Methods_Comparison_via_Handwritten_Recognition_Dataset

May 20, 2019

1 Content

2 Introduction

In this paper, 2 machine learning methods (Logistic Regression and Random Forest) are implemented to classify the 10 categories (0-9) from MNIST dataset. For each method, we are not only use it to predict, but also we will try to predict after dimension reduction.

Finally, the results will be detailed compared. We will compare within the method, at the same time, we will compare the result across methods.

3 Model

There is very detailed analysis and comparison within each methods.

For Logistic Regression(LR), we firstly code a LR class from scratch and do prediction. Then, we plan to do dimension deduction before classification and see what happens on this condition. Therefore, we seperately retain 90% and 95% eigenvalues at PCA part, and generate new dataset with related eigenvectors.

For Random Forest(RF), we repeat the similar process. But this time, we call scikit-learn package to train random forest model.

Additionally, the MNIST dataset contains 60000 images in training dataset. This kind of volume of dataset is a big burden for training LR model and RF model. To save time and protect our laptop, we keep 25000 training images for LR model and 20000 training images for RF models.

This try can improve time-consuming problem, but will reduce the overall accuracy. Hence, we trust that the full training dataset will bring us the best prediction.

4 Result

4.1 Logistic Regression

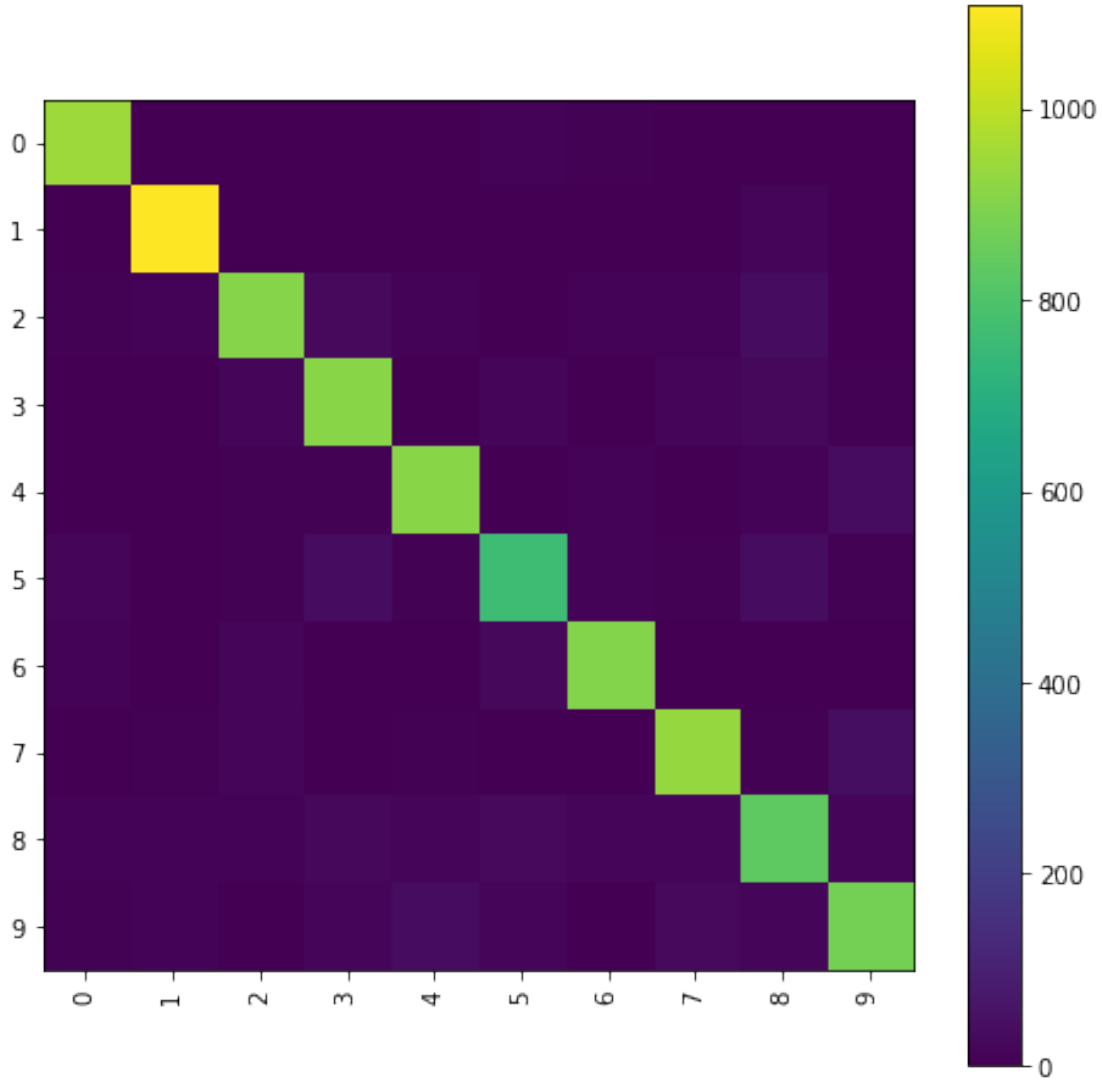
Here, we set learning rate as 0.00002, iteration time as 200001 and choose stochastic gradient descent to do iterataion.
For all categories, the cost is bigger than 300 for every iteration. Even if we have the high cost, we get good prediction result, the overall accuracy is 91.03%. The micro average accuracy is 0.910; the macro average is 0.909; the weighted average is 0.910.

```
In [0]: from sklearn.metrics import classification_report, confusion_matrix
        print(confusion_matrix(y_test, y_pred))
        print(classification_report(y_test, y_pred,
                                    target_names=list(label_dict.values()),digits=3))
        plt.figure(figsize=(8,8))
        cnf_matrix = confusion_matrix(y_test, y_pred)
        classes = list(label_dict.values())
        plt.imshow(cnf_matrix, interpolation='nearest')
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        _ = plt.xticks(tick_marks, classes, rotation=90)
        _ = plt.yticks(tick_marks, classes)
```

```
[[ 948    0    4    3    0   11    8    2    2    2]
 [   0 1109    3    2    1    1    4    1   13    1]
 [   7   12  909   28    9    4   10   12   38    3]
 [   4    3   20  913    4   20    3   13   22    8]
 [   1    1    5    6  910    0   10    4    9   36]
 [  13    4    7   36    8  765   11    6   36    6]
 [  10    3   13    0    4   22  902    1    3    0]
 [   4    8   20    4    7    4    1  932    6   42]
 [   9   11   10   23   14   28   15   15  836   13]
 [   8    9    1   18   37   14    0   30   13  879]]
              precision    recall  f1-score   support

           0      0.944     0.967     0.956       980
           1      0.956     0.977     0.966      1135
           2      0.916     0.881     0.898      1032
           3      0.884     0.904     0.894      1010
           4      0.915     0.927     0.921       982
           5      0.880     0.858     0.869       892
           6      0.936     0.942     0.939       958
           7      0.917     0.907     0.912      1028
           8      0.855     0.858     0.857       974
           9      0.888     0.871     0.879      1009
```

micro avg	0.910	0.910	0.910	10000
macro avg	0.909	0.909	0.909	10000
weighted avg	0.910	0.910	0.910	10000



