ECE521 Assignment 1

Feb 2nd, 2018

Team Members:

Yilin Chen 1000311281

Wenyu Mao 1000822292

Jiaying Wang 1000337502

Table of Contents

1. **Contributions**……..1
2. **Euclidean Distance Function**……1
3. **Making Predictions for Regression**……...……...2
4. **Making Predictions for Classifications**……7

1. Contributions

Yilin Chen: 33.3%

Wenyu Mao: 33.3%

Jiaying Wang: 33.3%

2. Euclidean distance function

By observing the equation and result matrix for the Euclidean distance provided in the assignment handout (Figure 2.1), we found that for two matrices, X (N1 * D) and Z(N2 * D), the Euclidean distance between them is an N1 * N2 pairwise squared matrix.

$$\begin{bmatrix} \|\mathbf{x}^{(1)} - \mathbf{z}^{(1)}\|_2^2 & \dots & \|\mathbf{x}^{(1)} - \mathbf{z}^{(N_2)}\|_2^2 \\ \vdots & \ddots & \vdots \\ \|\mathbf{x}^{(N_1)} - \mathbf{z}^{(1)}\|_2^2 & \dots & \|\mathbf{x}^{(N_1)} - \mathbf{z}^{(N_2)}\|_2^2 \end{bmatrix}$$

Figure 2.1 Euclidean distance

```
def euclid_distance(X, Y):
    X = (tf.expand_dims(X, 1))#shape(4,1,2)
    Y = (tf.expand_dims(Y, 0))#shape(1,3,2)
    #shape(4, 3, 2) due to broadcasting
    square_diff = tf.squared_difference(X, Y)
    return tf.reduce_sum(square_diff, 2)
```

Figure 2.2 Euclidean Distance Function

The solution is shown in Figure 2.2. `tf.squared_difference` results in a pairwise matrix between two matrices. It follows TensorFlow broadcasting, which will automatically match the dimensions for two matrices. In order to use this functionality, we first expand the dimensions for two matrices. Then reduce the extra dimension and return the result.

3. Making Predictions for Regression

3.1 Choosing the nearest neighbours

```
def responsibility(distance_matrix, k):
    # We need to index the closest values
    # We need a matrix that each row represents 80 training data correspond to 1 test point, in total, we have 10 rows of such data)
    d_trans = tf.transpose(distance_matrix)
    # Now we have a matrix has all the distance as positive number
    # tf.nn.top_k gets the kth largest
    # Since we need to get the nearest
    d_nearest = tf.negative(d_trans)  # the largest negative is the nearest
    distance, indices = tf.nn.top_k(d_nearest, k)  # the kth nearest
    # get number of training Data
    td_num = tf.shape(d_nearest)[1]
    d_nearest_range = tf.range(td_num)
    # to compare with the indices, reshape the d_nearest_range in order to broadcasting later
    index_compare = tf.reshape(d_nearest_range, [1, 1, -1])  # 1 1 4
    # in order to broadcast, we should change indices's last dimension to 1
    indices = tf.expand_dims(indices, 2)  # 3 2 1
    # find the indices for the nearest distance
    nearest_index = tf.reduce_sum(tf.to_float(tf.equal(index_compare, indices)), 1)
    tf.cast(nearest_index, tf.float32)
    result = nearest_index / tf.to_float(k)
    tf.cast(result, tf.float32)
    return result
```

Figure 3.1 Responsibility Function

`tf.nn.top_k` function is used to find values and indices of the k largest entries for the last dimension. Since the order of the input arguments of the euclid_distance function matters and we use X = trainData and Y = testData, the resulted distance_matrix has to be transposed first to obtain the correct matrix shape (N2 * N1, N1 is the number of train data, N2 is the number of test data) before using `tf.nn.top_k` function. Then the elements of the distance matrix are changed to negative since `tf.nn.top_k` function find the largest entries. The nearest data should have the largest values after negation. After the data processing, `tf.nn.top_k` function is used to obtain the indices of the k nearest data in the train dataset for each test data. The resulted indices matrix is then expanded and compared with a matrix of all the possible training data indices in order to get a vector like [R1, R2, … Rn] where Ri = 1 if the i-th training data belongs the the neighbor of the closest k samples and Ri = 0 otherwise. The above vector is then divided by k following the formula in the handout.

