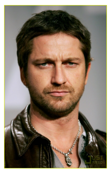
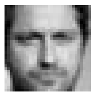
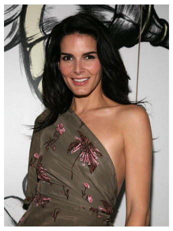
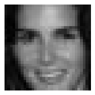
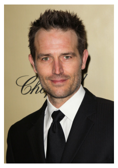
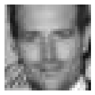
CSC411: Assignment 1

Due on Wednesday, February 1, 2017

Chloe Duan

February 4, 2017

Table 1: 5 Examples of Dataset Images with their cropped counterparts

Figure	Unrefined	Refined
Image 1		
Image 2		
Image 3		

Part 1

Describe the dataset of faces. In particular, provide at least three examples of the images in the dataset, as well as at least three examples of cropped out faces. Comment on the quality of the annotation of the dataset: are the bounding boxes accurate? Can the cropped-out faces be aligned with each other?

This dataset consists of images generated from photos extracted from online using a subset of the Facescrub database - (*get_data.py*). The raw images were later grayscaled, scaled to 32×32 as well as cropped according to the bounding boxes - (*resize_and_grayscale.py*). The cropped photos consist of just the face of the actor or actress.
The majority of the bounding boxes were accurate such as Images 1-3. However, a few images showed misbounding of the face, leaving the image cropped only at the eyes and nose. These images might effect the program performance slightly because they would not provide an effective comparison for the test data and might skew results. This is an example of something that can cause difficulty or error in the prediction performance. Looking at Images 1-3, the cropped-out faces can be aligned for comparison. Although this is a very small subset of the dataset and the ability to align images may vary from photo to photo, the majority of the cropped faces can be aligned to some degree.

Part 2

Separating Datasets

All images for an actor / actress are put into a matrix. The training, validation and test sets were separated randomly using the Random module by shuffling the images for each actor and and assigning 100, 10 and 10 images to the training, validation and test sets respectfully from the original set x. The y values are done in parallel with this assignment to keep track of the labels.
The sets are then returned to the calling function with their respective labels split into the set sizes specified.
This is implemented in the get_sets() function.
Note. All images are in folder called "filtered" in the same directory as faces.py.

Part 3

Linear Classifier: distinguish between Steve Carell and Bill Hader

Minimized Cost function: Least Squares
Training Set Cost Function Value: 27.506001423
Validation Set Cost Function Value: 3.77946337408
Training Set Classifier Performance: 0.85
Validation Set Classifier Performance: 0.80
Alpha: 5E-7
Code of the Linear Binary Classifier:

```
def grad_descent(f, df, x, y, init_t, alpha, dim_row, dim_col, multiclass=0):
    EPS = 1e-5   #EPS = 10**(-5)
    prev_t = init_t-10*EPS
    t = init_t.copy()
    max_iter = 3000
    iter  = 0
    while norm(t - prev_t) >  EPS and iter < max_iter:
        prev_t = t.copy()
        if multiclass:
            t -= alpha*df_multiclass(x, y, t)
        else:
            t -= alpha*df(x, y, t).reshape(dim_row+1, dim_col)
        if iter % 500 == 0:
            print "Iter", iter
            print "df = ", alpha*df(x, y, t)
            print "t = ", t
            print "Gradient: ", df(x, y, t), "\n"
        iter += 1
    return t

def linear_classifier(training_set, training_y, dim_row, dim_col, alpha=5E-7):
    theta = np.random.rand(dim_row+1, dim_col)*(1E-9)
    t = grad_descent(f, df, training_set.T,
    training_y, theta, alpha, dim_row, dim_col, multiclass=0)
    return t

def part3():
    TRAINING_SIZE = 100
    VAL_SIZE = 10
    TEST_SIZE = 10
    TOTAL_SIZE = TRAINING_SIZE + VAL_SIZE + TEST_SIZE

    # get all images
    actors = ["carell", "hader"]
    x, y, filename_to_img = get_imgs("filtered_male/", actors, TOTAL_SIZE)
    x /=255.

    training_set, training_y, validation_set, validation_y, test_set, test_y
    = get_sets(x, y, actors, filename_to_img, TRAINING_SIZE, VAL_SIZE, TEST_SIZE)

    # linear classifier y has one column
    training_y = reshape(training_y.T[1], (1, TRAINING_SIZE*len(actors)))
    validation_y = reshape(validation_y.T[1], (1, VAL_SIZE*len(actors)))
    test_y = reshape(test_y.T[1], (1, TEST_SIZE*len(actors)))

    t = linear_classifier(training_set, training_y, 1024, 1)

    train_p = performance(training_set.T, training_y, t, TRAINING_SIZE*len(actors))
    val_p = performance(validation_set.T, validation_y, t, VAL_SIZE*len(actors))
    test_p = performance(test_set.T, test_y, t, TEST_SIZE*len(actors))

    print("TRAIN PERFORMANCE: %f", train_p*100)
    print("VALIDATION PERFORMANCE: %f", val_p*100)
    print("TEST PERFORMANCE: %f", test_p*100)
```