4.2 PCA + Logistic Regression, Retain 90% Eigenvalues

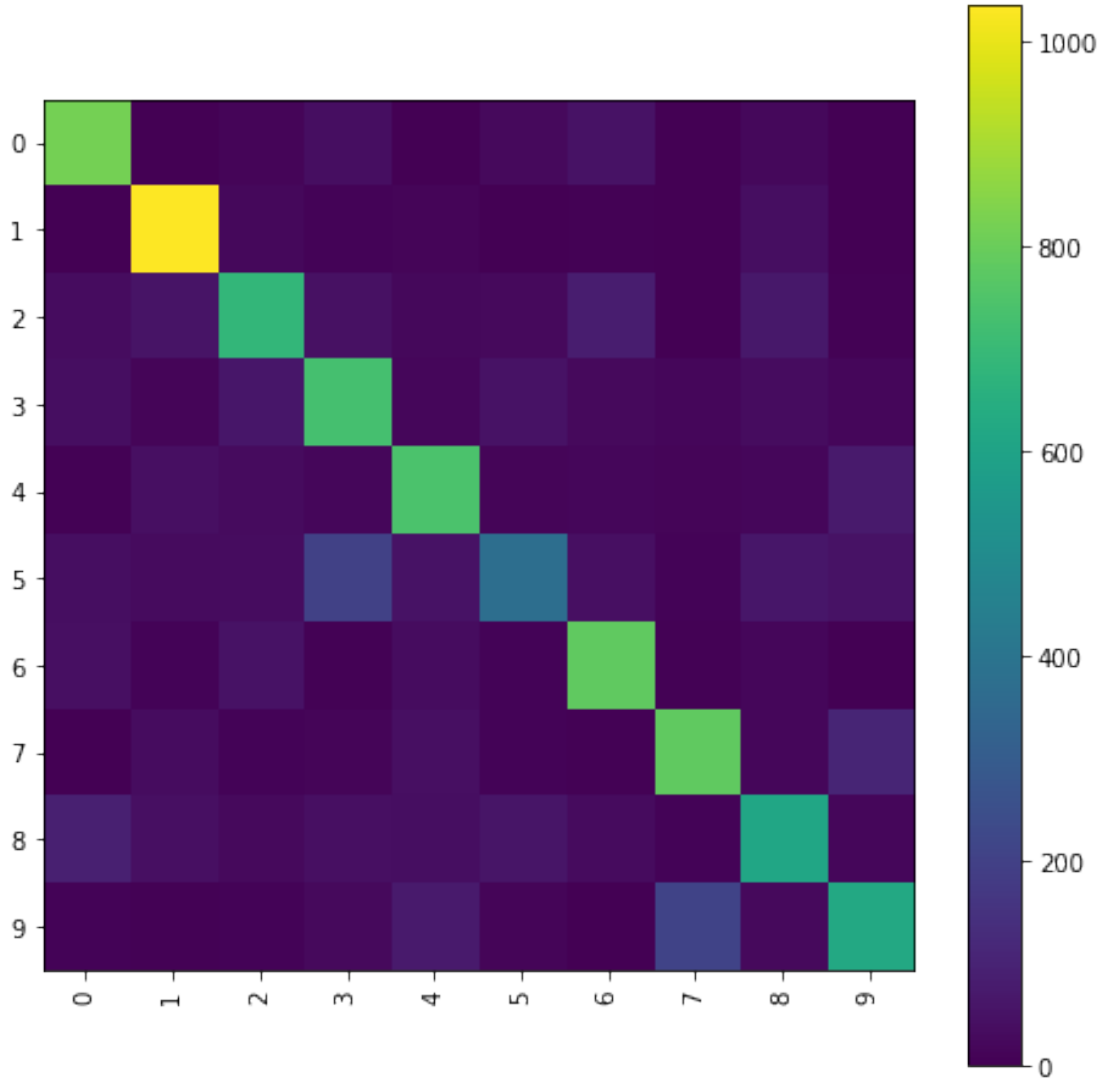
As we can see, we keep 86 eigenvectors, set learning rate as 0.00002, iteration time as 200001 and choose stochastic gradient descent to do iterataion. For all categories, the cost is bigger than 2000 for every iteration. The overall accuracy is 71.97%. The micro average accuracy is 0.720; the macro average is 0.713; the weighted average is 0.716.

```
In [0]: print(confusion_matrix(y_test, y_pred))
        print(classification_report(y_test, y_pred,
                                    target_names=list(label_dict.values()),digits=3))
        plt.figure(figsize=(8,8))
        cnf_matrix = confusion_matrix(y_test, y_pred)
        classes = list(label_dict.values())
        plt.imshow(cnf_matrix, interpolation='nearest')
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        _ = plt.xticks(tick_marks, classes, rotation=90)
        _ = plt.yticks(tick_marks, classes)
```

```
[[ 819    0   14   37    4   28   51    2   22    3]
 [   1 1037   21   11   14    4    7    1   38    1]
 [  34   55  685   45   24   28   84    3   67    7]
 [  40   13   61  731   17   49   28   17   35   19]
 [   6   41   31   20  745   15   19   13   19   73]
 [  38   30   34  199   52  375   44    9   61   50]
 [  42   10   50    5   33    9  784    5   19    1]
 [   3   34   10   14   41    9    8  784   17  108]
 [  92   43   27   43   37   58   31   11  613   19]
 [  10    8    9   27   75   16    2  210   28  624]]
              precision    recall  f1-score   support

           0      0.755     0.836     0.793       980
           1      0.816     0.914     0.862      1135
           2      0.727     0.664     0.694      1032
           3      0.646     0.724     0.683      1010
           4      0.715     0.759     0.736       982
           5      0.635     0.420     0.506       892
           6      0.741     0.818     0.778       958
           7      0.743     0.763     0.753      1028
           8      0.667     0.629     0.648       974
           9      0.690     0.618     0.652      1009

   micro avg      0.720     0.720     0.720     10000
   macro avg      0.713     0.714     0.710     10000
weighted avg      0.716     0.720     0.714     10000
```



4.3 PCA + Logistic Regression, Retain 95% Eigenvalues

As we can see, we keep 153 eigenvectors, set learning rate as 0.00002, iteration time as 200001 and choose stochastic gradient descent to do iterataion. For all categories, the cost is bigger than 1000 for every iteration. The overall accuracy is 87.75%. The micro average accuracy is 0.877; the macro average is 0.876; the weighted average is 0.877.