3.2 Prediction

$$\hat{\mathbf{y}}(\mathbf{x}^*) = \mathbf{Y}^T\mathbf{r}^*, \text{ where } \mathbf{r}^* = [r_1, \ldots, r_N], r_n = \begin{cases} \frac{1}{k}, & \mathbf{x}^{(n)} \in \mathcal{N}_{\mathbf{x}^*}^k \\ 0, & \text{otherwise.} \end{cases}$$

The formula above is used in the prediction function to calculate the predicted results.

```
def prediction(train, test, train_target, k):
    # get reposibility
    distance_matrix = euclid_distance(train, test)
    r_star = tf.transpose(tf.cast(responsibility(distance_matrix, k), tf.float32))
    prediction_y = tf.matmul(tf.transpose(train_target), r_star)
    return tf.transpose(prediction_y)
```

Figure 3.2 Prediction function

3.3 MSE Loss

$$\mathcal{L} = \frac{1}{2N}\sum_{n=1}^{N}\|\hat{\mathbf{y}}(\mathbf{x}^{(n)}) - \mathbf{y}^{(n)}\|_2^2$$

The formula above is used to calculate the MSE loss. Figure 3.3 shows the implementation function.

```
def MSE(train,test,train_target,test_target,k):
    #get prediction
    predicted_result = prediction(train,test,train_target,k)
    #get squared sum and cast the type
    squared_sum=tf.reduce_sum(tf.pow(tf.to_float(tf.subtract(predicted_result, test_target)),2))
    num_test = tf.to_float(tf.shape(test)[0])
    result = squared_sum/(num_test*2)
    return result
```

Figure 3.3 MSE function

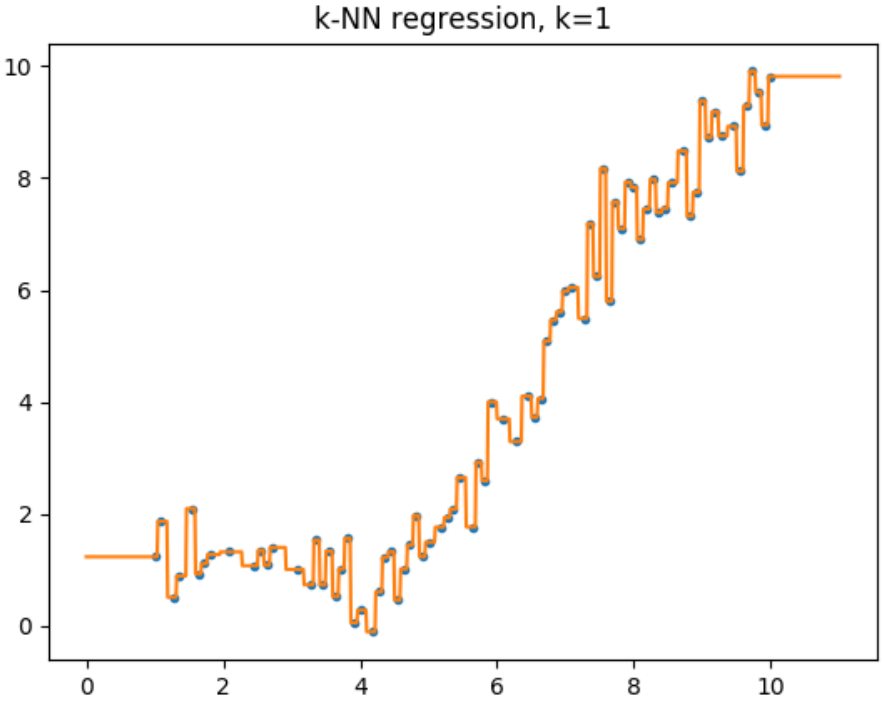
Tables below shows the results. The lowest validation MSE losses is found to occur with k = 1.

