CSC2515: Project #1: Face Recognition and Gender Classification with Regression

Due on Monday, January 29, 2018

Tianyi Chen

January 28, 2018

Part 0

Code configuration

The code for this project is written in Python 3.6. It requires a txt file *faces_subset.txt* in the same directory as the code to run. *faces_subset.txt* is provided and should remain unchanged.

Part 1

Dataset description

faces_subset.txt contains the names of all 12 actors (i.e. Lorraine Bracco, Peri Gilpin, Angie Harmon, Alec Baldwin, Bill Hader, Steve Carell, Daniel Radcliffe, Gerard Butler, Michael Vartan, Kristin Chenoweth, Fran Drescher and America Ferrera), links to the images, and the coordinates of the face in each image. Use the *download()* function in *part_0.py* to download the images of certain actors from the links and to crop and resize the images. The images are reshaped into vectors of *1024*1* when downloaded. The cropped images of a particular actor are stored as an *m*1024* numpy array (m is the number of images), where each row represents one image. These arrays are stored as elements of the list X, i.e. each element of X is an array of all images of a particular actor.

To explore the image data, first download the images of Lorraine Bracco, Peri Gilpin, Angie Harmon, Alec Baldwin, Bill Hader and Steve Carell. Figure 1 shows three examples of the images in the dataset. Randomly select one cropped image for each actor to display. We get the images as shown in Figure 2.

(a) Baldwin

(b) Bracco

(c) Carell

Figure 1: A random selection of images for each actor

In total, 176 cropped images of Alec Baldwin have been obtained. One of them has inaccurate bounding boxes (as shown in Figure 3). The number of cropped images obtained for Lorraine Bracco, Peri Gilpin, Angie Harmon, Bill Hader, and Steve Carell are: 143, 120, 137, 177 and 182 respectively. The average accuracy of the bounding boxes (the number of correct images divided by the total number of images obtained) is roughly 99.0%. In general, the bounding boxes are quite accurate. This means that the image data is in good quality and can be used for further classification tasks. Most of the faces are aligned with each other in terms of the positions of nose, eyes, etc. Whereas, in some of the photos the person is facing a different direction. Overall, the aligness of the faces is good. See Figure 4 for examples of aligned faces and slightly unaligned faces.

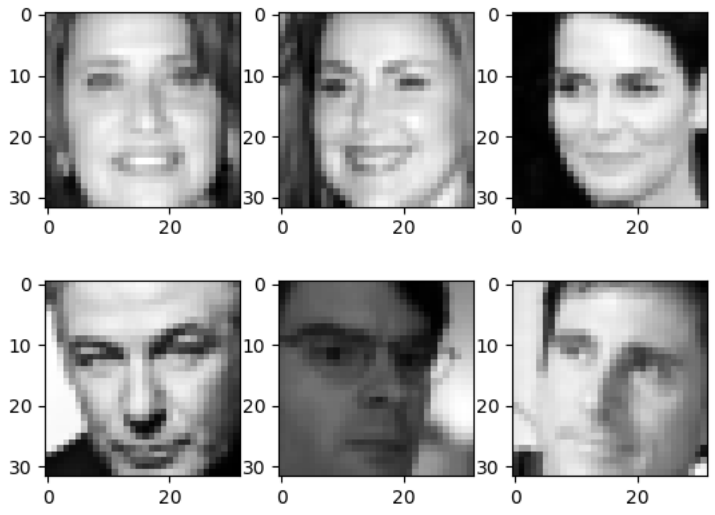
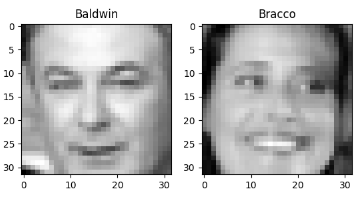
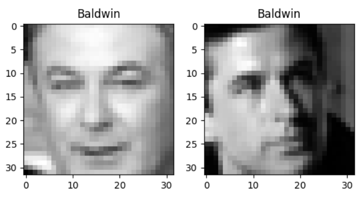


