CSC411: Project 1: Face Recognition and Gender Classification with Regression

Due on Wednesday, February 1, 2017

Pham-Hung, Michael

October 14, 2017

The Dataset

Part 1

For our dataset, we've taken a collection of photos from six different actors and actresses (Fran Drescher, America Ferrera, Kristin Chenoweth, Alec Baldwin, Bill Hader, Steve Carell). The photos are a subset of the FaceScrub dataset, which have the coordinates to crop just the faces from the photo. We've collected the photos, cropped and then grey-scaled them for use in this project.

Looking at the retrieved and uncropped photos

For each set of photos, there are different types of images. There are some photos that have dark contrasts on their face, some pictures with the actor looking in different directions. Additionally, some pictures contain obstructions such as some actors sporting facial hair and/or glasses. See Figure 1 for examples. Images (a) - (d) show examples of the different features from the uncropped images. Figures 1.a and 1.b show Alec Baldwin in high contrast lighting and at an angle, respectively. Figures 1.c and 1.d show Gerard Butler with and without a beard, respectively.

In addition to the regular images of the actors, during the data retrieval, some images do not come out as expected due to expired image links (Figure 2). This introduces a bit of noise to our data. For reproducibility, we've kept these images throughout project.

(a) Image with high contrast.

(b) Image taken at an angle.

(c) Image with facial hair.

(d) Image without facial hair.

Figure 1: Examples of uncropped images.

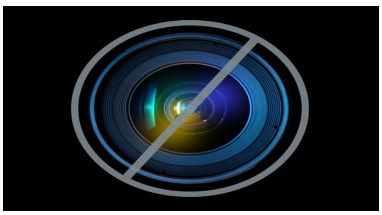


Figure 2: Image retrieved from an expired link

Looking at the cropped images

Cropping introduces additional complications as well as few benefits. Overall, the eyes of the actors/actress are around the same height, but for the photos taken at an angle or with the actor's head tilted, the eyes are also misaligned. Additionally, in order for the eyes to be aligned, some of the chins and other facial features of the actors are cut off. For the most part, the images can be aligned with each other.

Additionally, grey-scaling also removes any colors that would help identify an actor (i.e. the color of their hair) or distinguish their face from the background. Figure 3 shows three cropped images of Lorraine Bracco. Note that in figures 3.b-3.d that the **center** of the eyes are all aligned, however in 1.d, the actual eyes are offset. Additionally, in figure 3.b, it is not clear if the left side of the face is hair or the background.

(a) Actress with red hair

(b) Cropped version of 3.a

(c) Image with straight eyes

(d) Image with slanted eyes

Figure 3: Examples of uncropped images.

Part 2

After downloading and cropping the images into a folder, the create_set() function in the faces.py file created the datasets given the size and actor. The function iteratively goes through a **randomly** sorted list of the images to create a training set of the given size, validation set of size 10, and a test set of size 10. Randomizing the order of the images removes any biases, but for reproducibility, the randomization was seeded.

Using the make() function in the faces.py file iteratively calls create_set() for every actor to be used as data. As an input, a **dictionary** of the actors and their respective labels are used to create the input and output label data to be used in the next sections.

Building a classifier

Part 3

The Algorithm

In order to create our classifier between Bill Hader and Steve Carell using linear regression, we needed to find a function $h_\theta(x) = \theta_0 + x_1\theta_1 + ... + x_n\theta_n$. Finding the theta's that would best fit this hypothesis, we can minimize the following cost function:

$$J(\underline{\theta}) = (\frac{1}{2})\sum_j (\underline{\mathbf{X}\theta} - \mathbf{y})_j^2 \tag{1}$$

$\underline{\mathbf{X}}$ is the training matrix with $n \times m$ dimensions where n is the number of images in the training set and m is the number of bits per image (1024+1 bits per image and a dummy term for biases). $\underline{\theta}$ is the weight vector with dimensions $m \times 1$ and y is the target vector of each image (i.e. if the image is Steve Carell or Bill Hader). Note the vectorization which removes the need of multiple for-loops when running the algorithm.

We want to find a suitable weight vector $\underline{\theta}$ which minimizes the cost function (1). To do so, we used the gradient descent algorithm: $\theta \leftarrow \theta - \alpha\frac{\delta J}{\delta\theta}$, where α is the step size or **learning rate** of the algorithm and $\frac{\delta J}{\delta\theta} = \underline{\mathbf{X}}^T(\underline{\mathbf{X}\theta} - \mathbf{y})$, which is the gradient of the cost function.

