CSC411 Project 1

Shichen Lu

January 29, 2018

Introduction

The python code associated with this report is included in three separate files: main.py, learner.py, and get_data.py. It is written in Python 3.6.4 and references image files that are included in the folder "uncropped". This folder should be placed in the directory that the program is run from.

Part 1

The images were taken from the online "FaceScrub" database, which can be found at http://vintage.winklerbros.net/facescrub.html. It contains images of various people as well as information for where on the image the person's face is located. For this project, we have taken a set of images from certain actors from the website to use in an image classification project. To evaluate the quality of the images collected, some images from the dataset are produced below:

(a) Alec Baldwin

(b) Lorraine Bracco

(c) Daniel Radcliffe

Figure 1: Uncropped Images

Based on the data given by the database, the images are automatically cropped, resized, and converted to grayscale to produce the following:

(a) Alec Baldwin

(b) Lorraine Bracco

(c) Daniel Radcliffe

Figure 2: Cropped Images

As we can see, there is varying data in the quality of the images. While most images are head-on images of the actor which produce good quality faces when cropped, such as the Daniel Radcliffe example given above, there are

a small amount of images in the database where the actors are shot from the side, or are not directly facing the camera, as seen in the Alec Baldwin example. Additionally, there are also a small amount of pictures in which the provided information to crop out the actor's face is completely wrong, as seen in the Lorraine Bracco example. However, the amount of images that fall into the third category is very small (estimated $< 1\%$). Overall, the quality of cropped images are fairly good.

Part 2

Initially, we will analyise six total actors: Lorraine Bracco Peri Gilpin Angie Harmon Alec Baldwin Bill Hader Steve Carell.

The following algorithm was used to separate the cropped face images of each actor:

```
# Generates and returns training, validation, and test sets
    ↪ for each actor
def generate_sets(actors):
    extensions = [".jpg", ".JPG", ".png", ".PNG", ".jpeg", ".
        ↪ JPEG"]
    image_counts = image_count("./cropped")
    training_sets = {key: [] for key in actors}
    validation_sets = {key: [] for key in actors}
    test_sets = {key: [] for key in actors}
    for actor in actors:
        for i in range(image_counts[actor] - 20):
            for extension in extensions:
                if (os.path.isfile("./cropped/" + actor.split()
                    ↪ [1].lower() + str(i) + extension)):
                    training_sets[actor].append((actor, actor.
                        ↪ split()[1].lower() + str(i) +
                        ↪ extension))
        for i in range(image_counts[actor] - 20, image_counts[
            ↪ actor] - 10):
            for extension in extensions:
                if (os.path.isfile("./cropped/" + actor.split()
                    ↪ [1].lower() + str(i) + extension)):
                    validation_sets[actor].append((actor, actor.
                        ↪ split()[1].lower() + str(i) +
                        ↪ extension))
        for i in range(image_counts[actor] - 10, image_counts[
            ↪ actor]):
            for extension in extensions:
```

```
            if (os.path.isfile("./cropped/" + actor.split()
                ↪ [1].lower() + str(i) + extension)):
                test_sets[actor].append((actor, actor.split
                    ↪ ()[1].lower() + str(i) + extension))
    return (training_sets, validation_sets, test_sets)
```

After getting the images of each actor from the database, this function is ran with an argument that dictates the list of actors to generate test, validation, and training sets for. The function uses helper function "image_count" to count how many images for each actor we were able to collect from the database (as some links on the database are dead). This "image_count" is reproduced below:

```
# Returns a dictionary that lists how many images of each
    ↪ actor are present in a directory
def image_count(path):
    res = {key: 0 for key in actors}
    for file in os.listdir(path):
        for actor in actors:
            if file.startswith(actor.split()[1].lower()):
                res[actor] += 1
    return res
```

Overall, for each actor, the function simply puts the last ten collected images into the test set, the 20th to 11th last collected images into the validation set, and the rest of the images into the training set. Note that although the images are split the same way for each time the program is run, due to how the datacollection parses the database images, the images are not collected in the same order each time the program is run. So the generated sets are not necessarily the same for each time the program is run.