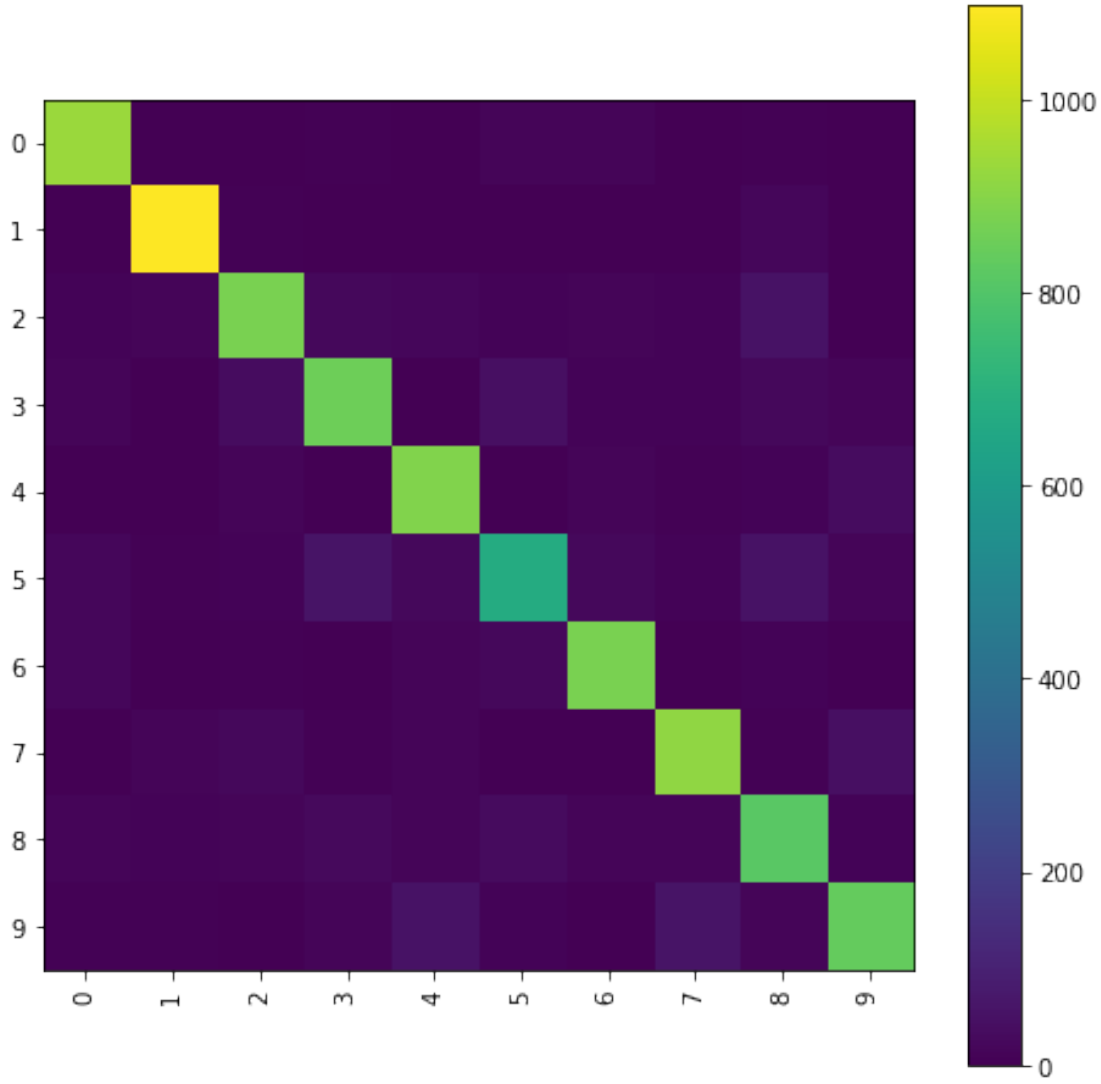
```
In [0]: print(confusion_matrix(y_test, y_pred))
        print(classification_report(y_test, y_pred,
                                    target_names=list(label_dict.values()),digits=3))
        plt.figure(figsize=(8,8))
        cnf_matrix = confusion_matrix(y_test, y_pred)
        classes = list(label_dict.values())
```

```
        plt.imshow(cnf_matrix, interpolation='nearest')
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        _ = plt.xticks(tick_marks, classes, rotation=90)
        _ = plt.yticks(tick_marks, classes)
```

```
[[ 932    0    2    5    1   13   17    1    8    1]
 [   0 1098    5    2    0    3    4    3   20    0]
 [  11   15  877   22   18    9   15    9   53    3]
 [  13    2   36  854    1   45   11   10   24   14]
 [   3    4   16    1  888    4   14    8    9   35]
 [  19    6   11   59   22  674   23   12   53   13]
 [  20    3    6    2   13   22  879    3   10    0]
 [   2   14   23    7   13    4    0  915    6   44]
 [  15   11   17   27   16   32   14   14  818   10]
 [   8    7    1   14   52    9    4   58   16  840]]
              precision    recall  f1-score   support

           0      0.911     0.951     0.931       980
           1      0.947     0.967     0.957      1135
           2      0.882     0.850     0.866      1032
           3      0.860     0.846     0.853      1010
           4      0.867     0.904     0.885       982
           5      0.827     0.756     0.790       892
           6      0.896     0.918     0.907       958
           7      0.886     0.890     0.888      1028
           8      0.804     0.840     0.822       974
           9      0.875     0.833     0.853      1009

   micro avg      0.877     0.877     0.877     10000
   macro avg      0.876     0.875     0.875     10000
weighted avg      0.877     0.877     0.877     10000
```



4.4 Random Forest

Here, to save time, we only build 200 trees, but get good prediction result.

As we can see, the Mean Squared Error(MSE) is 0.7315. The overall accuracy is 96.08%. Then we try different max_depth (3,7,10,13,16,20) to predict and see if the model has potential to improve and we find that none of them have accuracy higher than 96.08% Thus, the current model is the best one. And the micro average accuracy is 0.961; the macro average is 0.961; the weighted average is 0.961.

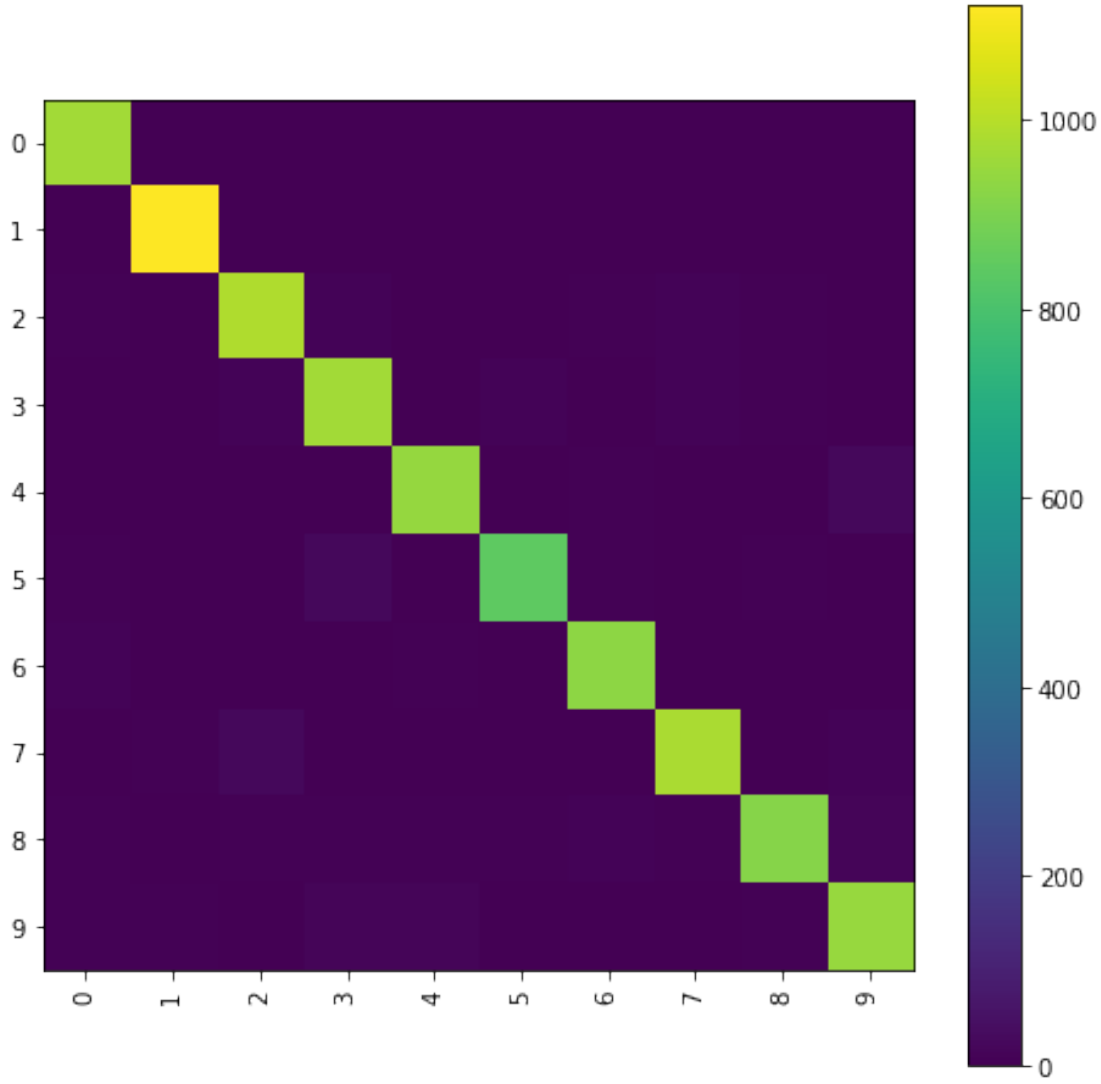
```
In [33]: print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, y_pred,
                                     target_names=list(label_dict.values()),digits=3))
         plt.figure(figsize=(8,8))
```

```
        cnf_matrix = confusion_matrix(y_test, y_pred)
        classes = list(label_dict.values())
        plt.imshow(cnf_matrix, interpolation='nearest')
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        _ = plt.xticks(tick_marks, classes, rotation=90)
        _ = plt.yticks(tick_marks, classes)
```

```
[[ 968    0    0    0    0    4    4    1    3    0]
 [   0 1122    3    3    0    2    2    1    2    0]
 [   6    0  987   10    3    0    7   11    7    1]
 [   1    0   11  968    0   10    0    9    7    4]
 [   1    0    2    0  944    0    8    0    3   24]
 [   5    1    0   22    3  843    8    2    6    2]
 [  11    3    0    0    7    4  931    0    2    0]
 [   1    6   24    1    3    0    0  979    4   10]
 [   5    1    6    6    5    6    9    5  917   14]
 [   5    6    3   14   15    4    2    3    8  949]]
              precision    recall  f1-score   support

           0       0.965     0.988     0.976       980
           1       0.985     0.989     0.987      1135
           2       0.953     0.956     0.955      1032
           3       0.945     0.958     0.952      1010
           4       0.963     0.961     0.962       982
           5       0.966     0.945     0.955       892
           6       0.959     0.972     0.965       958
           7       0.968     0.952     0.960      1028
           8       0.956     0.941     0.949       974
           9       0.945     0.941     0.943      1009

   micro avg       0.961     0.961     0.961     10000
   macro avg       0.961     0.960     0.960     10000
weighted avg       0.961     0.961     0.961     10000
```