The following code illustrates the function, its gradient, and the gradient descent algorithm. The gradient descent algorithm was taken from the course website.

```
1  def f(x, y, theta):
2      return (1/2)*((dot(x,theta) - y) ** 2)
3
4  def df(x, y, theta):
5      return dot(x.T,(dot(x,theta)-y))
6
7  def grad_descent(f, df, x, y, init_t, alpha):
8      EPS = 1e-5   #EPS = 10**(-5) error per step
9      prev_t = init_t-10*EPS
10     t = init_t.copy()
11     max_iter = 30000
12     iter  = 0
13     while norm(t - prev_t) >  EPS*150 and iter < max_iter:
14         prev_t = t.copy()
15         t -= alpha*df(x, y, t)
16         iter += 1
17
18     print t
19     print "Iter", iter
20     return t
```

Making it Work

In order for this system to work, a few parameters needed to be configured.

Firstly, the image vectors were **normalized** so that the gradient vector wouldn't become too large. Each element was normalized such that $x_i \leftarrow \frac{(x_i - 127)}{255}$. Doing such a normalization makes a vector with values in the range of $1 - 255$ have a range of values between -1 and 1. These values would also carry over to the gradient vector and if it was too large, the required step size would become extremely small.

Another parameter that was tuned in order to make the system work was α or the **step size** of the gradient descent function. If α was too large, the gradient descent would be unstable and the output would go

```
(grad_descent)>>> t
array([[ nan],
       [ nan],
       [ nan],
       ...,
       [ nan],
       [ nan],
       [ nan]])

(grad_descent)>>> iter
192
```

(a)

```
(grad_descent)>>> t
array([[  1.27490196e-09],
       [  1.03098039e-09],
       [  1.07294118e-09],
       ...,
       [ -1.60784314e-11],
       [  1.74117647e-10],
       [  0.00000000e+00]])

(grad_descent)>>> iter
1
```

(b)

Figure 4: Output of the gradient descent function given a) small alpha b) large alpha.

to infinity. However, if α was too small, the gradient descent algorithm would stop at the first iteration by exiting the while loop too early. Figure 4 shows the output of the gradient descent function given the described alphas, where t is the outputted weight vector θ and iter is the number of iterations of the gradient descent function.

The last parameter that was altered was the error per step (**EPS**) parameter used in the gradient descent function. EPS controls when the gradient descent stops searching for a new theta. If too large, the function will only go through one iteration. Too small, and the function may never end (A max_iter variable is assigned to prevent an infinite loop). Additionally, as we increased EPS, we were able to reduce overfitting of the training set, increasing the classifier's performance on the test and validating set.

Note: It was also found that the initial θ also had an effect on the performance of the algorithm and that setting all the weights to 0 provided the best performance.

In order to find the best configuration of these parameters, they were each slightly altered and their performance was evaluated using a validation set. Ideally, it would be suitable to create a function that would run the gradient descent function and choose the alpha that outputted the greatest performance with the validation set. This would find the perfect alpha that would find the best theta to achieve the best hypothesis function for the classifier. In this project however, alpha was chosen manually to reduce the runtime needed when running the entire program.

The function validate(), shown below, evaluates the performance θ with respect to the validation set. It was used manually during the project to find a good α. The function takes in as input the test/validation set (test), its respective labels (target) and the weight vector outputted by the gradient descent function (theta). The resulting output was then evaluated against the target output and the function returns the total performance as a ratio of correctly classified inputs.

```
1  def validate(test, target, theta):
2      '''Returns the performance of the given theta using the test/validation set'''
3      results = dot(test,theta)
4      m = int(test.shape[0]) #Number of test cases
5      k = 0
6
7      for i in range(m):
8          if results[i]*target[i] > 0: # Positive value means elements match.
9              k+=1
10
11     return float(k)/m*100
```