Figure 2: A random selection of images for each actor

Figure 3: Example of inaccurate bounding boxes



(a) Aligned faces



(b) Slightly unaligned faces

Figure 4: Examples of aligned and slightly unaligned faces

Part 2

Splitting data

Before performing classification, we need to split the data into non-overlapping training, validation and test sets. The splitting algorithm is implemented in the *split()* function in *part_2.py*. See below for details.

```python
def split(X, k_train=70, k_val=10, k_test=10):
    '''
    Split the feature data into training, validation and test sets for each actor.
    :param X: list, of which each element corresponding to a particualr actor
    :param k_train: int, the size of the training set
    :param k_val: int, the size of the validation set
    :param k_test: int, the size of the test set
    :return: X_train, X_val, X_test
    '''
    X_train, X_val, X_test = [], [], []
        for i in range(len(X)):
        # all possible indices of the images for the i-th actor
        ix = list(range(len(X[i])))
        ix_train = np.random.choice(ix, k_train, replace=False)
        X_train_i = np.zeros((len(ix_train), 1024))  # training images of the i-th actor
        for j in range(len(ix_train)):
            X_train_i[j] = X[i][ix_train[j]]
        X_train.append(X_train_i)

        # remove indices for training set
        ix_rest = [n for n in ix if n not in ix_train]
        ix_val = np.random.choice(ix_rest, k_val, replace=False)
        X_val_i = np.zeros((len(ix_val), 1024))  # validation images of the i-th actor
        for j in range(len(ix_val)):
            X_val_i[j] = X[i][ix_val[j]]
        X_val.append(X_val_i)

        # remove indices for validation set
        ix_rest = [n for n in ix_rest if n not in ix_val]
        ix_test = np.random.choice(ix_rest, k_test, replace=False)
        X_test_i = np.zeros((len(ix_test), 1024))  # testing images of the i-th actor
        for j in range(len(ix_test)):
            X_test_i[j] = X[i][ix_test[j]]
        X_test.append(X_test_i)
    return X_train, X_val, X_test
```

The algorithm takes in a list X, which contains 6 sub-lists of images of the six actors, the training size *k_train*, validation size *k_val* and the test size *k_test*. For each actor, the algorithm randomly generates *k_train* integers and selects the images with these integers as index from X to form the training set. Then the algorithm randomly generates two other different series of integers to form the validation set and the test set. The algorithm makes sure the indices are within bound by generating them from *list(range(len(X[i])))*, i.e. all possible image indices of a particular actor. It also ensures the training set, validation set and test set are non-overlapping by removing the indices selected before.

Part 3

Classification to distinguish Baldwin and Carell

To build a classifier using linear regression for binary face recognition, the cost function to minimize is the quadratic loss function, i.e. $J(\theta) = \sum_{i=1}^{m} \left(\theta^T x^{(i)} - y^{(i)}\right)^2$, where $\boldsymbol{\theta}$ is a vector of the weights for the features, $x^{(i)}$ is the i^{th} training example, $y^{(i)}$ is the label (1 if it's a photo of Baldwin; 0, otherwise).

When training the model, we run gradient descent to gradually reduce the value of the cost function on the training set to obtain the set of thetas that best approximate the distribution of the image data. For the algorithm to work, the learning rate alpha cannot be too large (greater than *1e-4*) as it will lead to overflow in this specific setting. A large learning rate may also make the algorithm miss the minimal point. Alpha cannot be too small either as it would take a very long time to converge. The value of EPS and max_iter should be carefully chosen as well. If EPS is too large or max_iter is small, the gradient cannot descend much, which makes the model underfit. If EPS is too small or max_iter is too large, it will take a very long time for the algorithm to converge and might lead to overfitting.

The algorithm encodes Baldwin as 1 and Carell as 0. When we use the model to do classification, the model assigns a value $\theta^T x$ to the input image $\boldsymbol{x}$. If it's greater than 0.5, the image is classified as Baldwin's; otherwise, it's classified as Carell's. The code that does so is shown below.

```python
def classify(X, theta):
    '''
    Classify images into Baldwin or Carell with linear regression model parameters theta.
    :param X: ndarray, input features
    :param theta: ndarray, model parameters
    :return: y_pred
    '''
    X = np.vstack((np.ones((1, X.shape[1])), X))
    y_pred = np.dot(theta.T, X)
    for i in range(y_pred.shape[0]):
        if y_pred[i] > 0.5:
            y_pred[i] = 1
        else:
            y_pred[i] = 0
    return y_pred
```

In this task, we train a model with a training set of 70 images per actor, and test it with a validation set of 10 images per actor. Fix the initial theta as a vector of 0.01s, EPS being *1e-6* and the maximum number of iterations being 5,000. If we set alpha to *1e-5*, the trained model gives a 99.3% accuracy on the training set and 90.0% accuracy on the validation set. The values of the cost function on the training and the validation sets are 2.43149009609 and 2.64127925282. If we set the value of alpha to *1e-6*, the values of the cost function on the training and the validation sets are 10.0291842905 and 2.42949918122 respectively. The accuracies on the training and the validation sets are 94.3% and 85.0%.