System Logistics:
First, the input images were normalized to values between 0 and 1. This helped to scale the affect of alpha within the gradient descent to a reasonable amount.
I found that if alpha is too large, the performance of the classifier significantly decreases when convergence does occur. Otherwise, there would be no convergence and gradients would lean towards infinity. If the alpha was too small, the performance would also be negligable although the theta's would look more like faces. At times, when the algorithm wouldn't converge under 30000 iterations, I would adjust the EPS to a larger value or decrease max_iterations. In these cases, the performances did turn out okay.

This is implemented in the part3() function in faces.py.

Part 4

Figure 1: Training size = 100

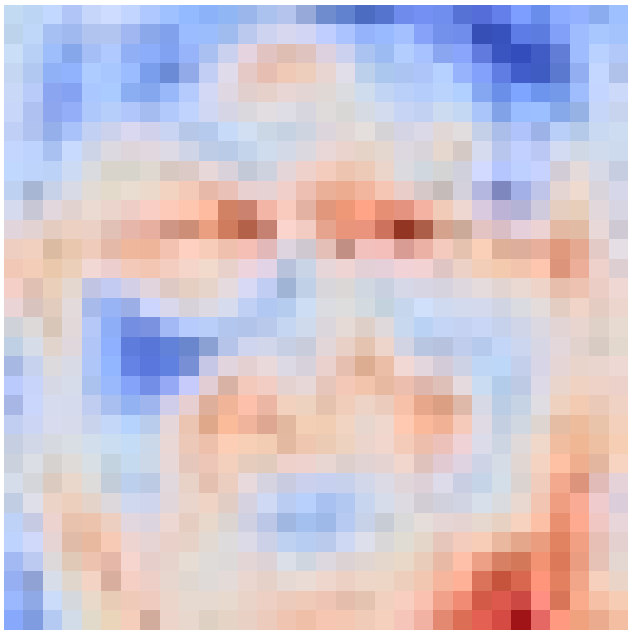
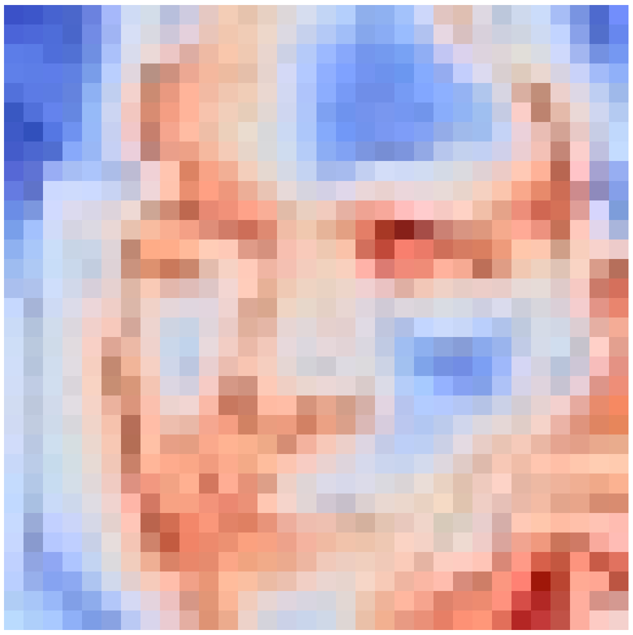