Running the Algorithm

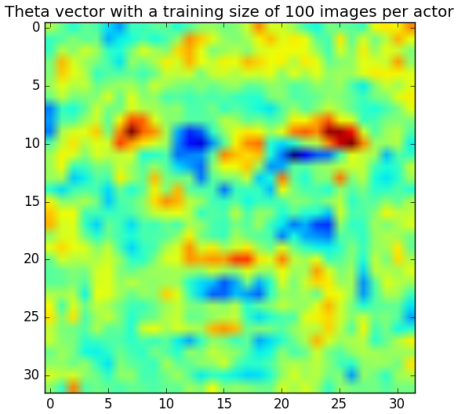
Below, the code used for part 3 and its corresponding outputs are provided. The actors are defined in a dictionary with Carell as positive 1 and Hader as negative 1. *Note the values for alpha and init_t.* Figure 5 shows the output of running the code. *Note the better performance in the test set then the validation set: it was found that a bad image (See Figure 2) was amongst the validation set.*

```
def part3():
    print "\n_________________Part 3:_________________"

    '''Defining Variables'''
    act = {'carell':1, 'hader':-1}
    x, val, y, y_val, test, ytst = make(act, 100)
    init_t = np.zeros((1025,1))
    alpha = 0.0001

    '''Running Gradient Descent'''
    t = grad_descent(f, df, x, y, init_t, alpha)

    '''Printing Out Results'''
    print "\nRecieved a score of", validate(val, y_val, t), "on the validation set"
    print "\nRecieved a score of", validate(test, ytst, t), "on the test set"
    print "\nValue of cost function on validation set:", f(val, y_val, t)
    print "\nValue of cost function on test set:", f(test, ytst, t)

    '''Showing Thetas as an image'''
    t_to_graph = np.delete(t,-1)
    figure()
    title("Theta vector with a training size of 100 images per actor")
    imshow(t_to_graph.reshape(32,32))
```

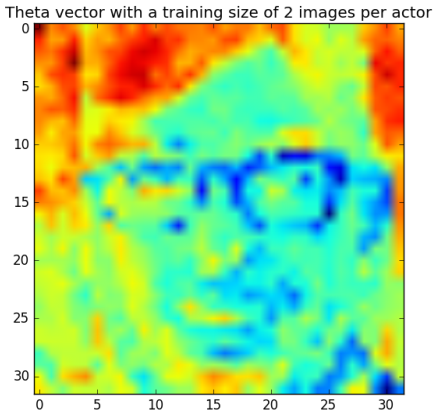
```
_________________Part 3:_________________
Final Theta =  [[ 0.00493295]
 [-0.07043611]
 [-0.13341758]
 ...,
 [-0.00150261]
 [-0.03345971]
 [-0.10552076]]
Total number of Iterations: 13595

Recieved a score of 80.0 on the validation set

Recieved a score of 85.0 on the test set

Value of cost function on validation set: 5.87874929009

Value of cost function on test set: 5.03226348002
```

(a)

Figure 5: Output of part 3



(a)



(b)

Figure 6: Visualization of $(\theta_1, ..., \theta_n)$ with training size of a) 100, b) 2

Part 4

Taking the outputted weight vectors from running gradient descent with different training sizes and displaying them as a 32×32 image, we can visualize the effect of training size on the weight vectors. In figure 6, we can see that with a smaller size, the vector looks more like a face. In 6.b, you can see that the weight vectors are memorizing the training set (Steve Carell's glasses are evident) and in doing so, performance on the test and validation were poor. In 6.a we can see blotches that reflect regular features from the training set as opposed to one whole face which helps increase the performance of the classifier.

Extending the Classifier

Part 5

In this section, we extend the data set to include more actors in order to classify between male and female. Additionally, the size of the training sets are varied in order to demonstrate overfitting. Using a for-loop we've looped through the different training sizes and trained the classifier with each training size. Similar to previous sections, we've assigned assigned a dictionary to represent the actors and their labels. Female actors were labeled with a positive 1 and male actors with negative 1. Two different dictionaries were assigned to differentiate the actors in the validation set and in the test set.

Figure 7 illustrates the performance of the classifier. over different sizes of the training size. The training set showed increased performance as the training size increased. This makes sense as we're feeding more information and more regularities to classify the images it is training with. Additionally, the training set never has a perfect performance most likely due to bad images. The validation and test sets both showed similar performance.

The validation set, which contained the same actors as the training set, showed better performance then the test set overall. Up to around training size equal to 17 images per actor, the validation set and test set both performed equally. At around 80 images per actor, the validation set has a noticeable peak in performance, which then decreases after.

Overall, the test set had a worse performance than the validation set, which makes sense since the actors in the test set shared less **features** than the the actors in the training set. The classifier is then more likely to classify the new actors wrongly. However, for a small training size, the test set performed slightly better. This is probably due to training the classifier to recognize more general features (i.e. getting the gist of male vs. female). Furthermore, similarly to the validation set performance, and ignoring the slight peaks, the overall performance of the test seems to peak around the same region as the validation set.

From the analyzing the performance of the training set vs the validation and test sets, it is evident that overfitting decreases the performance of the classifier overall.