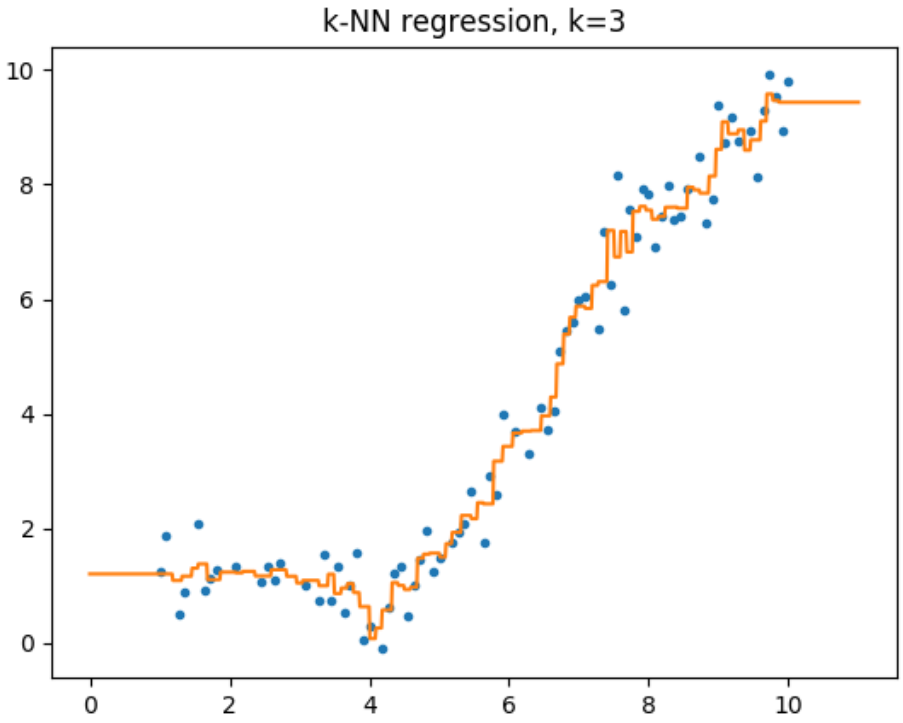
Note: The MSE losses vary when different data type (float32 and float64) are used. But the best k is k = 1 in both cases.

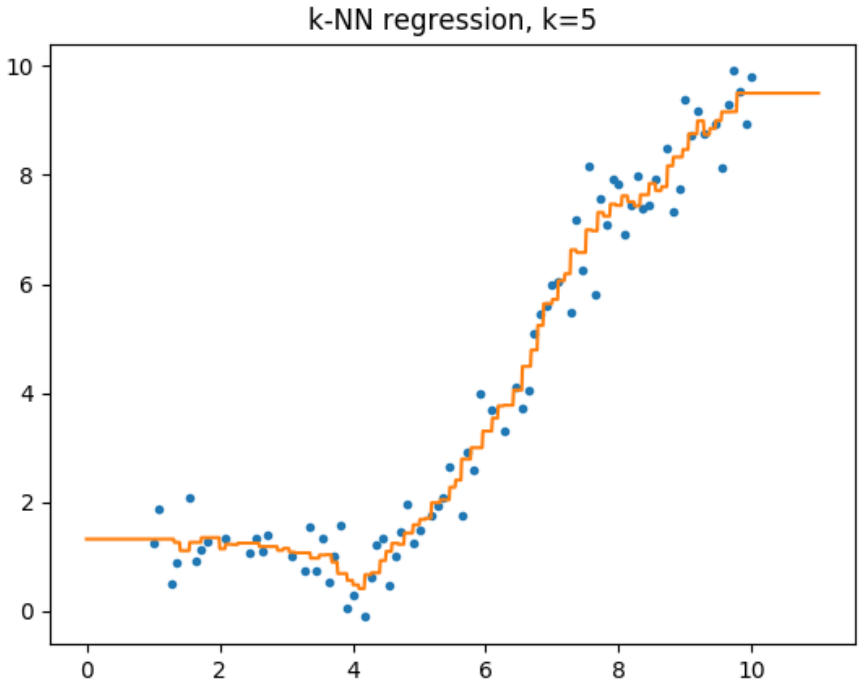
Table 1: MSE Losses (using float32)			
	Training MSE Loss	Validation MSE Loss	Testing MSE Loss
k=1	0.000000	**0.271550**	0.311004
k=3	0.105241	0.326278	0.145092
k=5	0.118541	0.310439	0.178327
k=50	1.248009	1.228702	0.707935