Part 3

The following code was called to generate a classifier through Linear Regression for distinguishing between pictures of Alec Baldin and pictures of Steve Carell:

```
# Part 3: Steve Carell vs Alec Baldwin
# Steve Carell: 1
# Alec Baldwin: -1
print("\n\n >>>PART 3<<<")

x,y,thetas = learner.generate_xyt(training_sets["Steve Carell
    ↪ "] + training_sets["Alec Baldwin"], [1 for i in range(
```

```
    ↪ len(training_sets["Steve Carell"]))] + [-1 for i in
    ↪ range(len(training_sets["Alec Baldwin"]))])
thetas_p3 = learner.grad_descent(learner.quad_loss, learner.
    ↪ quad_loss_grad, x, y, thetas, 0.001, 10000)

testactors_p3 = {key:test_sets[key] for key in ["Alec Baldwin
    ↪ ", "Steve Carell"]}
testanswers_p3 = {"Alec Baldwin": np.array((-1)), "Steve
    ↪ Carell": np.array((1))}
learner.test(testactors_p3, testanswers_p3, thetas_p3)
```

This code uses three worker functions:

1. learner.generate_xyt to generate the appropriate numpy arrays for the initial x, y and theta inputs to the gradient descent function

```
# Generates corresponding x's, y's and thetas for
    ↪ gradient descent function
# Takes sorted input of training images and their
    ↪ corresponding training label
def generate_xyt(input_sets, labels):
    x = np.zeros((len(input_sets), 1025))
    try:
        y = np.zeros((len(input_sets),len(labels[0])))
        thetas = zeros((1025, len(labels[0])))
    except:
        y = np.zeros((len(input_sets), 1))
        thetas = zeros((1025, 1))
    for i in range(len(input_sets)):
        imdata = (imread("./cropped/" + input_sets[i][1])
            ↪ / 255).reshape(1024)
        imdata = np.concatenate(([1], imdata))
        x[i] = imdata
        y[i] = labels[i]
    return x.T, y, thetas
```

2. learner.gradient_descent which implements gradient descent using a quadratic cost funtion to minimize the thetas (this was modified slightly from the version on the CSC411 course website)

```
# Gradient descent function. Taken from CSC411 website.
def grad_descent(f, df, x, y, init_t, alpha, _max_iter):
    print("------------------------- Starting Grad Descent
        ↪  -------------------------")
    EPS = 1e-5 #EPS = 10**(-5)
```

```
    prev_t = init_t-10*EPS
    t = init_t.copy()
    max_iter = _max_iter
    iter = 0
    while norm(t - prev_t) > EPS and iter < max_iter:
        prev_t = t.copy()
        grad = df(x, y, t, x.shape[1])
        t -= alpha*(grad)
        if iter % 1000 == 0:
            print("Iter %i: cost = %.2f" % (iter, f(x, y, t
                ↪ )))
            #print("Gradient: ", grad, "\n")
        iter += 1
    return t
```

3. learner.test, which tests our linear regression results on the testing sets and reports the results

```
# Test a set of thetas for their accuracy on the test set
# Takes in a dictionary of test sets per actor, a
    ↪ dictionary of the corresponding correct answer for
    ↪ each actor,
# and the computed thetas
def test(test_sets, answers, thetas):
    correct = 0
    count = 0
    conc_sets = []
    print("------------------------ Testing
        ↪ -------------------------")
    for actor in test_sets:
        i = 0
        conc_sets += test_sets[actor]
        for image in test_sets[actor]:
            imdata = (imread("./cropped/" + image[1]) /
                ↪ 255).reshape(1024)
            imdata = np.concatenate(([1], imdata))
            prediction = np.dot(imdata, thetas)
            print("%s %i|pred:" % (actor, i), end=" ")
            print(prediction, end=" ")
            print("ans:", end=" ")
            print(answers[actor], end=" ")
            min = sum(abs(prediction - answers[actor]))
            guess = actor
            for answer in answers:
```

```
            if sum(abs(prediction - answers[answer])) <
                ↪ min:
                min = sum(abs(prediction - answers[
                    ↪ answer]))
                guess = answer
        if guess == image[0]:
            correct += 1
            print("correct!")
        else:
            print("incorrect!")
        i += 1
        count += 1
conc_sets_labels = [0 for i in range(len(conc_sets))]
for i in range(len(conc_sets)):
    conc_sets_labels[i] = answers[conc_sets[i][0]]
x, y, t = generate_xyt(conc_sets, conc_sets_labels)
print("Cost: %.2f" % quad_loss(x, y, thetas, x.shape
    ↪ [1]))
print("Score: %.2f" % (correct / count))
return (correct / count)
```

