
check_gradient_multi(init_t_multi, X_train, y_train, 230, 3, h=1e-7)
```

Output:

```
theta [2, 4]
--------------------------------------
```

```
finite-difference approximation: 76.6186713
dJ/dthetaij: 76.6186629
absolute difference: 0.0000084
theta [456, 2]
--------------------------------------
finite-difference approximation: -913.2256037
dJ/dthetaij: -913.2256218
absolute difference: 0.0000181
theta [304, 0]
--------------------------------------
finite-difference approximation: 325.4574949
dJ/dthetaij: 325.4574675
absolute difference: 0.0000274
theta [100, 5]
--------------------------------------
finite-difference approximation: -48.6844510
dJ/dthetaij: -48.6844563
absolute difference: 0.0000053
theta [230, 3]
--------------------------------------
finite-difference approximation: -1093.2998975
dJ/dthetaij: -1093.2999105
absolute difference: 0.0000130
```

To compared the approximated values to the output of the gradient function, we take the absolute difference between them for a particular index [i, j] in the gradient.

h makes sense as long as it is relatively very small compared to the value of the corresponding component of the θ matrix.

Part 7

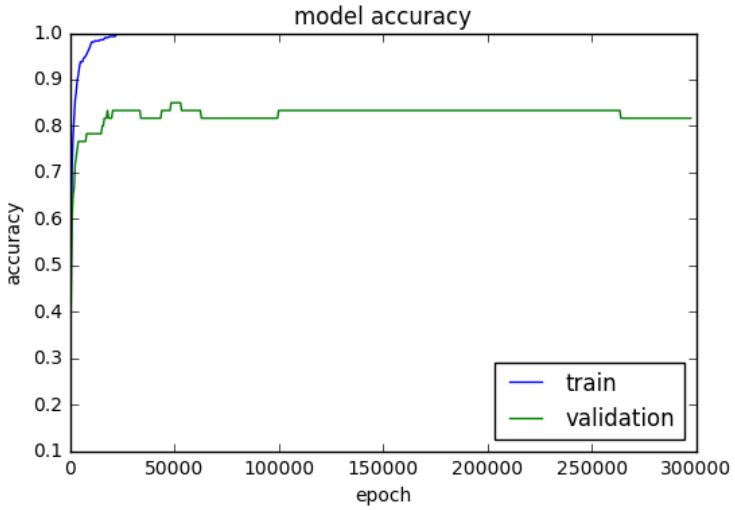
Performance:

```
--------final performance--------

training accuracy: 100.00%
validation accuracy: 81.67%
```

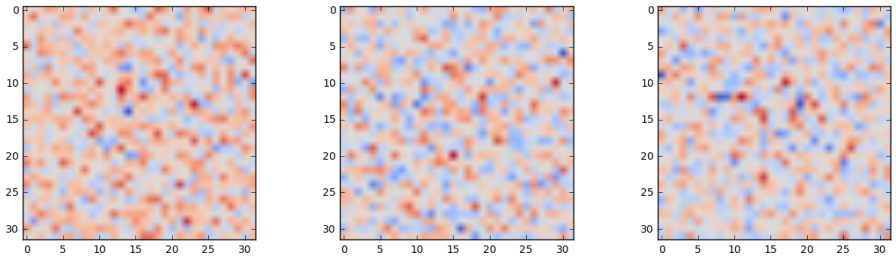
To initialize the parameter, I still chose the random θs that has distribution $N(0, 0.1)$ and a learning rate of $\alpha = 10^{-5}$.

To obtain the labels from the output θ matrix, we multiply the θ matrix with the X matrix and use the argmax function to see, for each column (a flattened image), which index contains the largest value. If that index is equal to that of the column (a flatten image with correct label) in the Y matrix, then this particular classification of the image is correct. (See multi_accuracy function)

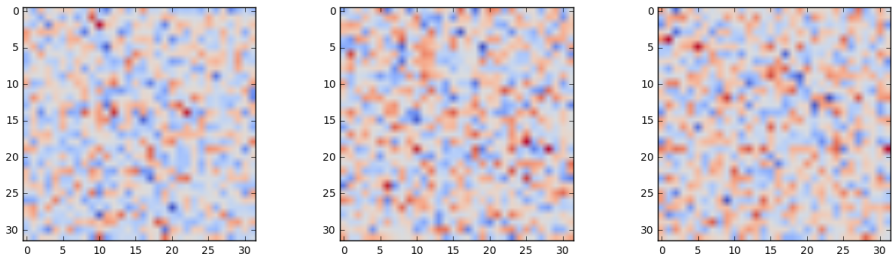


Part 8

To figure out which actors the 6 components of the θ matrix correspond to, we take one image of each actor from the training set and compute the argmax of the matrix multiplication between θ matrix and the flattened image. The resulting index corresponds to the actor in the that image. We know that this is correct because training set has 100% accuracy.



(a) Lorraine Bracco (at 0) (b) Peri Gilpin (at 1) (c) Angie Harmon (at 2)



(d) Alec Baldwin (at 3) (e) Bill Hader (at 4) (f) Steve Carell (at 5)

visualization of 6 components of θ matrix labeled by actors' names