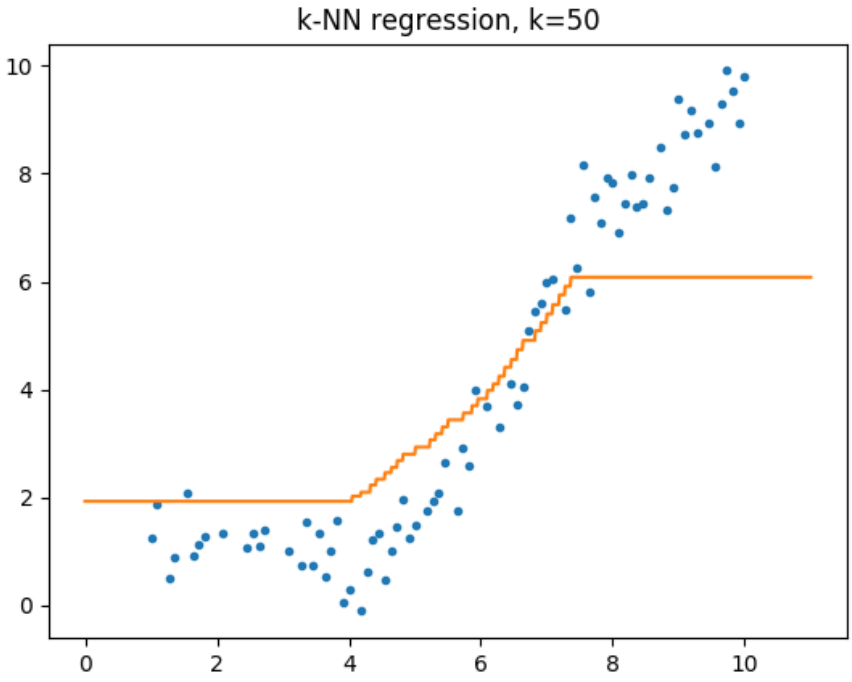
Table 2: MSE Losses (using float64)			
	Training MSE Loss	Validation MSE Loss	Testing MSE Loss
k=1	0.000000	**0.271550**	0.139432
k=3	0.106465	0.324376	0.158797
k=5	0.121068	0.316699	0.185096
k=50	1.245985	1.228702	0.702632

The following figures show the prediction results using different k. At k = 1, the predictions follow the true values exactly with sharp jumps across data points. This model is too complex with too many parameters and the trend does not reflect the reality of the data. So at k=1, the model is overfitting. When k gets larger, the deviation from true values starts to appear but the prediction still follows the general trend of the true values. When k = 50, the prediction tends to approach the average of a larger pool of data within the training dataset where a lot of irrelevant data are used. Thus, more errors are incurred with large k and the model is underfitting. The best k should be within the intermediate range.


k-NN regression, k=1
0
2
4
6
8
10
0
2
4
6
8
10


k-NN regression, k=3
0
2
4
6
8
10
0
2
4
6
8
10


k-NN regression, k=5
0
2
4
6
8
10
0
2
4
6
8
10


k-NN regression, k=50
0
2
4
6
8
10
0
2
4
6
8
10

4. Making Predictions for Classifications

4.1 Prediction

In order to make predictions for name and gender, the data are splitted into three portions, train data, valid data and test data. We use train data to train the model first, and then find the nearest k train data and corresponding train target to valid data. Lastly, majority voting is used to find the most frequent result in these k train target (Figure 4.1).

```
def predict_class(row_test,train,train_target,k):

    distance_matrix=euclid_distance(row_test,train) #rowtest=1x1024 train=747*1024
    d_nearest = tf.negative(distance_matrix) #the largest negative is the nearest
    distance, indices = tf.nn.top_k(d_nearest, k) #the kth nearest
    #pick corresponding train target according to indices
    t_target=tf.gather_nd(tf.reshape(train_target,[-1]),tf.transpose(indices))
    #use majority voting to find result
    majority,idx,count = tf.unique_with_counts(tf.reshape(t_target,shape=[-1]))
    max_count,max_idx = tf.nn.top_k(count,k=1)
    #the majority train target corresponding to test data
    majority_result = tf.gather(majority,max_idx)

    return majority_result, tf.reshape(indices,[-1])
```