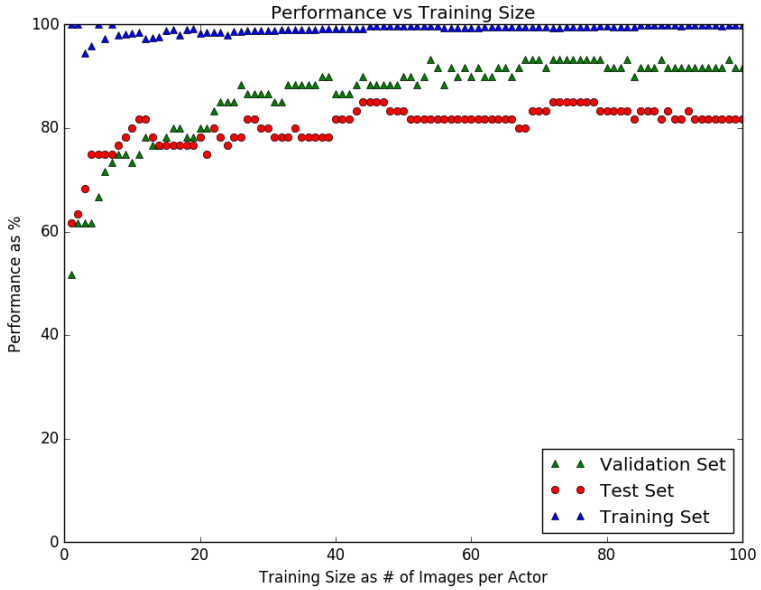


Figure 7

Part 6

a) Given the cost function $J(\theta) = \sum_i \left(\sum_j (\theta^T x^{(i)} - y^{(i)})_j^2 \right)$, we can derive $\frac{\delta J(\theta)}{\delta \theta_{pq}}$. Consider the inner sum as function $L = \sum_j (\theta^T x - y)_j^2$. Taking the derivative of L with respect to θ_{pq}, the term appears only in the qth row of the vector resulted from computing $\theta^T x^{(i)} - y^{(i)}$. This gives us $\frac{\delta L(\theta)}{\delta \theta_{pq}} = 2x_p(\theta_q^T x - y_q)$. Now, $J(\theta)$ can be defined as :

$$J(\theta) = \sum_i L(x^{(i)}, y^{(i)}, \theta) \tag{2}$$

Then,

$$\frac{\delta J(\theta)}{\delta \theta_{pq}} = \sum_i \frac{\delta L(\theta)}{\delta \theta_{pq}} = 2 \sum_i x_p^{(i)} (\theta_q^T x^{(i)} - y_q^{(i)}) \tag{3}$$

b) From here, we can vectorize. We can note that we can simplify the gradient for each row to:

$$\frac{\delta J(\theta)}{\delta \theta_p} = 2 \sum_i x^{(i)} (\theta_q^T x^{(i)} - y_q^{(i)}) = 2X(\theta_q^T X - Y_q) \tag{4}$$

And further vectorizing by column we arrive at the result:

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

X has dimensions $n \times m$, θ has dimensions $n \times k$ and Y has dimensions $k \times m$. Where n, m, and k are the training size, image size and number of labels respectively. c) Below is a snippet of the code to implement the cost function and its gradient.

```
def f_part6(x,y,theta):
    return np.sum((dot(theta.T,x) - y)**2)

def df_part6(x,y,theta):
    return 2*dot(x.T,(dot(x,theta) - y))
```

d) From finite differences we have the following equation:

$$\frac{\delta f}{\delta x_k} = lim_{h \to 0} \frac{f(x_0, ..., x_k + h, ..., x_n) - f(x_0, ..., x_k - h, ..., x_n)}{2h} \tag{6}$$

Using equation 6, we can see the correctness of the calculated gradient. The following snippet of code shows how the checking was implemented. It was found that both results matched up to 10 significant figures.

```
def part6d():
    act ={'drescher':[1,0,0,0,0,0], 'ferrera':[0,1,0,0,0,0], 'chenoweth':[0,0,1,0,0,0],
            'baldwin':[0,0,0,1,0,0], 'hader':[0,0,0,0,1,0], 'carell':[0,0,0,0,0,1]}
    x, val, y, y_val, dum, dum1 = make(act, 4)
    row = [0,1,2]
    col = [0,1,2]

    for i in row:
        for j in col:
            theta0 = np.zeros((1025,6))
            theta1= np.zeros((1025,6))
            theta2 = np.zeros((1025,6))
            h = 0.01
            theta1[i][j] = h
```

```
            theta2[i][j] = -h

            print i, "th row, ", j, "th column"
            print "Finite Differences:", (f_part6(x,y,theta1)-f_part6(x,y,theta2))/(2*h)
            print "Gradient:", df_part6(x, y, theta0)[i][j], "\n"
```

Part 7

Implementing one hot encoding, we were able to use previously defined functions such as **create_set()** and **make()**, and only need to create a new function to evaluate the performance of the validation and test sets (due to the new definition of labels). As inputs, the matrix **X** was defined as in the previous sections. The target vector y however was a $m \times k$ matrix where m was the number of images and k was the number of labels. Similarily, the θ matrix had dimensions $n \times k$.