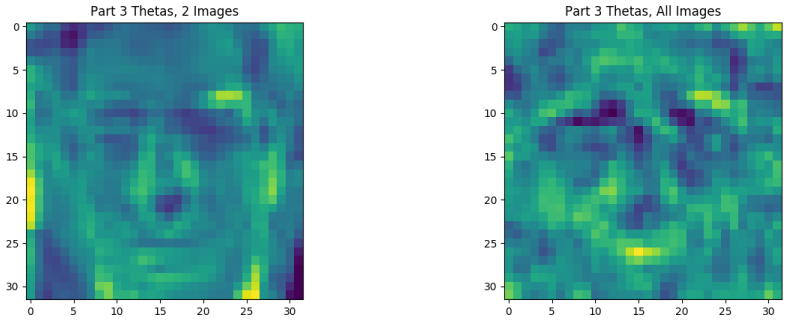
The values used for the alpha and number of iterations in gradient descent were 0.001 and 10000 respectively. It was noted that any alpha greater than 0.008 would cause an overflow error during the first iteration of the gradient descente process and crash the program. Additionally, it was found that running around 10000 iterations of the gradient descent process produced the best results, and using a greater number of iterations actually decreased the accuracy of the algorithm.

In the end, the algorithm was able to hit a 90% accuracy with a 0.33 cost on the testing sets, and 95% accuracy with a 0.24 cost on the validation sets.

Part 4

Subsection A

We can visualize the thetas from Part 3 as an image itself. By using just two images from each actor as the training set, we obtain an set of thetas that, when visualized, very closely resembles the face of one of the actors, as seen in Figure 3(a). On the other hand, by using a the full training set for each actor, we were able to obtain thetas that look fairly random when visualized.
One interesting thing to note is that when we use just 2 images, the gradient descent function stops very early due to reaching the EPS bound. This may



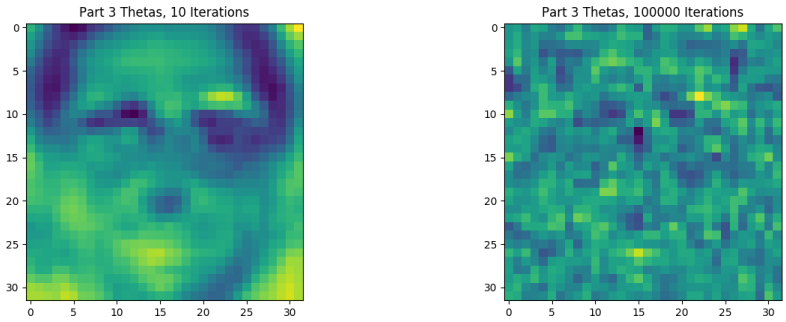
(a) Using 2 Images (b) Using All Images

Figure 3: Visualized Thetas while Varying Training Set

be a factor in explaining the resemblence between the visualizations in Figure 3(a) and Figure 4(a).

Subsection B

By using a relatively low amount of iterations (around 10) of gradient descent, we were able to obtain a set of thetas that, when visualied, somewhat resembles a face, as seen in Figure 4(a).



(a) Using 10 Iterations (b) Using 100000 Iterations

Figure 4: Visualized Thetas while Varing Amount of Iterations

On the other hand, by using a high amount of iterations (around 100000) in the gradient descent, we were able to obtain a set of thetas that looked a bit more random, as seen in Figure 4(b).