4.5 PCA + Random Forest, Retain 90% Eigenvalues

As we can see, we keep 86 eigenvectors, and the Mean Squared Error(MSE) is 3.2954. The overall accuracy is 79.12%. Then we try different max_depth (3,7,10,13,16,20) to predict and see if the model has potential to improve and we find that the accuracies from all the max_depths are below 79.05%. Thus, the current model is the best one. And the micro average accuracy is 0.790; the macro average is 0.789; the weighted average is 0.791.

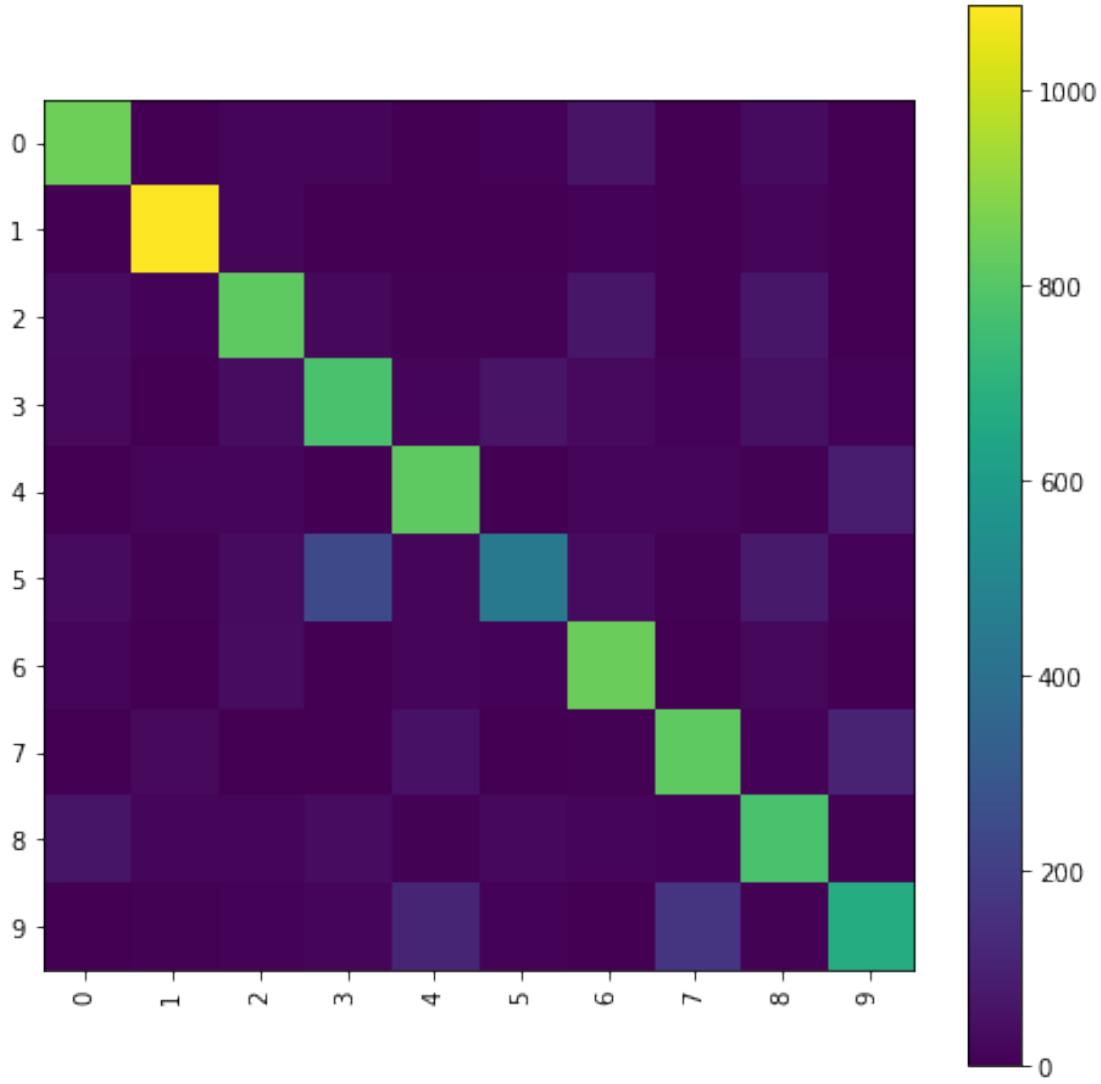
```
In [0]: print(confusion_matrix(y_test, y_pred))
        print(classification_report(y_test, y_pred,
                                    target_names=list(label_dict.values()),digits=3))
        plt.figure(figsize=(8,8))
        cnf_matrix = confusion_matrix(y_test, y_pred)
```

```
        classes = list(label_dict.values())
        plt.imshow(cnf_matrix, interpolation='nearest')
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        _ = plt.xticks(tick_marks, classes, rotation=90)
        _ = plt.yticks(tick_marks, classes)
```

```
[[ 848    0   16   13    2   12   56    0   33    0]
 [   0 1089   13    2    4    0   10    1   16    0]
 [  34    9  817   24    5    6   67    4   66    0]
 [  27    1   36  778   13   57   27   12   49   10]
 [   0   20   19    3  819    1   16   18    5   81]
 [  33    5   33  241   18  439   30    6   78    9]
 [  20    4   37    0   15   11  846    0   25    0]
 [   0   26    4    2   54    0    5  818   10  109]
 [  62   14   16   38    7   24   21    9  777    6]
 [   4    6   10   17  113    9    0  170    6  674]]
              precision    recall  f1-score   support

           0      0.825     0.865     0.845       980
           1      0.928     0.959     0.943      1135
           2      0.816     0.792     0.804      1032
           3      0.696     0.770     0.731      1010
           4      0.780     0.834     0.806       982
           5      0.785     0.492     0.605       892
           6      0.785     0.883     0.831       958
           7      0.788     0.796     0.792      1028
           8      0.730     0.798     0.762       974
           9      0.758     0.668     0.710      1009

   micro avg      0.790     0.790     0.790     10000
   macro avg      0.789     0.786     0.783     10000
weighted avg      0.791     0.790     0.787     10000
```



4.6 PCA + Random Forest, Retain 95% Eigenvalues

As we can see, we keep 153 eigenvectors, and the Mean Squared Error(MSE) is 1.7849. The overall accuracy is 89.28%. Then we try different max_depth (3,7,10,13,16,20) to predict and see if the model has potential to improve and we find that the accuracy from all different max_depth is less than 89.28%. Thus, the current model is the best one. And the micro average accuracy is 0.893; the macro average is 0.892; the weighted average is 0.893.