Figure 2: Training size = 2

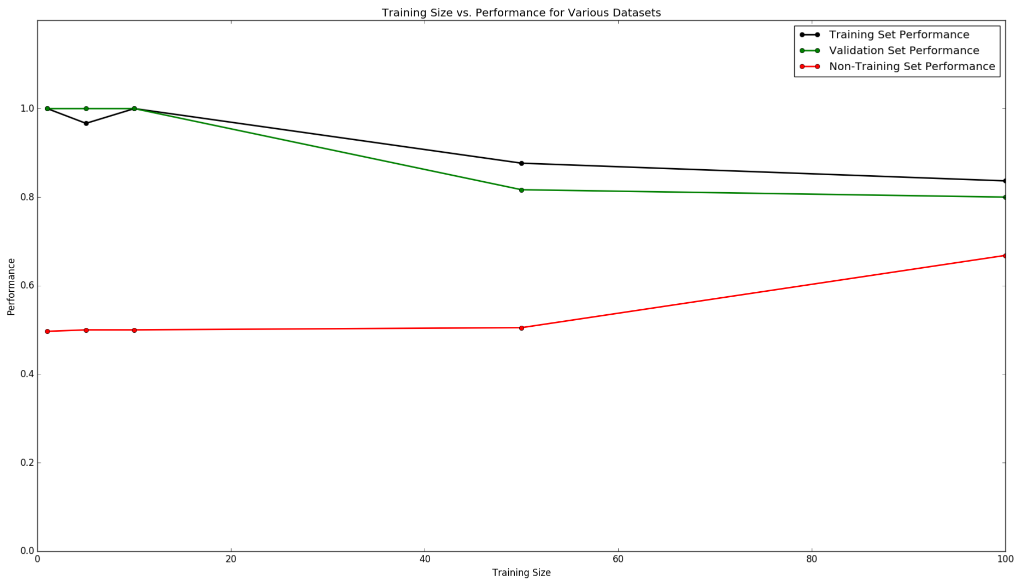


This is implemented in the part3() function in faces.py.

Part 5

Plotted Training Set Size vs Classifier Performance on Training, Validation and non-act Actor Sets

Classifier Performance on actors not in act



This is implemented in the part5() and part5$_p$lot() functions in faces.py.

Part 6

Part A

Compute $\partial J/\partial\theta_{pq}$.
$J(\theta) = \sum_{i=1}^{m}(\theta_0^{(i)}x_0^{(i} + \theta_1^{(i)}x_1^{(i} + ... + \theta_q^{(i)}x_q^{(i} + ... + \theta_k^{(i)}x_k^{(i} - y_0^{(i)} - y_1^{(i)} - ... - y_q^{(i)} ... - y_k^{(i)})^2$
$\partial J/\partial(\theta_{pq}) = \sum_{i=1}^{m}\partial/\theta_{pq}(\theta_0^{(i)}x_0^{(i)} + \theta_1^{(i)}x_1^{(i)} + ... + \theta_q^{(i)}x_q^{(i)} + ... + \theta_k^{(i)}x_k^{(i)} - y_0^{(i)} - y_1^{(i)} - ... - y_q^{(i)} ... - y_k^{(i)})^2$
$= \partial/\theta_{pq}(\theta_0^{(p)}x_0^{(p)} + \theta_1^{(p)}x_1^{(p)} + ... + \theta_q^{(p)}x_q^{(p)} + ... + \theta_k^{(p)}x_k^{(p)} - y_0^{(p)} - y_1^{(p)} - ... - y_q^{(p)} ... - y_k^{(p)})^2$
$= 2(\theta_q^{(p)}x_q^{(p)} - y_q^{(p)})x_q^{(p)}$

Part B

k - labels
m - training samples
X Dimensions: (1+32*32, m) - 1 - row of 1s
Y Dimensions: (k, m)
θ Dimensions: (1+32*32, k) - 1 for θ_0

Given 2X(θ^TX-Y)^T.

Then given the dimensions specified:
$(1025, m) \times (((k, 1025) \times (1025, m)) - (k, m))^T$
$= (1025, m) \times (m, k)$
$= (1025, k)$ - which is the dimension for θ

Part C

Implemented Cost Function Vectorized Gradient Function:

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

Part D

Demonstrate the vectorized gradient function works by computing several components of the gradient using finite differences.