Note that the values of the cost function and the accuracy could change a bit if different random samples of the images are used in training/validation process. Also note that the input images are divided by 255.0.

Part 4

Plotting theta as an image

The images required by (a) are displayed in the first row of Figure 5. The images required by (b) are displayed in the second row of Figure 5 and in Figure 6.

Set the initial theta as a vector of 0.01s, the learning rate being *1e-5* and EPS being *1e-6*. The image in the upper left corner is produced by the theta obtained from Part 3. We can see that it looks more or less like a face. The image in the upper right corner is produced by the theta from a training size of 2 images per actor. The face is much more obvious in this image. However, the model suffers from overfitting as the training size is too small. The image in the lower left corner is produced by the theta from gradient descent with early stopping at 500 iterations. Again, the face is much more obvious in the image. Underfitting may have occured in this setting since the gradient descent process is stopped at only 500 iterations and the gradient hasn't descended near 0 yet. The last image is produced by the theta from gradient descent with *max_iter* being 50,000. The real process stops at about 30,000 iterations when it reaches the EPS boundary. The image looks a bit like a face but more noisy than the previous three. The model seems to be overfitted to the training data in this setting.

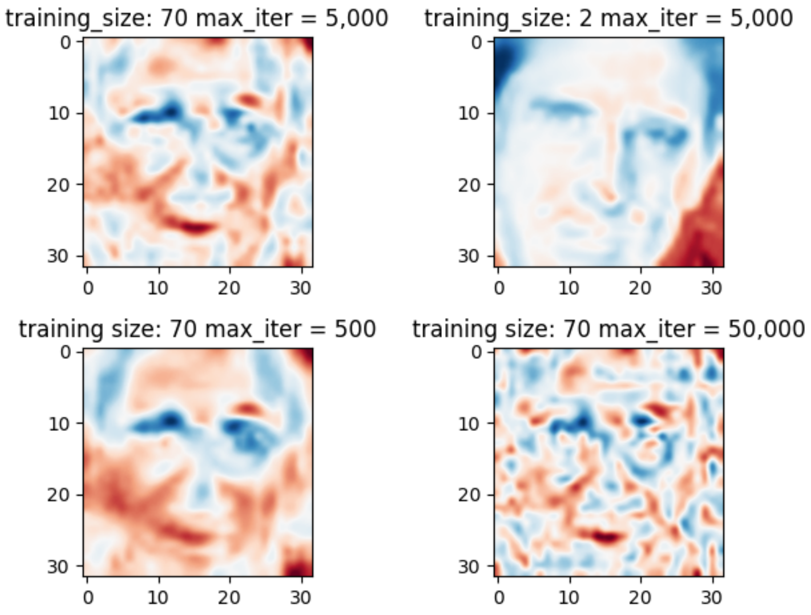


Figure 5: Images of thetas with different training sizes and max iterations

If we fix the learning rate as *1e-5*, EPS as *1e-6*, *max_iter* being 5000 and change the initial theta, different images are produced as shown in Figure 6. The left image is obtained from the initial theta being a series of 0.01s, while the right one is produced from randomly-initialized theta. We can see that the image on the right is very noisy. The model with this theta seems to be overfitted to the training data.