Figure 4.1 Prediction Function

4.2 Find Accuracy

K is chosen to be 1, 5, 10, 25, 50,100 and 200 . Afterwards, predicted results for valid data are compared with valid targets to determine the best k, and then with that k, test data is used to get test accuracy. To manipulate the test data easily, we chose to pass single test data (one image) to our predict function each time and iterate all test images outside the prediction function (Figure 4.2).

```
def face_recognition(task):
    trainData, validData, testData, trainTarget, validTarget, testTarget \
            = data_segmentation("data.npy", "target.npy", task)
    print(np.shape(trainTarget.reshape(1,-1)))
    print(np.shape(validData[1,:].reshape(1,-1)))
    #return
    train = tf.placeholder(tf.float32, [None, None], name="train")
    train_target = tf.placeholder(tf.float32, [None, None], name="train_target")
    valid = tf.placeholder(tf.float32, [None, None], name="valid")
    valid_target = tf.placeholder(tf.float32, [None, None], name="valid_target")
    test = tf.placeholder(tf.float32, [None, None], name="test")
    test_target = tf.placeholder(tf.float32, [None, None], name="test_target")
    K = tf.placeholder("int32", name="k")

    row_prediction = predict_class(test,train,train_target,K)

    sess = tf.InteractiveSession()
```

```
    K_values = [1,5,10, 25, 50, 100, 200]
    k_errors = []
    # run validation dataset
    for k in K_values:
        results = []
        #validData_result = predict_class(valid,train,train_target,k)
        for row_index in range(np.shape(validData)[0]):
            row_result, indices = sess.run(row_prediction, \
            feed_dict={ test:validData[row_index,:].reshape(1,-1),\
                train:trainData, train_target: trainTarget.reshape(1,-1), K:k})

            if row_result-validTarget[row_index] != 0:
                results.append(1)
            else:
                results.append(0)
```

```
        error = np.sum(results)
        k_errors.append(error)

        print("For k = %d, accuracy of valid data is %f \n"%\
              (k, 1-error/float(np.shape(validData)[0])))

    k_best = K_values[np.argmin(k_errors)]
    print("best k is k=%d\n"%(k_best))

    # run test dataset with k_best
    results = []
    for row_index in range(np.shape(testData)[0]):
            row_result, indices = sess.run(row_prediction,\
            feed_dict={ test:testData[row_index,:].reshape(1,-1),\
            train:trainData, train_target: trainTarget.reshape(1,-1), K:k_best})

            if row_result-testTarget[row_index] != 0:
                results.append(1)
            else:
                results.append(0)
            #for i in range(np.shape(indices)[0]):
             #   print("i = %d\n"%(indices[i]))

    error = np.sum(results)

    print("Using k=%d, accuracy of test data is %f\n"%(k_best, \
                          1-error/float(np.shape(testData)[0])))
```

Figure 4.2 Load Data and find corresponding accuracy

4.3 Conclusion

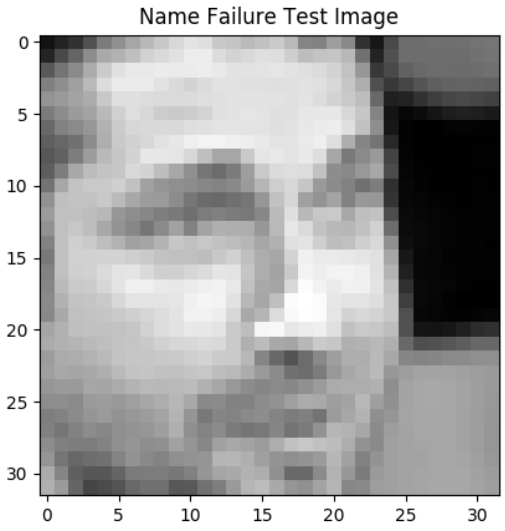
As shown in the table below, it is clear that k=1 is the best choice for both name prediction and gender prediction. With k=1, the accuracy of test data for name is 0.709677 and for gender is 0.924731. Generally speaking, the accuracy of predicted results for gender is much more higher than name. Because there are more classes for name test. And the accuracy decreased as increasing k, except for k=25, which has better accuracy than k=10 for both name and gender predictions.

Face Recognition Accuracy (Validation Data)		
	Name Recognition	Gender Recognition
k=1	**0.663043**	**0.913043**
k=5	0.608696	0.913043
k=10	0.576087	0.891304
k=25	0.597826	0.902174
k=50	0.576087	0.891304
k=100	0.478261	0.858696
k=200	0.315217	0.782609
Face Recognition Accuracy (Test Data) Using k=1 since it provides the highest accuracy		
k=1	0.709677	0.924731