Part 5

Using the following code, male-female classifiers were built and tested using images of the 6 actors mentioned in Part 2

```
# Part 5: Overfitting
# Male: 1
# Female: -1
print("\n\n >>>PART 5<<<")

testanswers_p5 = {"Steve Carell": np.array((1)), "Alec Baldwin
    ↪ ": np.array((1)), "Bill Hader": np.array((1)),
                  "Lorraine Bracco": np.array((-1)), "Peri
                      ↪ Gilpin": np.array((-1)), "Angie Harmon
                      ↪ ": np.array((-1))}

test_results_training = np.zeros((22,1))
test_results_validation = np.zeros((22,1))
for i in range (22):
    print("\nUsing %i Training Images" % ((i+1)*5))
    training_set_orig_6 = []
    labels_malefemale = []

    for actor in actors_orig:
        training_set_orig_6 += training_sets[actor][:(i+1)*5]
        if actor in ["Steve Carell", "Alec Baldwin", "Bill
            ↪ Hader"]:
            labels_malefemale += [1 for i in range(len(
                ↪ training_sets[actor][:(i+1)*5]))]
        elif actor in ["Lorraine Bracco", "Peri Gilpin", "Angie
            ↪  Harmon"]:
            labels_malefemale += [-1 for i in range(len(
                ↪ training_sets[actor][:(i+1)*5]))]

    x, y, thetas = learner.generate_xyt(training_set_orig_6,
        ↪ labels_malefemale)
    thetas_p5 = learner.grad_descent(learner.quad_loss, learner
        ↪ .quad_loss_grad, x, y, thetas, 0.001, 100000)

    training_sets_p5 = {key: training_sets[key][:(i+1)*5] for
        ↪ key in actors_orig}
    validation_sets_p5 = {key:validation_sets[key] for key in
        ↪ actors_orig}
    test_results_training[i] = learner.test(training_sets_p5,
```

```
        ↪ testanswers_p5, thetas_p5, False)
    test_results_validation[i] = learner.test(
        ↪ validation_sets_p5, testanswers_p5, thetas_p5, False
        ↪ )

plt.plot(range(5,115,5), test_results_validation*100, label="
    ↪ Validation")
plt.plot(range(5,115,5), test_results_training*100, label="
    ↪ Training")
plt.ylabel("% Correct")
plt.xlabel("Number of Training Images Used")
plt.axis([0,115,0,110])
plt.legend()
plt.title("Part 5 Test Results for Varying Number of Training
    ↪ Images")
plt.show()

testactors_p5b = {key: test_sets[key] for key in actors_new}
testanswers_p5b = {"Michael Vartan": np.array((1)), "Gerard
    ↪ Butler": np.array((1)), "Daniel Radcliffe": np.array((1)
    ↪ ),
                   "Kristin Chenoweth": np.array((-1)), "America
                       ↪  Ferrera": np.array((-1)),
                   "Fran Drescher": np.array((-1))}
learner.test(testactors_p5b, testanswers_p5b, thetas_p5)
```

A plot of classifier performance on the validation and training sets versus the number of images used in the training set is produced below. Note that in this case, gradient descent was ran with an alpha of 0.001 and an enforced iteration number of 100000.
We can see that while the accuracy of the classifier generally increased as the number of images in the training set increased, there was a slight decrease of accuracy as we neared a large number of training sets. Specifically, the performance on the training set decreased from a constant 100% accuracy to around a 98% accuracy.

The performance of the classifier was also tested on a set of six different actors, using the thetas obtained from training on the largest amount of images in the training set. Here, we were able to achieve a 85% accuracy.