Code used to compute gradient components using finite differences:

```
def check_grad_multiclass(x, y, theta, coords):
    for coord in coords:
        h_r_c = 0.000001
        h = np.zeros([1025, 6])
        h[coord[0],coord[1]] = h_r_c
        x = x.T.reshape(1024, 1)
        y = y.reshape(1,6)
        print "Validating gradient function at: ", coords
        print "\t Finite Difference=",
            (f_multiclass(x, y, theta+h) - f_multiclass(x, y, theta-h))/(2*h_r_c)
        print "\t df[{},{}]= {}".format(coord[0], coord[1],
            df_multiclass(x, y, theta)[coord[0], coord[1]])
```

Finite Differences Gradient vs. Implemented GD

```
>>> check_grad_multiclass(training_set[1], training_y[1], t, [(2,3), (100,5), (500,5)])
Validating gradient function at:  [(2, 3), (100, 5), (500, 5)]
      Finite Difference= 0.162464167786
      df[2,3]= 0.162464167795
Validating gradient function at:  [(2, 3), (100, 5), (500, 5)]
      Finite Difference= 0.0514390513517
      df[100,5]= 0.0514390513723
Validating gradient function at:  [(2, 3), (100, 5), (500, 5)]
      Finite Difference= 0.0610303697302
      df[500,5]= 0.0610303697323
```

Part 7

Training Set Performance: 88.5
Validation Set Performance: 71.6
Alpha: 1.5E-6
Indicate what parameters you chose for gradient descent and why they seem to make sense.
The alpha value increased compared to the alpha values of the previous cases. This could be explained by the addition of more grooves within the gradient surface. The larger alpha value may optimize the minimization of the cost function because it allows for a wider search over the surface for a minimum.

This is implemented in the part7() function in faces.py.