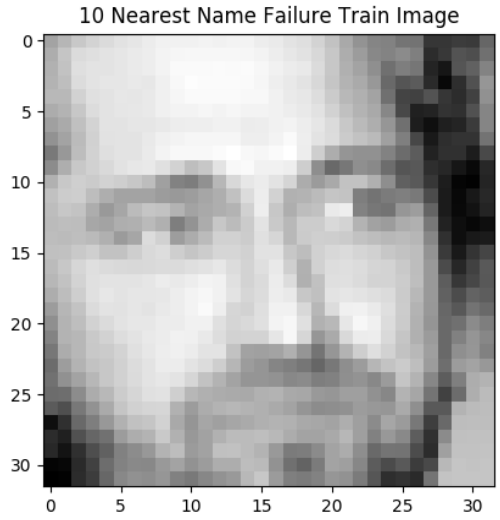
Our algorithm picks the first incorrect prediction for both name and gender and it turns out to be same test data. As we can see in the 10 images below, most are female but the test image is male. In this case, the majority voting is wrong.

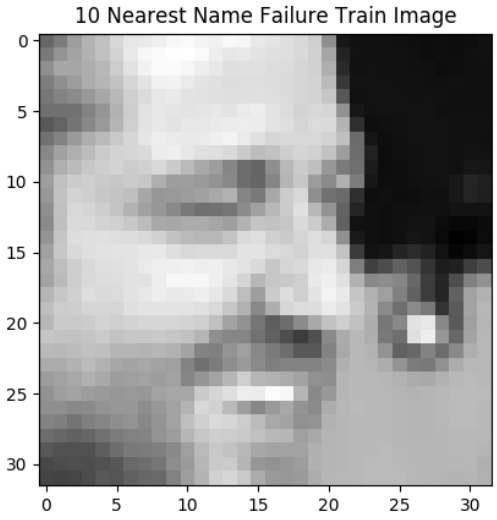
In conclusion, as the number of our train data are not large, k=1 has the best performance for both name and gender predictions.

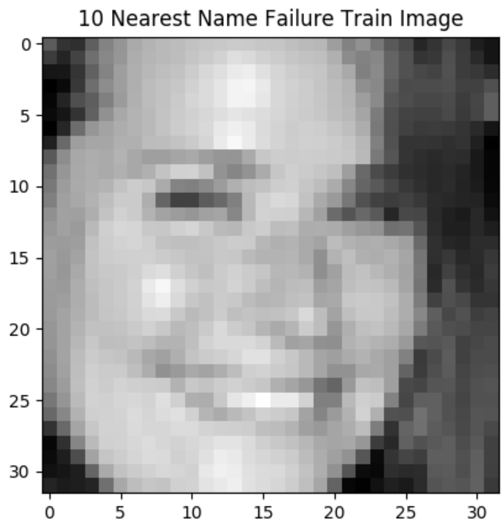
Test image for both name and gender test:

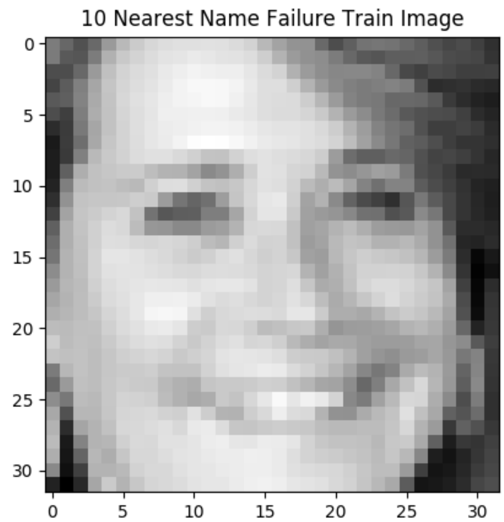


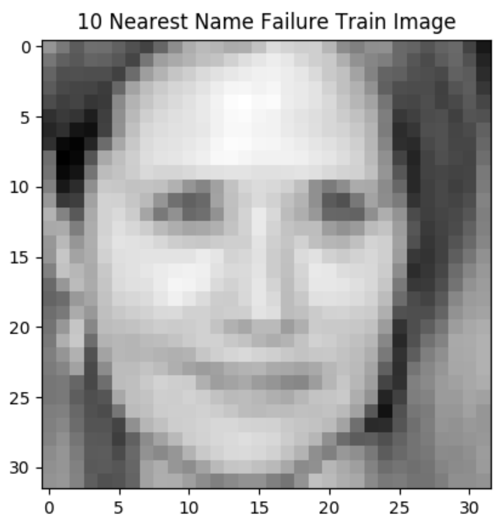
Ten nearest train images for both name and gender test:

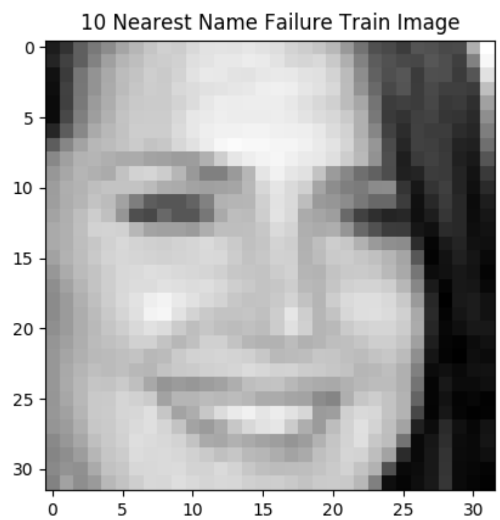


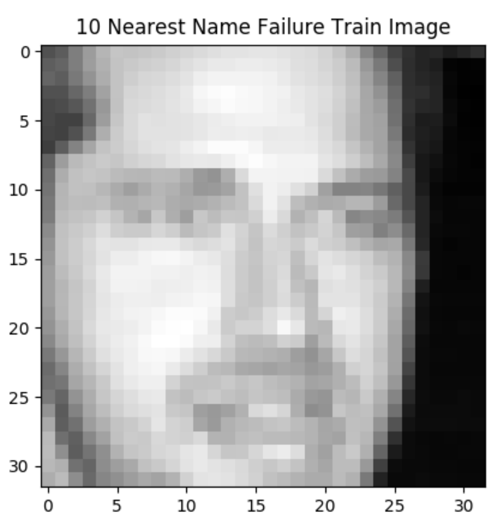


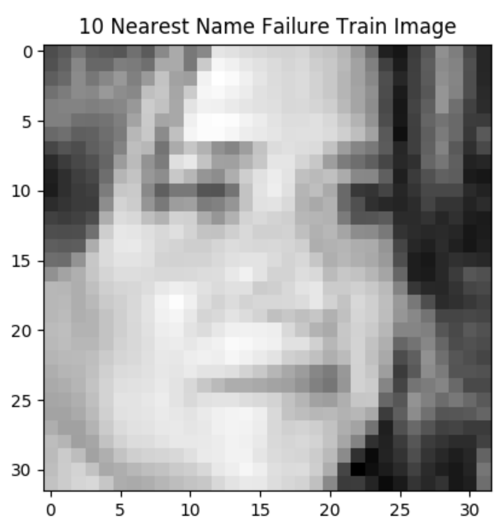

10 Nearest Name Failure Train Image
0 5 10 15 20 25 30
0 5 10 15 20 25 30


10 Nearest Name Failure Train Image
0 5 10 15 20 25 30
0 5 10 15 20 25 30

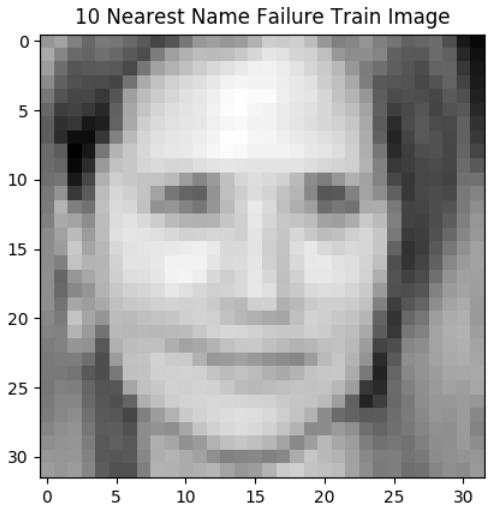

10 Nearest Name Failure Train Image
0 5 10 15 20 25 30
0 5 10 15 20 25 30


10 Nearest Name Failure Train Image
0 5 10 15 20 25 30
0 5 10 15 20 25 30


10 Nearest Name Failure Train Image
0 5 10 15 20 25 30
0 5 10 15 20 25 30


10 Nearest Name Failure Train Image
0 5 10 15 20 25 30
0 5 10 15 20 25 30


10 Nearest Name Failure Train Image
0
5
10
15
20
25
30
0
5
10
15
20
25
30


10 Nearest Name Failure Train Image
0
5
10
15
20
25
30
0
5
10
15
20
25
30