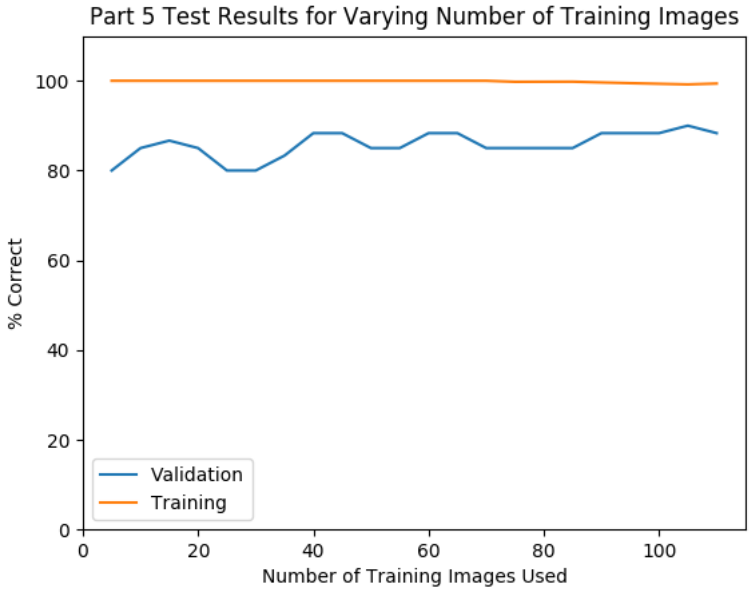


Figure 5: Part 5 Classifier

Part 6

Subsection A

We have

$$J(\theta) = \sum_i (\sum_j (\theta^T x^{(i)} - y^{(i)})_j^2) \tag{1}$$

We can use the chain rule compute the partial derivative with respect to θ_{pq} (ie. to each individual theta element in the vector) as

$$\frac{\partial J}{\partial \theta_{pq}} = 2 \sum_i (\sum_j (\theta^T x^{(i)} - y^{(i)})_j * \frac{\partial (\theta^T x^{(i)})_j}{\partial \theta_{pq}}) \tag{2}$$

Additionally, we know that as long as $j \neq q$ we have $\frac{\partial (\theta^T x^{(i)})_j}{\partial \theta_{pq}} = 0$. Additionally, when we have $j = q$, then $\frac{\partial (\theta^T x^{(i)})_j}{\partial \theta_{pq}} = x_p^{(i)}$ so in the end we can remove the sum over j and we are left with

$$\frac{\partial J}{\partial \theta_{pq}} = 2 \sum_i ((\theta^T x^{(i)} - y^{(i)})_q * x_p^{(i)}) \tag{3}$$

Subsection B

Let us say that n represents the number of features per image (ie. number of pixels + 1), k represents the number of actors we are classifying between

(in this case, 6), and m is the number of training examples. Then we have the following matricies:

1. X, which is a n by m matrix that holds all the training examples
2. θ, which is a n by k matrix that holds our thetas
3. Y, which is a k by m matrix that holds our labels for each training example

Now, to show part 6(b), let us first note that

$$\frac{\partial J}{\partial \theta_{pq}} = 2\sum_i ((\theta^T x^{(i)} - y^{(i)})_q * x_p^{(i)}) = 2(\theta^T X - Y)_q X_p \tag{4}$$

Where the subscripts p and q represent the rows of their respective matricies.

Now, to find the gradient for all thetas, we have

$$\frac{\partial J}{\partial \theta} = \begin{bmatrix} \frac{\partial J}{\partial \theta_{11}} & \cdots & \frac{\partial J}{\partial \theta_{1k}} \\ \cdots & \ddots & \\ \frac{\partial J}{\partial \theta_{n1}} & & \frac{\partial J}{\partial \theta_{nk}} \end{bmatrix} \tag{5}$$

Now, subbing (4) into (5) gives us

$$\frac{\partial J}{\partial \theta} = \begin{bmatrix} 2(\theta^T X - Y)_1 X_1 & \cdots & 2(\theta^T X - Y)_k X_1 \\ \cdots & \ddots & \\ 2(\theta^T X - Y)_1 X_n & & 2(\theta^T X - Y)_k X_n \end{bmatrix} \tag{6}$$

Here, we can factor out the X_p from each entry to obtain

$$\frac{\partial J}{\partial \theta} = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix} \begin{bmatrix} 2(\theta^T X - Y)_1 & \cdots & 2(\theta^T X - Y)_k \end{bmatrix} \tag{7}$$

Which easily becomes

$$\frac{\partial J}{\partial \theta} = 2X(\theta^T X - Y)^T \tag{8}$$

As desired

Subsection C

The code for the vectorized gradient function is reproduced below

```
# Gradient of quadratic loss function
def quad_loss_grad(x, y, theta, norm_const):
    # x = vstack((ones((1, x.shape[1])), x))
    return -2 * dot(x, (y.T - dot(theta.T, x)).T) / norm_const
```

Subsection D

We use the finite-difference approximation to check if our partial derivatives are correct. This is accomplished by additing a small variation, h, to a randomly chosen θ_{pq} in our current θ_{curr} to generate a new θ_{new}. We then estimate the gradient using the following formula:

$$cost'(\theta_{curr}) = \frac{cost(\theta_{new}) - cost(\theta_{curr})}{h} \tag{9}$$