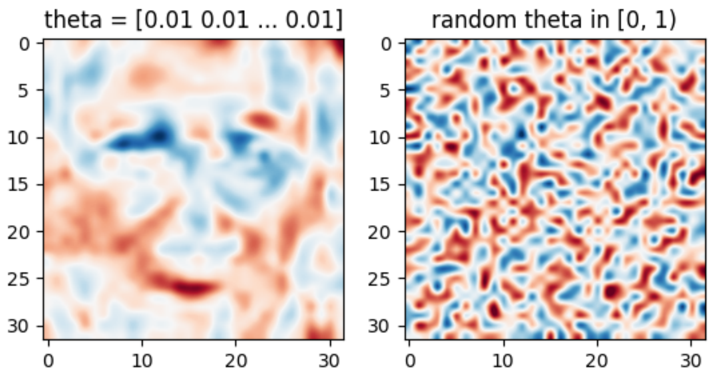


Figure 6: Images of thetas with different initial thetas

Part 5

Gender classification

In this part, we build another classifier using linear regression and gradient descent to classify the actors as male or female using the training set with the six actors Lorraine Bracco, Peri Gilpin, Angie Harmon, Alec Baldwin, Bill Hader and Steve Carell. The algorithm encodes make as 1 and female as 0. Similar to Part 3, the classifier assigns a value $\theta^T x$ to the input image $\boldsymbol{x}$. If the value is greater than 0.5, the actor in the image is classified as male; otherwise, the actor is classified as female.

When training the model, set the learning rate as *1e-6*, EPS as *1e-6* and *max_iter* as 30,000 to achieve a higher accuracy. Train the first model with a training size of 1 image per actor, and increase the number of images by 5 per actor every time. In total, 21 models are trained. Then validate and test each model with a validation set constructed by 10 images of every actor in the above list and a test set constructed by 10 images of every actor not in the list (i.e. Daniel Radcliffe, Gerard Butler, Michael Vartan, Kristin Chenoweth, Fran Drescher and America Ferrera). Use the same validation and test set for different training sets, so that the accracy is steady and can used for comparison. We also make sure in the validation and test set, there are equal number of images for every actor, so that if the model doesn't generalize well to one particular actor, the accuracy wouldn't be affected too much. The training, validaton and test accuracy with regard to the training size are shown in Figure 7. Specifically, With a training size of 100 images per actor, the accuracies on the training, validation and test set are 96.2%, 95% and 91.7%.

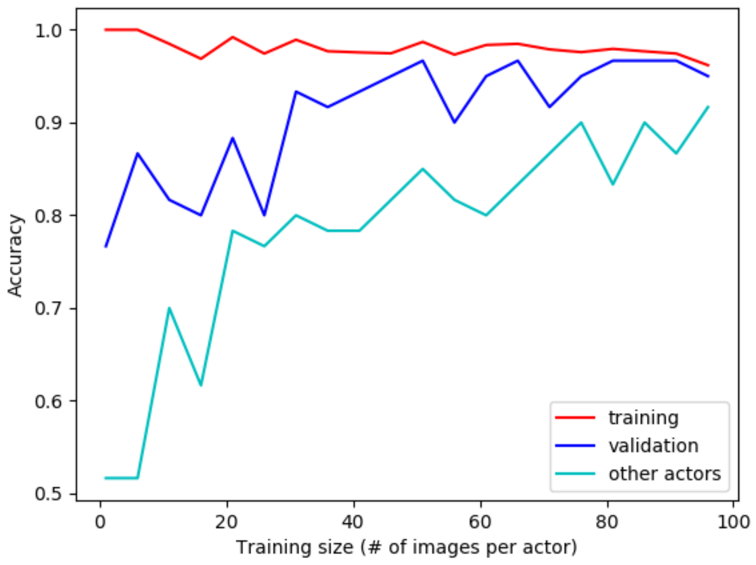


Figure 7: Images of thetas with different initial thetas

We can see that when the training size is small, the training accuracy is really high, while the validation and test accuracy are quite low. We can infer that overfitting occurs when the training size is small because the model doesn't generalize well to the validation and the test set. As the number of images in the training set grows larger, the training accuracy goes down a bit, while the validation and test accuracy increase gradually. Although the validation accuracy is high, roughly 95% when the training set grows large, the test accuracy is much lower. This shows that the model generalizes okay to images of the same set of actors, but not as well on images of other actors.

Part 6

Gradient computation

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

$J(\theta) = \sum_{i=1}^{m}\sum_{j=1}^{k}\left(\theta_j^T x^{(i)} - y_j^{(i)}\right)^2$ where θ_j^T is the *jth* row of the matrix θ^T, m is # of training examples

$\frac{\partial J}{\partial \theta_{pq}} = \sum_{i=1}^{m}\frac{\partial}{\partial \theta_{pq}}\left(\theta_q^T x^{(i)} - y_q^{(i)}\right)^2$ where θ_q^T is the *qth* row of the matrix θ^T

$\frac{\partial J}{\partial \theta_{pq}} = \sum_{i=1}^{m} 2\left(\theta_q^T x^{(i)} - y_q^{(i)}\right)\frac{\partial}{\partial \theta_{pq}}\left(\theta_q^T x^{(i)}\right)$

$\frac{\partial J}{\partial \theta_{pq}} = 2\sum_{i=1}^{m}\left(\theta_q^T x^{(i)} - y_q^{(i)}\right) x_p^{(i)}$

(b) Show that the derivative of $J(\theta)$ with respect to all the components of θ can be written in matrix form as $2X(\theta^T X - Y)^T$.