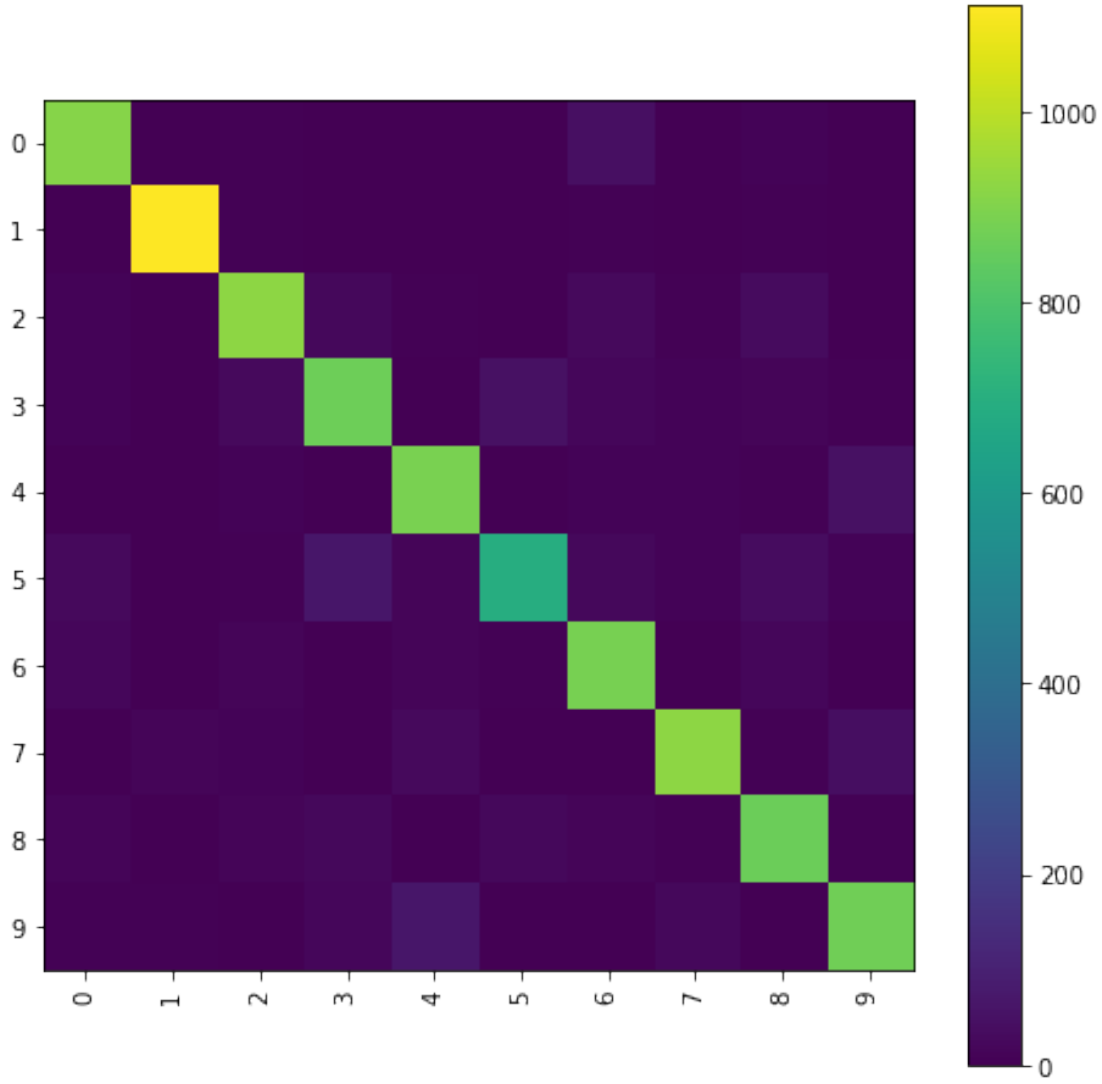
```
In [17]: print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, y_pred,
                                     target_names=list(label_dict.values()),digits=3))
         plt.figure(figsize=(8,8))
         cnf_matrix = confusion_matrix(y_test, y_pred)
```

```
        classes = list(label_dict.values())
        plt.imshow(cnf_matrix, interpolation='nearest')
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        _ = plt.xticks(tick_marks, classes, rotation=90)
        _ = plt.yticks(tick_marks, classes)
```

```
[[ 910    0    8    3    0    3   45    1   10    0]
 [   0 1112    6    4    1    0    5    1    6    0]
 [  12    3  922   23    5    0   29    7   31    0]
 [  12    0   27  866    3   48   20   13   15    6]
 [   0    3   11    0  890    0   10    9    7   52]
 [  27    3    8   66   15  691   23    9   38   12]
 [  19    1   14    1   14    5  883    0   21    0]
 [   0   15   10    1   29    0    2  921    7   43]
 [  17    2   17   22    2   23   16    8  861    6]
 [   6    8    2   19   68    4    1   26    3  872]]
              precision    recall  f1-score   support

           0      0.907     0.929     0.918       980
           1      0.969     0.980     0.975      1135
           2      0.900     0.893     0.896      1032
           3      0.862     0.857     0.860      1010
           4      0.867     0.906     0.886       982
           5      0.893     0.775     0.830       892
           6      0.854     0.922     0.887       958
           7      0.926     0.896     0.911      1028
           8      0.862     0.884     0.873       974
           9      0.880     0.864     0.872      1009

   micro avg      0.893     0.893     0.893     10000
   macro avg      0.892     0.891     0.891     10000
weighted avg      0.893     0.893     0.892     10000
```



5 Discussion

Compared with LR, RF performs apparently better. The following is accuracy table.

Accuracy Table	Basic Model	Retain 90%	Retain 95%
Logistic Regression	91.03%	71.97%	87.75%
Random Forst	96.08%	79.12%	89.28%

Meanwhile, RF takes less time than LR in training on our condition, which means that our scratched class has huge potential to improve the processing time as fast as matured package.

Additionally, given that we use the partial dataset, it is reasonable to believe both models will

bring about better prediction result if we train full dataset.

Generally speaking, if the scale of dataset is proper and there are many factor variables, we will be inclined to random forest model.

6 Appendix

```
In [0]: # import dataset and seperate them as train set and test set
        # index x represents image, index y represents label
        import os
        import random
        import sklearn
        import numpy as np
        import sklearn.metrics
        import tensorflow as tf
        from numpy.linalg import *
        import matplotlib.pyplot as plt
        from sklearn.preprocessing import StandardScaler
        from sklearn.ensemble import RandomForestClassifier
        from sklearn.metrics import classification_report, confusion_matrix
```

```
In [3]: # download MNIST dataset from keras
        (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
        # convert data type to float 32
        x_train=x_train[0:25000,]
        y_train=y_train[0:25000,]
        x_train=np.float32(x_train)
        x_test=np.float32(x_test)
        x_train = x_train.reshape(np.shape(x_train)[0], 28*28)/255.0
        x_test = x_test.reshape(np.shape(x_test)[0], 28*28)/255.0
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
```

```
In [0]: # make sure the 10 classes
        label_dict = {0: '0', 1: '1', 2: '2', 3: '3', 4: '4',
                      5: '5', 6: '6', 7: '7', 8: '8', 9: '9'}
```