Then, by looking at the estimated gradient in the $[p][q]$ index of our calculated result, and comparing it to the result we obtain from matrix multiplication, we can see how close our results are.
The code to perform the above is reproduced below.

```
# Estimates the gradient using a finite difference formula
def grad_est(x, y, theta, norm_const, cost, cost_grad):
    np.random.seed(0)
    EPS = 0.0001
    for i in range(5):
        j = np.random.random_integers(1, 1000)
        act = cost_grad(x, y, theta, norm_const)[j][i]
        h = 0.0001
        prev_est = 999999
        new_theta = np.copy(theta)
        new_theta[j][i] = new_theta[j][i] + h
        est = (cost(x, y, new_theta, norm_const) - cost(x, y,
            ↪ theta, norm_const)) / h
        while abs(prev_est - est) > EPS:
            prev_est = est
            h = h/2
            new_theta = np.copy(theta)
            new_theta[j][i] = new_theta[j][i] + h
            est = (cost(x, y, new_theta, norm_const) - cost(x,
                ↪ y, theta, norm_const)) / h
        print("Grad Difference for theta[%i][%i]: %f" % (j, i,
            ↪ abs(est - act)))
```

Note that we only selectively test 5 different indicies of the θ, as per the problem prompt, and we are testing this on the results from Part 7. Additionally, we select the correct h to use by iteratively computing estimated gradients using smaller and smaller h values until the difference between two successive computations of the estimated gradient is within our error bound. This ensures that we are not using too large of an h value and wrongly estimating our gradient.
The results from running this function on the θ's obtained from Part 7 are produced below, and we can see that our vector multiplication estimation

of the gradient is actually very close to the finite-difference estimation, indicating that our method is working.

```
Grad Difference for theta[685][0]: 0.000019
Grad Difference for theta[560][1]: 0.000024
Grad Difference for theta[630][2]: 0.000024
Grad Difference for theta[193][3]: 0.000007
Grad Difference for theta[836][4]: 0.000011
```