(As I was just reminded by piazza @180 `https://piazza.com/class/jbcszug53kt1g9?cid=180`, I would like to provide two ways to solve the problem referring to (2) and (1) in @180 respectively.)
(2) Define m as the number of training examples, n as the number of features and k as the number of labels

$$X = \begin{bmatrix} X_1^{(1)} & ... & X_1^{(m)} \\ ... & ... & ... \\ X_n^{(1)} & ... & X_n^{(m)} \end{bmatrix} \text{ is a } n * m \text{ matrix} \qquad Y = \begin{bmatrix} Y_1^{(1)} & ... & Y_1^{(m)} \\ ... & ... & ... \\ Y_k^{(1)} & ... & Y_k^{(m)} \end{bmatrix} \text{ is a } k * m \text{ matrix}$$

$$\theta^T = \begin{bmatrix} \theta_{11}^T & ... & \theta_{1n}^T \\ ... & ... & ... \\ \theta_{k1}^T & ... & \theta_{kn}^T \end{bmatrix} \text{ is a } k * n \text{ matrix, where } \theta_{qp}^T \text{ is } pth \text{ element on the } qth \text{ row of } \theta^T, \text{ i.e. } \theta_{pq}$$

$$\theta^T X = \begin{bmatrix} \sum_{j=1}^{n}\theta_{1j}^T X_j^{(1)} & ... & \sum_{j=1}^{n}\theta_{1j}^T X_j^{(m)} \\ ... & ... & ... \\ \sum_{j=1}^{n}\theta_{kj}^T X_j^{(1)} & ... & \sum_{j=1}^{n}\theta_{kj}^T X_j^{(m)} \end{bmatrix}$$

$$= \begin{bmatrix} \theta_1^T X^{(1)} & ... & \theta_1^T X^{(m)} \\ ... & ... & ... \\ \theta_k^T X^{(1)} & ... & \theta_k^T X^{(m)} \end{bmatrix} \text{ is a } k * m \text{ matrix, where } \theta_q^T \text{ is the } qth \text{ row of } \theta^T$$

$$\theta^T X - Y = \begin{bmatrix} \theta_1^T X^{(1)} - Y_1^{(1)} & ... & \theta_1^T X^{(m)} - Y_1^{(m)} \\ ... & ... & ... \\ \theta_k^T X^{(1)} - Y_k^{(1)} & ... & \theta_k^T X^{(m)} - Y_k^{(m)} \end{bmatrix} \text{ is a } k * m \text{ matrix}$$

$$\left(\theta^T X - Y\right)^T = \begin{bmatrix} \theta_1^T X^{(1)} - Y_1^{(1)} & ... & \theta_k^T X^{(1)} - Y_k^{(1)} \\ ... & ... & ... \\ \theta_1^T X^{(m)} - Y_1^{(m)} & ... & \theta_k^T X^{(m)} - Y_k^{(m)} \end{bmatrix} \text{ is a } k * m \text{ matrix}$$

$$2X\left(\theta^TX-Y\right)^T=\begin{bmatrix}2\sum_{i=1}^m X_1^{(i)}\left(\theta_1^TX^{(i)}-Y_1^{(i)}\right) & ... & 2\sum_{i=1}^m X_1^{(i)}\left(\theta_k^TX^{(i)}-Y_k^{(i)}\right)\\ ... & ... & ...\\ 2\sum_{i=1}^m X_n^{(i)}\left(\theta_1^TX^{(i)}-Y_1^{(i)}\right) & ... & 2\sum_{i=1}^m X_n^{(i)}\left(\theta_k^TX^{(i)}-Y_k^{(i)}\right)\end{bmatrix}\text{is a } k * m \text{ matrix}$$

The element on the pth row and the qth colomn of $2X\left(\theta^TX-Y\right)^T$ is $2\sum_{i=1}^m X_p^{(i)}\left(\theta_q^TX^{(i)}-Y_q^{(i)}\right)$, which is the same as what is obtained in (a).