6.1 Logistic Regression Class

```
In [0]: # build a LR class
        class SJLogis_Regre(object):
            def __init__(self):
                pass

            def train(self, X, Y):
                self.x_train = X
                self.y_train = Y
```

```
def split_category1(self, category_name):
    yy_train=[]
    for i in range(len(y_train)):
        if (self.y_train[i]==category_name):
            yy_train.append(1)
        else: yy_train.append(0)
    return yy_train

def sigmoid(self, a):
    return 1 / (1 + np.exp(-a))

def log_likelihood1(self, ytrain_c, weight):

    # add intercept
    intercept = np.ones((np.shape(self.x_train)[0], 1))
    xtrain_c = np.hstack((intercept, self.x_train))

    weight = np.reshape(weight,(np.shape(xtrain_c)[1], 1))
    a = np.dot(xtrain_c, weight)

    ll = np.sum( np.multiply(ytrain_c,a.T) - np.log(1+np.exp(a.T)) )
    return ll

def gradient_descent1(self, ytrain_c, learning_rate, iteration_time):

    # add intercept
    intercept = np.ones((np.shape(self.x_train)[0], 1))
    xtrain_c = np.hstack((intercept, self.x_train))

    # initial weight
    weight = np.zeros((1,np.shape(xtrain_c)[1]))
    ytrain_c = np.reshape(ytrain_c,(1, np.shape(ytrain_c)[0]))

    # do iteration
    for i in range(iteration_time):
        a = np.dot(weight, xtrain_c.T)
        pred = self.sigmoid(a)

        diff = ytrain_c - pred

        gradient = np.dot(diff, xtrain_c)
        weight = weight + learning_rate * gradient

        # Print the cost
        if (i % 10000 == 0):
            cost = -self.log_likelihood1(ytrain_c, weight)
```

```
                    print ("the cost in %d step is" %(i),cost)

            return weight

        def get_pcx(self, xtest_c, weight_c):

            add_intercept = np.hstack((1, xtest_c))

            p_c = np.dot(weight_c,add_intercept)
            result_c = self.sigmoid(p_c)
            return result_c

        def predict_c(self, total_result):
            value = np.where(total_result == np.max(total_result))
            return value[0][0]
```

In [0]:
```
# split the train as 10 categories
JS = SJLogis_Regre()
JS.train(x_train,y_train)
y0_train = JS.split_category1(0)
y1_train = JS.split_category1(1)
y2_train = JS.split_category1(2)
y3_train = JS.split_category1(3)
y4_train = JS.split_category1(4)
y5_train = JS.split_category1(5)
y6_train = JS.split_category1(6)
y7_train = JS.split_category1(7)
y8_train = JS.split_category1(8)
y9_train = JS.split_category1(9)
```