Part 7

The following code was used to run gradient descent to classify between the 6 original actors:

```
# Part 7: Multiple Actor Classification
# Alec Baldwin: [1,0,0,0,0,0]
# Steve Carell: [0,1,0,0,0,0]
# Bill Hader: [0,0,1,0,0,0]
# Lorraine Bracco: [0,0,0,1,0,0]
# Angie Harmon: [0,0,0,0,1,0]
# Peri Gilpin: [0,0,0,0,0,1]
print("\n\n >>>PART 7<<<")

training_set_orig_6 = []
labels_by_actor = []

for actor in actors_orig:
    training_set_orig_6 += training_sets[actor]
    if actor == "Alec Baldwin":
        labels_by_actor += [[1, 0, 0, 0, 0, 0] for i in range(
            ↪ len(training_sets[actor]))]
    elif actor == "Steve Carell":
        labels_by_actor += [[0, 1, 0, 0, 0, 0] for i in range(
            ↪ len(training_sets[actor]))]
    elif actor == "Bill Hader":
        labels_by_actor += [[0, 0, 1, 0, 0, 0] for i in range(
            ↪ len(training_sets[actor]))]
    elif actor == "Lorraine Bracco":
        labels_by_actor += [[0, 0, 0, 1, 0, 0] for i in range(
            ↪ len(training_sets[actor]))]
    elif actor == "Angie Harmon":
        labels_by_actor += [[0, 0, 0, 0, 1, 0] for i in range(
            ↪ len(training_sets[actor]))]
    elif actor == "Peri Gilpin":
```

```
        labels_by_actor += [[0, 0, 0, 0, 0, 1] for i in range(
            ↪ len(training_sets[actor]))]

x, y, thetas = learner.generate_xyt(training_set_orig_6,
    ↪ labels_by_actor)
thetas_p7 = learner.grad_descent(learner.quad_loss, learner.
    ↪ quad_loss_grad, x, y, thetas, 0.002, 10000)

# Part 7: Multiple Actor Classification
# Alec Baldwin: [1,0,0,0,0,0]
# Steve Carell: [0,1,0,0,0,0]
# Bill Hader: [0,0,1,0,0,0]
# Lorraine Bracco: [0,0,0,1,0,0]
# Angie Harmon: [0,0,0,0,1,0]
# Peri Gilpin: [0,0,0,0,0,1]
print("\n\n >>>PART 7<<<")

training_set_orig_6 = []
labels_by_actor = []

for actor in actors_orig:
    training_set_orig_6 += training_sets[actor]
    if actor == "Alec Baldwin":
        labels_by_actor += [[1, 0, 0, 0, 0, 0] for i in range(
            ↪ len(training_sets[actor]))]
    elif actor == "Steve Carell":
        labels_by_actor += [[0, 1, 0, 0, 0, 0] for i in range(
            ↪ len(training_sets[actor]))]
    elif actor == "Bill Hader":
        labels_by_actor += [[0, 0, 1, 0, 0, 0] for i in range(
            ↪ len(training_sets[actor]))]
    elif actor == "Lorraine Bracco":
        labels_by_actor += [[0, 0, 0, 1, 0, 0] for i in range(
            ↪ len(training_sets[actor]))]
    elif actor == "Angie Harmon":
        labels_by_actor += [[0, 0, 0, 0, 1, 0] for i in range(
            ↪ len(training_sets[actor]))]
    elif actor == "Peri Gilpin":
        labels_by_actor += [[0, 0, 0, 0, 0, 1] for i in range(
            ↪ len(training_sets[actor]))]

x, y, thetas = learner.generate_xyt(training_set_orig_6,
    ↪ labels_by_actor)
thetas_p7 = learner.grad_descent(learner.quad_loss, learner.
```

```
    ↪ quad_loss_grad, x, y, thetas, 0.003, 10000)

testactors_p7 = {key: test_sets[key] for key in actors_orig}
testanswers_p7 = {"Alec Baldwin": [1, 0, 0, 0, 0, 0], "Steve
    ↪ Carell": [0, 1, 0, 0, 0, 0],
                "Bill Hader": [0, 0, 1, 0, 0, 0], "Lorraine
                    ↪ Bracco": [0, 0, 0, 1, 0, 0],
                "Angie Harmon": [0, 0, 0, 0, 1, 0], "Peri
                    ↪ Gilpin": [0, 0, 0, 0, 0, 1]}
learner.test(testactors_p7, testanswers_p7, thetas_p7)
```

For this part, an alpha of 0.003 and max iterations of 10000 was used for the training. Due to the multiple classification going on, a higher alpha value that previous was needed for better performance. This training obtained an accuracy of 80% on the validation set and 96% on the training set.
To obtain the label from the output of the model, the learner.test function, which was mentioned previously in Part 3, was used. Essentially, we take the output of the classifier and see which index in the output array is closest to 1, and set the corresponding actor to that index that as the "guess" that our classifier produces.

Part 8

Again, we visualize the thetas obtained from the classifier. See below.

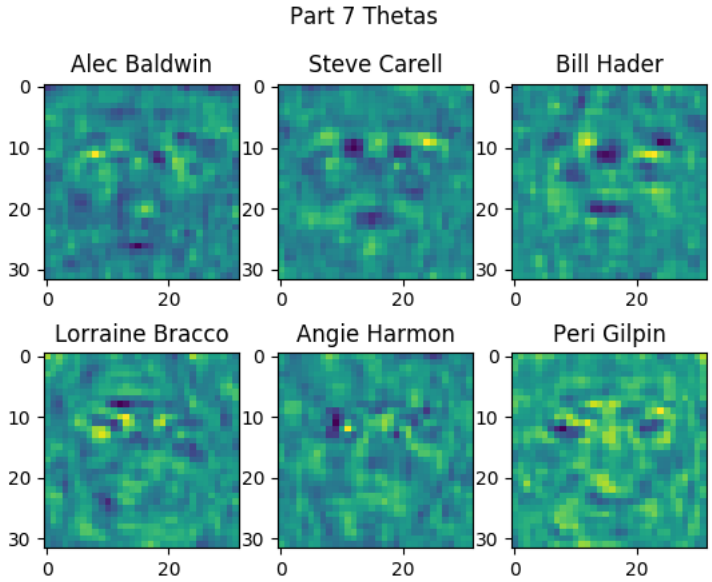


Figure 6: Part 7 Theta Visualization