(1) $\nabla J\left(\theta\right)=\begin{bmatrix}\frac{\partial J}{\partial\theta_{00}} & ... & \frac{\partial J}{\partial\theta_{0k}}\\ ... & ... & ...\\ \frac{\partial J}{\partial\theta_{n0}} & ... & \frac{\partial J}{\partial\theta_{nk}}\end{bmatrix}$

$$=\begin{bmatrix}2\sum_{i=1}^m\left(\theta_0^Tx^{(i)}-y_0^{(i)}\right)x_0^{(i)} & ... & 2\sum_{i=1}^m\left(\theta_k^Tx^{(i)}-y_k^{(i)}\right)x_0^{(i)}\\ ... & ... & ...\\ 2\sum_{i=1}^m\left(\theta_0^Tx^{(i)}-y_0^{(i)}\right)x_n^{(i)} & ... & 2\sum_{i=1}^m\left(\theta_k^Tx^{(i)}-y_k^{(i)}\right)x_n^{(i)}\end{bmatrix}$$

a $n * k$ matrix, where where θ_q^T is the qth row of θ^T

$$=2\sum_{i=1}^m\begin{bmatrix}x_0^{(i)}\\ ...\\ x_n^{(i)}\end{bmatrix}\begin{bmatrix}\theta_0^Tx^{(i)}-y_0^{(i)} & ... & \theta_k^Tx^{(i)}-y_k^{(i)}\end{bmatrix}$$

$$=2\begin{bmatrix}x_0^{(1)} & ... & x_0^{(m)}\\ ... & ... & ...\\ x_n^{(1)} & ... & x_n^{(m)}\end{bmatrix}\begin{bmatrix}\theta_0^Tx^{(1)}-y_0^{(1)} & ... & \theta_k^Tx^{(1)}-y_k^{(1)}\\ ... & ... & ...\\ \theta_0^Tx^{(m)}-y_0^{(m)} & ... & \theta_k^Tx^{(m)}-y_k^{(m)}\end{bmatrix}$$

$$=2\begin{bmatrix}x_0^{(1)} & ... & x_0^{(m)}\\ ... & ... & ...\\ x_n^{(1)} & ... & x_n^{(m)}\end{bmatrix}\left(\begin{bmatrix}\theta_0^Tx^{(1)} & ... & \theta_k^Tx^{(1)}\\ ... & ... & ...\\ \theta_0^Tx^{(m)} & ... & \theta_k^Tx^{(m)}\end{bmatrix}-\begin{bmatrix}y_0^{(1)} & ... & y_k^{(1)}\\ ... & ... & ...\\ y_0^{(m)} & ... & y_k^{(m)}\end{bmatrix}\right)$$

$$=2X\left(\left(\theta^TX\right)^T-Y^T\right)$$

$$=2X(\theta^TX-Y)^T$$

(c) The cost function and the vectorized gradient function are implemennted as follows.

```python
def cost(X, y, theta):
    '''
    X: n*m ndarray
    y: k*m ndarray
    theta: n*k ndarray
    '''
    X = np.vstack( (np.ones((1, X.shape[1])), X))
    return np.sum((np.dot(theta.T, X) - y) ** 2)

def gradient(X, y, theta):
    '''
    X: n*m ndarray
    y: k*m ndarray
    theta: n*k ndarray
    '''
    X = np.vstack( (np.ones((1, X.shape[1])), X))
    inner_product = np.dot(theta.T, X) - y
    return 2 * np.dot(X, inner_product.T)
```

Note that the cost function is implemented in vectorized form to reduce computation overhead.

(d) The code to compute the gradient components using finite differences is as follows.

```
def finite_difference(X, y, theta, p, q, h=1e-8):
    '''
    p, q: index of the element to approximate the gradient
    '''
    theta0 = np.copy(theta)
    theta[p,q] = theta[p,q] + h
    return (cost(X, y, theta) - cost(X, y, theta0))/h
```