In [0]:
```
import time
# calculate weight seperately
learning_rate = 0.00002
iteration_time = 200001
tic = time.time()
w0 = JS.gradient_descent1(y0_train, learning_rate, iteration_time)
w1 = JS.gradient_descent1(y1_train, learning_rate, iteration_time)
w2 = JS.gradient_descent1(y2_train, learning_rate, iteration_time)
w3 = JS.gradient_descent1(y3_train, learning_rate, iteration_time)
w4 = JS.gradient_descent1(y4_train, learning_rate, iteration_time)
w5 = JS.gradient_descent1(y5_train, learning_rate, iteration_time)
w6 = JS.gradient_descent1(y6_train, learning_rate, iteration_time)
w7 = JS.gradient_descent1(y7_train, learning_rate, iteration_time)
w8 = JS.gradient_descent1(y8_train, learning_rate, iteration_time)
w9 = JS.gradient_descent1(y9_train, learning_rate, iteration_time)
toc = time.time()
print('Iteration took %f seconds' %(toc - tic))
```

```
the cost in 0 step is 44580.57417373677
```

```
the cost in 10000 step is 412.8080815512875
the cost in 20000 step is 370.7995168731614
the cost in 30000 step is 349.03500935521964
the cost in 40000 step is 334.941829873847
the cost in 50000 step is 324.77269404959617
the cost in 60000 step is 316.92963141079167
the cost in 70000 step is 310.5973263235666
the cost in 80000 step is 305.3111083395682
the cost in 90000 step is 300.78525776611934
the cost in 100000 step is 296.83421246816937
the cost in 110000 step is 293.3324977335265
the cost in 120000 step is 290.1926229696758
the cost in 130000 step is 287.3519579956931
the cost in 140000 step is 284.7643650903752
the cost in 150000 step is 282.3945779769092
the cost in 160000 step is 280.2144309343582
the cost in 170000 step is 278.20049895215016
the cost in 180000 step is 276.3327746489955
the cost in 190000 step is 274.5939815786974
the cost in 200000 step is 272.96918935784726
the cost in 0 step is 33080.59816744591
the cost in 10000 step is 550.7954826482951
the cost in 20000 step is 502.39294304266343
the cost in 30000 step is 474.8371769083394
the cost in 40000 step is 456.08750304100215
the cost in 50000 step is 442.14216534290154
the cost in 60000 step is 431.13918950006047
the cost in 70000 step is 422.0898601260789
the cost in 80000 step is 414.4224372127335
the cost in 90000 step is 407.78254310020645
the cost in 100000 step is 401.9377208418714
the cost in 110000 step is 396.72813455169273
the cost in 120000 step is 392.0391075465796
the cost in 130000 step is 387.78481705223066
the cost in 140000 step is 383.89844474536545
the cost in 150000 step is 380.32642993846326
the cost in 160000 step is 377.0251490963904
the cost in 170000 step is 373.9588013151387
the cost in 180000 step is 371.0978543060005
the cost in 190000 step is 368.41781150373436
the cost in 200000 step is 365.89822178999714
the cost in 0 step is 44875.62057028671
the cost in 10000 step is 1527.1935188878765
the cost in 20000 step is 1489.7624360660413
the cost in 30000 step is 1473.5812882240882
the cost in 40000 step is 1464.28255546705
the cost in 50000 step is 1458.0791150329392
the cost in 60000 step is 1453.5394920102412
```

```
the cost in 70000 step is 1450.004000446804
the cost in 80000 step is 1447.1263735208063
the cost in 90000 step is 1444.7070456540266
the cost in 100000 step is 1442.6225515197996
the cost in 110000 step is 1440.792128266716
the cost in 120000 step is 1439.1605139974408
the cost in 130000 step is 1437.6884669135852
the cost in 140000 step is 1436.3472416472746
the cost in 150000 step is 1435.115218222465
the cost in 160000 step is 1433.975762001227
the cost in 170000 step is 1432.9158182764052
the cost in 180000 step is 1431.9249618118909
the cost in 190000 step is 1430.994737384277
the cost in 200000 step is 1430.1181918494487
the cost in 0 step is 46327.867893498995
the cost in 10000 step is 1816.2657462181696
the cost in 20000 step is 1787.1868871628421
the cost in 30000 step is 1772.430541748359
the cost in 40000 step is 1762.813429003373
the cost in 50000 step is 1755.8587026929795
the cost in 60000 step is 1750.5340295258807
the cost in 70000 step is 1746.3028148101757
the cost in 80000 step is 1742.846746232482
the cost in 90000 step is 1739.9610531599155
the cost in 100000 step is 1737.507101823621
the cost in 110000 step is 1735.3877013943652
the cost in 120000 step is 1733.5328767750968
the cost in 130000 step is 1731.891113113401
the cost in 140000 step is 1730.4237316630456
the cost in 150000 step is 1729.1011669498246
the cost in 160000 step is 1727.9004444478055
the cost in 170000 step is 1726.8034377948327
the cost in 180000 step is 1725.7956434501432
the cost in 190000 step is 1724.8653052741406
the cost in 200000 step is 1724.0027796964225
the cost in 0 step is 40505.38480764192
the cost in 10000 step is 1058.9724320329392
the cost in 20000 step is 1008.2451543645313
the cost in 30000 step is 982.2594200311594
the cost in 40000 step is 965.1306038814467
the cost in 50000 step is 952.5801959991448
the cost in 60000 step is 942.8508114006631
the cost in 70000 step is 935.0358179852907
the cost in 80000 step is 928.5965332335497
the cost in 90000 step is 923.1839465907186
the cost in 100000 step is 918.5596481128854
the cost in 110000 step is 914.5547658067544
the cost in 120000 step is 911.046219268383
```

```
the cost in 130000 step is 907.9421077669203
the cost in 140000 step is 905.1723554856202
the cost in 150000 step is 902.6825363239252
the cost in 160000 step is 900.4296887680754
the cost in 170000 step is 898.379411905431
the cost in 180000 step is 896.503807090837
the cost in 190000 step is 894.77999067092
the cost in 200000 step is 893.1890004609638
the cost in 0 step is 39654.51584231053
the cost in 10000 step is 1739.7685373162744
the cost in 20000 step is 1687.2992591581701
the cost in 30000 step is 1662.5430717394986
the cost in 40000 step is 1647.091160348093
the cost in 50000 step is 1636.080763112063
the cost in 60000 step is 1627.651831524264
the cost in 70000 step is 1620.9133229460733
the cost in 80000 step is 1615.3647105055325
the cost in 90000 step is 1610.6916363130424
the cost in 100000 step is 1606.6815212279575
the cost in 110000 step is 1603.1837788583264
the cost in 120000 step is 1600.0887140703244
the cost in 130000 step is 1597.3150424818698
the cost in 140000 step is 1594.8017887330095
the cost in 150000 step is 1592.50270063014
the cost in 160000 step is 1590.3822915300527
the cost in 170000 step is 1588.4130121438147
the cost in 180000 step is 1586.5732242336433
the cost in 190000 step is 1584.8457454577533
the cost in 200000 step is 1583.2168010445241
the cost in 0 step is 43057.13023319075
the cost in 10000 step is 743.2105174581436
the cost in 20000 step is 705.3247057774153
the cost in 30000 step is 685.578184657003
the cost in 40000 step is 672.8821467495865
the cost in 50000 step is 663.8870778549996
the cost in 60000 step is 657.1372684968447
the cost in 70000 step is 651.8691983571443
the cost in 80000 step is 647.6346427959295
the cost in 90000 step is 644.1505519906297
the cost in 100000 step is 641.2287442404942
the cost in 110000 step is 638.7391307975424
the cost in 120000 step is 636.5889362581952
the cost in 130000 step is 634.7102634837019
the cost in 140000 step is 633.0523115558625
the cost in 150000 step is 631.576329325664
the cost in 160000 step is 630.2522470128223
the cost in 170000 step is 629.0563724529777
the cost in 180000 step is 627.9697814851294
```

```
the cost in 190000 step is 626.9771714816476
the cost in 200000 step is 626.0660302950516
the cost in 0 step is 40041.1028637697
the cost in 10000 step is 870.0987054949602
the cost in 20000 step is 820.1844442206137
the cost in 30000 step is 795.1644045015648
the cost in 40000 step is 779.3791886041912
the cost in 50000 step is 768.1676140523836
the cost in 60000 step is 759.5843307805343
the cost in 70000 step is 752.6713649924151
the cost in 80000 step is 746.9021552362489
the cost in 90000 step is 741.9618415260306
the cost in 100000 step is 737.6489303608413
the cost in 110000 step is 733.8271476526054
the cost in 120000 step is 730.4000943487407
the cost in 130000 step is 727.2970533157587
the cost in 140000 step is 724.4645980314056
the cost in 150000 step is 721.8613887901209
the cost in 160000 step is 719.4548076207383
the cost in 170000 step is 717.2187011168194
the cost in 180000 step is 715.1318169230318
the cost in 190000 step is 713.1766889817376
the cost in 200000 step is 711.3388210894836
the cost in 0 step is 48774.711780999794
the cost in 10000 step is 2594.1305563907304
the cost in 20000 step is 2566.009362965211
the cost in 30000 step is 2552.9363472600594
the cost in 40000 step is 2544.794264635882
the cost in 50000 step is 2538.9873528679345
the cost in 60000 step is 2534.5351050054383
the cost in 70000 step is 2530.9680014922924
the cost in 80000 step is 2528.0231322470154
the cost in 90000 step is 2525.537651287596
the cost in 100000 step is 2523.403751443929
the cost in 110000 step is 2521.546623361933
the cost in 120000 step is 2519.9124399113703
the cost in 130000 step is 2518.4612952410835
the cost in 140000 step is 2517.1628333398785
the cost in 150000 step is 2515.993437308706
the cost in 160000 step is 2514.934367728348
the cost in 170000 step is 2513.9704981096406
the cost in 180000 step is 2513.0894359943563
the cost in 190000 step is 2512.280898729597
the cost in 200000 step is 2511.536260830699
the cost in 0 step is 44391.80966426329
the cost in 10000 step is 2247.5034380243783
the cost in 20000 step is 2212.715214264926
the cost in 30000 step is 2195.4240583827955
```

```
the cost in 40000 step is 2184.1982696731015
the cost in 50000 step is 2176.0377929010806
the cost in 60000 step is 2169.733048193807
the cost in 70000 step is 2164.6769022977046
the cost in 80000 step is 2160.5177222489415
the cost in 90000 step is 2157.030070925836
the cost in 100000 step is 2154.059335548022
the cost in 110000 step is 2151.4948776040324
the cost in 120000 step is 2149.255236758325
the cost in 130000 step is 2147.2791309216004
the cost in 140000 step is 2145.519634292278
the cost in 150000 step is 2143.940262389375
the cost in 160000 step is 2142.512265093066
the cost in 170000 step is 2141.2127127018366
the cost in 180000 step is 2140.0231169173326
the cost in 190000 step is 2138.928420883862
the cost in 200000 step is 2137.9162488085763
Iteration took 74444.132998 seconds
```

```
In [0]: # calculate probability for each category
        y_pred = []
        for i in range(np.shape(x_test)[0]):

            pred_0 = JS.get_pcx(x_test[i], w0)
            pred_1 = JS.get_pcx(x_test[i], w1)
            pred_2 = JS.get_pcx(x_test[i], w2)
            pred_3 = JS.get_pcx(x_test[i], w3)
            pred_4 = JS.get_pcx(x_test[i], w4)
            pred_5 = JS.get_pcx(x_test[i], w5)
            pred_6 = JS.get_pcx(x_test[i], w6)
            pred_7 = JS.get_pcx(x_test[i], w7)
            pred_8 = JS.get_pcx(x_test[i], w8)
            pred_9 = JS.get_pcx(x_test[i], w9)
            pred = [pred_0[0],pred_1[0],pred_2[0],pred_3[0],pred_4[0],
                    pred_5[0],pred_6[0],pred_7[0],pred_8[0],pred_9[0]]
            value = JS.predict_c(pred)
            y_pred.append(value)
```

```
In [0]: # calculate accuracy
        num_test = len(y_test)
        num_correct = np.sum(y_pred == y_test)
        print('Got %d / %d correct' % (num_correct, num_test))
        print('Accuracy = %f' % (np.mean(y_test == y_pred)))
```

```
Got 9103 / 10000 correct
Accuracy = 0.910300
```

```
In [0]: from sklearn.metrics import classification_report, confusion_matrix
        print(confusion_matrix(y_test, y_pred))
        print(classification_report(y_test, y_pred,
                                    target_names=list(label_dict.values()),digits=3))
```

```
[[ 948    0    4    3    0   11    8    2    2    2]
 [   0 1109    3    2    1    1    4    1   13    1]
 [   7   12  909   28    9    4   10   12   38    3]
 [   4    3   20  913    4   20    3   13   22    8]
 [   1    1    5    6  910    0   10    4    9   36]
 [  13    4    7   36    8  765   11    6   36    6]
 [  10    3   13    0    4   22  902    1    3    0]
 [   4    8   20    4    7    4    1  932    6   42]
 [   9   11   10   23   14   28   15   15  836   13]
 [   8    9    1   18   37   14    0   30   13  879]]
              precision    recall  f1-score   support

           0      0.944     0.967     0.956       980
           1      0.956     0.977     0.966      1135
           2      0.916     0.881     0.898      1032
           3      0.884     0.904     0.894      1010
           4      0.915     0.927     0.921       982
           5      0.880     0.858     0.869       892
           6      0.936     0.942     0.939       958
           7      0.917     0.907     0.912      1028
           8      0.855     0.858     0.857       974
           9      0.888     0.871     0.879      1009

   micro avg      0.910     0.910     0.910     10000
   macro avg      0.909     0.909     0.909     10000
weighted avg      0.910     0.910     0.910     10000
```

```
In [0]: # PCA + Logistic Regression
        ## PCA Class
```

```
In [0]: # download MNIST dataset from keras
        (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
        # convert data type to float 32
        x_train=x_train
        y_train=y_train
        x_train=np.float32(x_train)
        x_test=np.float32(x_test)
        x_train = x_train.reshape(np.shape(x_train)[0], 28*28)/255.0
        x_test = x_test.reshape(np.shape(x_test)[0], 28*28)/255.0
```

```
In [0]: # stack them as a big one for dimension deduction
        big_X=np.vstack((x_train,x_test))
```

```
In [0]: # build PCA class
        class SJPCA(object):
            def __init__(self):
                pass

            def train(self, X):
                self.x_train = X

            def compute_mean_covar_eigen(self):
                # get average image and get mean image by summing each row
                tr_mean = np.mean(self.x_train, axis=0)
                tr_mean = np.reshape(tr_mean,(1,np.shape(tr_mean)[0]))

                # subtract the mean
                xtr_m = self.x_train - tr_mean

                # calculate covariance matrix
                tr_cov = np.dot(xtr_m.T,xtr_m)

                # get eigenvalue and eigenvector
                tr_val, tr_vec = eig(tr_cov)

                return xtr_m, tr_cov, tr_val, tr_vec

            def get_comp_K(self,tr_val, threshold):
                cum_lambda = np.cumsum(tr_val)
                total_lamda = cum_lambda[-1]

                # get the principal component number that we want to keep
                for keep_dim in range(len(tr_val)):
                    rate = cum_lambda[keep_dim]/total_lamda
                    if rate >= threshold:
                        return keep_dim
                        break
                    else: continue

            def deduct_img(self, tr_vec, keep_dim):
                x_proj= np.dot(self.x_train, tr_vec.T[:,0:keep_dim])
                return x_proj
```