Part 8

Visualized θs obtained

Figure 3: Fran Drescher

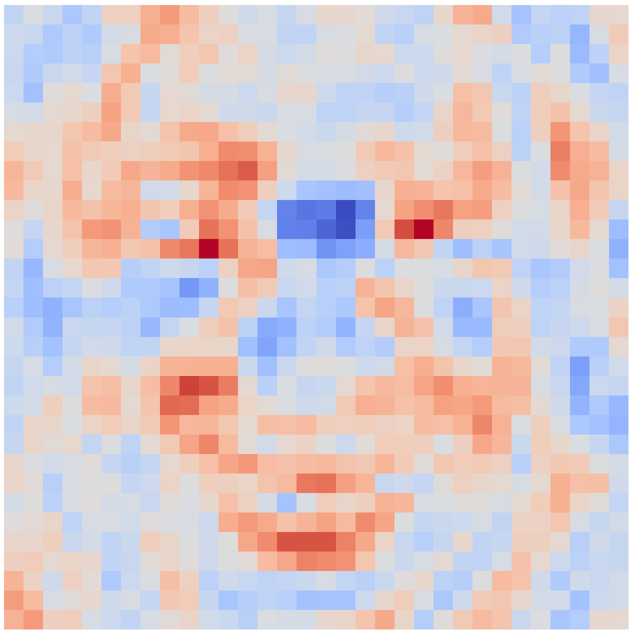


Figure 4: America Ferrera

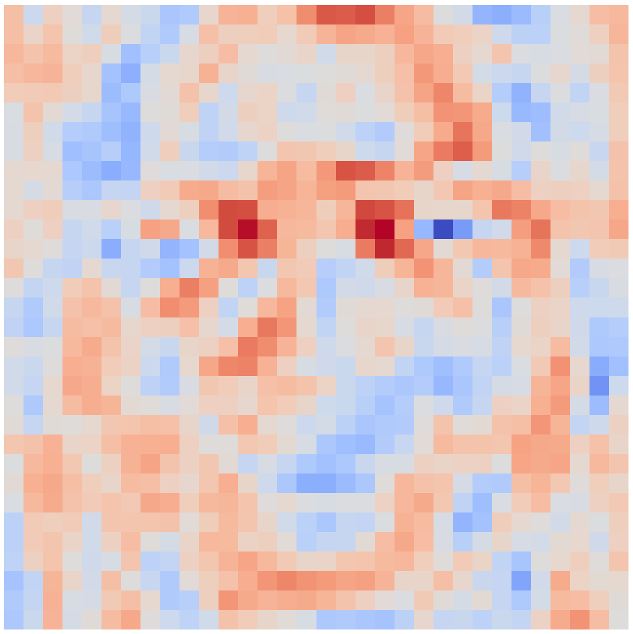


Figure 5: Kristin Chenoweth

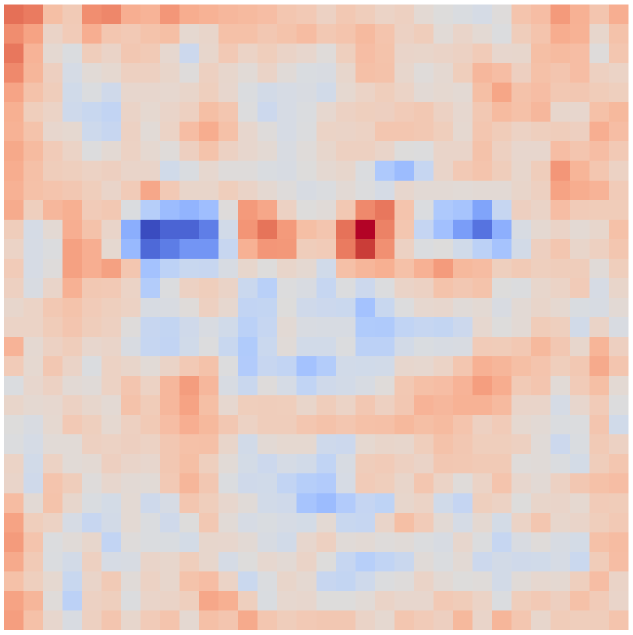


Figure 6: Alec Baldwin

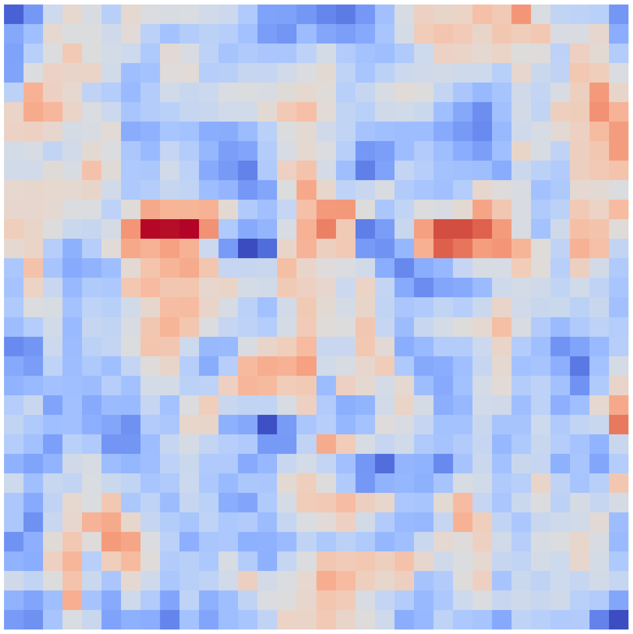


Figure 7: Bill Hader

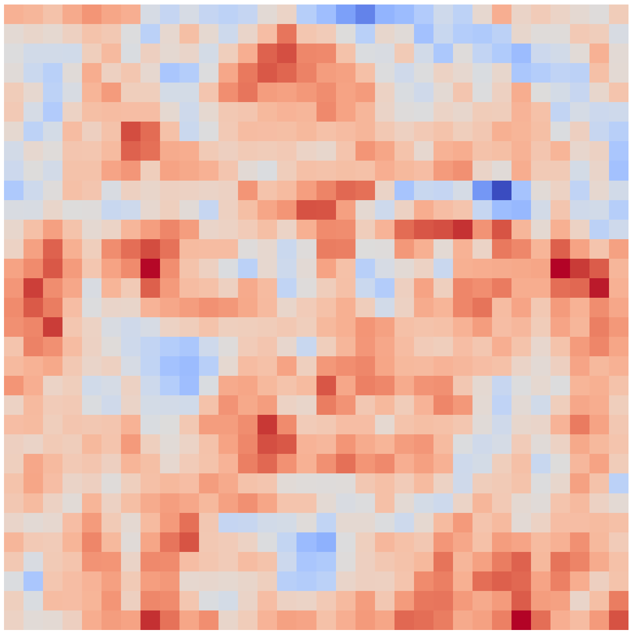


Figure 8: Steve Carell