For the evaluation function validate_part7(), the goal **matrix** had every row corresponding to each image. If the index of the maximum value in each row was the same index as the 1 in the target matrix, then a match would occur.

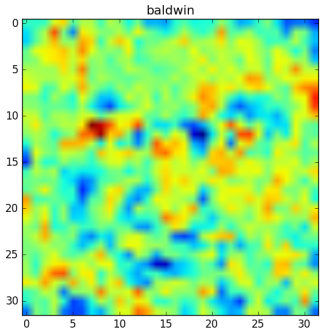
Figure 8 shows the output from part 7. Note the shape of the outputted theta is $n \times k$. In addition to the parameters discussed in part 3, the size of the training set needed to be tuned in order to obtain a good result. Using the results from part 5, we saw that there was a peak of performance when the number of images per actor was around 17. Again, it would be ideal to create a function that would go through all of the possible parameters and tune them against the validation set. However, tuning 4-5 parameters at the same time requires a large amount of processing, and so it was omitted in this project.

```
Final Theta =  [[-0.00788345 -0.01155399  0.01094008  0.01681337 -0.01087233 -0.00449228]
 [ 0.02625219  0.01361842  0.02188547 -0.00533114 -0.00652968 -0.05027809]
 [ 0.07359764 -0.07224277  0.00899941 -0.04095074  0.07024191 -0.03642267]
 ...,
 [-0.09237808  0.04140779  0.01757475  0.01541776  0.00677254  0.01375979]
 [ 0.02350667 -0.10410932  0.11869957 -0.10325475  0.0078093   0.06843098]
 [ 0.17586005 -0.02950553  0.17229628  0.20384649  0.02729918  0.06279307]]
Total number of Iterations: 4698
Theta shape =  (1025L, 6L)
Performance on validation set: 0.8
Performance on test set: 0.733333333333
```

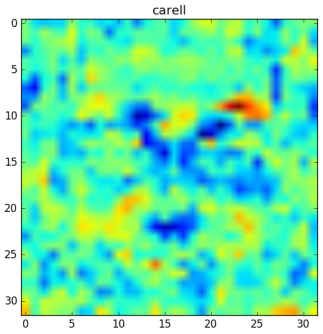
Figure 8: Output from part 7

Part 8

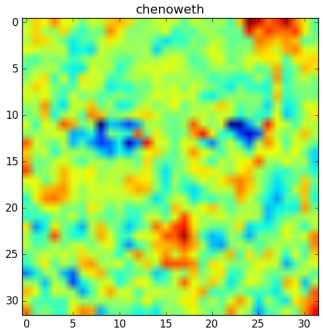
From the one-hot encoding, we could figure out which columns of the weight matrix corresponded to which actor. Essentially, each column of theta measures whether the image coressponds to a single actor or not. For example, Dreshcher was encoded with the label $[1, 0, 0, 0, 0, 0]$. Each column of theta weights the input with each corresponding column in the label vector. And consider the other actors where the first element of the their labels are all zero. Then the classifier can be thought of as classifying each actor individual, returning 1 or 0 if the actor is a certain label or not.



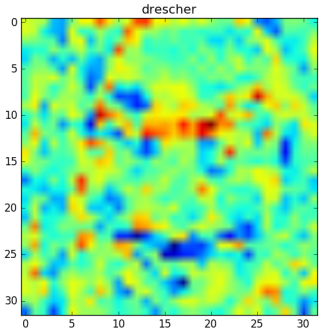
(a) Alec Baldwin



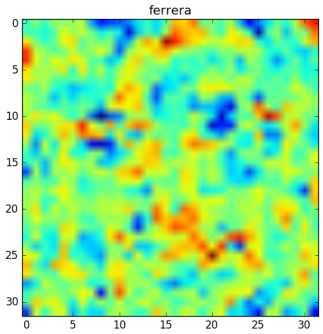
(b) Steve Carell



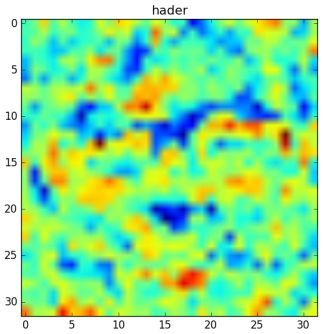
(c) Kristen Chenowith



(d) Dressher



(e) America Ferrera



(f) Bill Hader

Figure 9: Columns of Theta visualized.