Randomly initialize the values of theta to floats in [0,1) and randomly select 5 elements of theta to compute the gradient using finite difference approximation as above. Then compare the approximations with the results computed by the vetorized gradient function, i.e. *gradient()*. This process is implemented as below.

```
# Prepare x y
X_train, X_val, X_test = part_2.split(X, k_train=70, k_val=10, k_test=10)
x_train, x_val, y_train, y_val = part_7.prepare(X_train, X_val)
# Randomly initialize theta with values in [0,1)
theta = np.random.rand(1025, 6)
print("h=1e-6/abs difference", "h=1e-7/abs difference", "gradient")
# Randomly select 5 coordinates
for i in range(5):
    p = np.random.randint(0, 1025)
    q = np.random.randint(0, 6)
    g1_6 = part_6.finite_difference(x_train.T, y_train.T, theta, p, q, 1e-6)
    g1_7 = part_6.finite_difference(x_train.T, y_train.T, theta, p, q, 1e-7)
    g2 = part_6.gradient(x_train.T, y_train.T, theta)
    print(g1_6, abs(g1_6 - g2[p, q]), g1_7, abs(g1_7 - g2[p, q]), g2[p, q])
```

Experiement two different values of *h*, *1e-6*, *1e-7*. Compute the absolute difference between them and the output of the gradient function. As shown in Table 1, the results are in agreement up to 8 significant digits. In general, *h* being *1e-6* and *1e-6* are both good to be used in finite difference approximation.

h=1e-6 / abs difference	h=1e-7 / abs difference	Gradient
77582.1208954 / 0.0306735360209	77582.0016861 / 0.0885357535299	77582.0902218
132828.176022 / 0.0110555332212	132827.758789 / 0.406176980207	132828.164966
95564.2163754 / 0.00484163628425	95564.1269684 / 0.0845653308788	95564.2115337
154136.031866 / 0.0122068721685	154135.82325 / 0.196409384545	154136.019659
127415.895462 / 0.0220973256946	127415.657043 / 0.216321253407	127415.873365

Table 1: Comparison of finite difference approximation and vectorized gradient function

Part 7

Face recognition

In the task, We train the third classifier using linear regression and gradient descent on the six actors (Lorraine Bracco, Peri Gilpin, Angie Harmon, Alec Baldwin, Bill Hader and Steve Carell) to perform face recognition. The algorithm uses one hot encoding to label the images. For example, images of *Lorraine Bracco* are labeled as [1,0,0,0,0,0], and images of *Peri Gilpin* are labelled as [0,1,0,0,0,0]. We use a training set of 70 images per actor to train the model and a validation set of 10 images per actor to validate the model. During training, the learning rate is set to *1e-6*, the EPS value is set to *1e-6*, the max_iter is set to 30,000 and initial theta is set to all 0.01s. This parameter setting gives a relatively good accuracy and doesn't take very long time to converge. When we use the model to do classification, the model assigns a vector value $\theta^T x$ to the input image $\boldsymbol{x}$ and takes the index with the maximum value in the vector as the predicted label. Details are shown in the following piece of code.

```
def classify(x, theta):
    x = np.vstack( (np.ones((1, x.shape[1])), x))
    y_pred = np.argmax(np.dot(theta.T, x), axis=0)
    return y_pred
```

The classifier has a 96.2% accuracy on the training set and a 76.7% accuracy on the validation set.

Part 8

Plotting theta as an image

Take the theta obtained in Part 7 and visualize the each row of it as an image, we get the following figure.

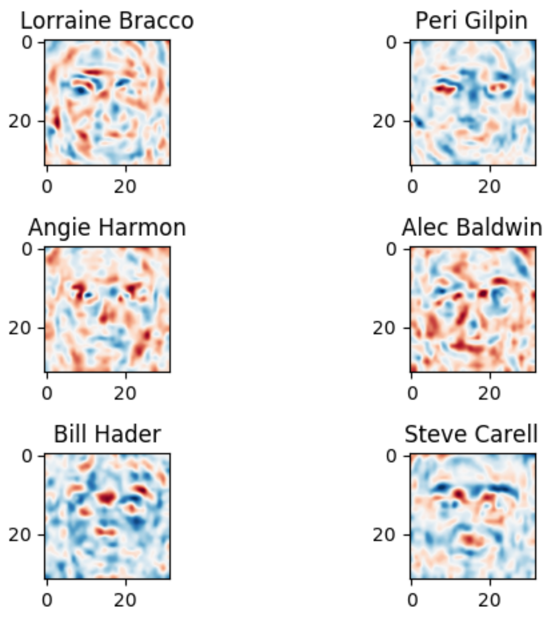


Figure 8: Images of rows of theta