6.2 Retain 90% Eigenvalues

```
In [0]: # Deduct Training Set
        SJ = SJPCA()
        SJ.train(big_X)
        xtr_m, tr_cov, tr_val, tr_vec = SJ.compute_mean_covar_eigen()
        keep_dim = SJ.get_comp_K(tr_val, 0.90)
        new_big_X = SJ.deduct_img(tr_vec, keep_dim)
```

```
        print(keep_dim)
```

```
86
```

In [0]:
```
# resplit the dataset and normalize them with min-max normalization
x_train = new_big_X[0:60000,:]
x_test = new_big_X[60000:70000,:]
tr_min = np.min(x_train,axis=1)
tr_cha = np.max(x_train,axis=1)-np.min(x_train,axis=1)
te_min = np.min(x_test,axis=1)
te_cha = np.max(x_test,axis=1)-np.min(x_test,axis=1)
for i in range(60000):
    x_train[i]=(x_train[i]-tr_min[i])/tr_cha[i]
for i in range(10000):
    x_test[i]=(x_test[i]-te_min[i])/te_cha[i]
```

In [0]:
```
# split the train as 10 categories
x_train = x_train[0:25000,]
y_train = y_train[0:25000,]
JS = SJLogis_Regre()
JS.train(x_train,y_train)
y0_train = JS.split_category1(0)
y1_train = JS.split_category1(1)
y2_train = JS.split_category1(2)
y3_train = JS.split_category1(3)
y4_train = JS.split_category1(4)
y5_train = JS.split_category1(5)
y6_train = JS.split_category1(6)
y7_train = JS.split_category1(7)
y8_train = JS.split_category1(8)
y9_train = JS.split_category1(9)
```

In [0]:
```
import time
# calculate weight seperately
learning_rate = 0.00002
iteration_time = 200001
tic = time.time()
w0 = JS.gradient_descent1(y0_train, learning_rate, iteration_time)
w1 = JS.gradient_descent1(y1_train, learning_rate, iteration_time)
w2 = JS.gradient_descent1(y2_train, learning_rate, iteration_time)
w3 = JS.gradient_descent1(y3_train, learning_rate, iteration_time)
w4 = JS.gradient_descent1(y4_train, learning_rate, iteration_time)
w5 = JS.gradient_descent1(y5_train, learning_rate, iteration_time)
w6 = JS.gradient_descent1(y6_train, learning_rate, iteration_time)
w7 = JS.gradient_descent1(y7_train, learning_rate, iteration_time)
w8 = JS.gradient_descent1(y8_train, learning_rate, iteration_time)
w9 = JS.gradient_descent1(y9_train, learning_rate, iteration_time)
```

```
        toc = time.time()
        print('Iteration took %f seconds' %(toc - tic))
```

```
the cost in 0 step is 11494.348784778926
the cost in 10000 step is 3126.412584415647
the cost in 20000 step is 3019.222058121433
the cost in 30000 step is 2980.548900777659
the cost in 40000 step is 2960.0637610517406
the cost in 50000 step is 2947.133525428829
the cost in 60000 step is 2938.150683857481
the cost in 70000 step is 2931.5252993140994
the cost in 80000 step is 2926.4329330647543
the cost in 90000 step is 2922.3987258713228
the cost in 100000 step is 2919.12874921704
the cost in 110000 step is 2916.4309997558444
the cost in 120000 step is 2914.1744244347587
the cost in 130000 step is 2912.266118229002
the cost in 140000 step is 2910.637984798607
the cost in 150000 step is 2909.238631302235
the cost in 160000 step is 2908.0282850389312
the cost in 170000 step is 2906.9755114693517
the cost in 180000 step is 2906.0550344696912
the cost in 190000 step is 2905.24624699049
the cost in 200000 step is 2904.5321641235228
the cost in 0 step is 13195.457526007835
the cost in 10000 step is 2605.1631217709282
the cost in 20000 step is 2527.082562647482
the cost in 30000 step is 2499.6754922581245
the cost in 40000 step is 2485.788203774236
the cost in 50000 step is 2477.2256707642114
the cost in 60000 step is 2471.252817453685
the cost in 70000 step is 2466.730449477056
the cost in 80000 step is 2463.109379705213
the cost in 90000 step is 2460.0948624613125
the cost in 100000 step is 2457.514770864773
the cost in 110000 step is 2455.26136017355
the cost in 120000 step is 2453.2631403257556
the cost in 130000 step is 2451.470289348657
the cost in 140000 step is 2449.8466256742327
the cost in 150000 step is 2448.3649560534345
the cost in 160000 step is 2447.004252767274
the cost in 170000 step is 2445.7478683169693
the cost in 180000 step is 2444.5823631870967
the cost in 190000 step is 2443.496709734964
the cost in 200000 step is 2442.4817349064124
the cost in 0 step is 11993.164788890155
the cost in 10000 step is 4253.653600759702
the cost in 20000 step is 4185.437801964854
```

```
the cost in 30000 step is 4163.6179129090915
the cost in 40000 step is 4153.833177792004
the cost in 50000 step is 4148.465262167557
the cost in 60000 step is 4144.997092878471
the cost in 70000 step is 4142.453628649349
the cost in 80000 step is 4140.418135989224
the cost in 90000 step is 4138.696054782977
the cost in 100000 step is 4137.187631041936
the cost in 110000 step is 4135.836447054798
the cost in 120000 step is 4134.607464676719
the cost in 130000 step is 4133.477112887989
the cost in 140000 step is 4132.428514731237
the cost in 150000 step is 4131.449016214509
the cost in 160000 step is 4130.528805732905
the cost in 170000 step is 4129.660084220133
the cost in 180000 step is 4128.836533230995
the cost in 190000 step is 4128.052955577392
the cost in 200000 step is 4127.30502232009
the cost in 0 step is 11774.667289332769
the cost in 10000 step is 4750.796715164135
the cost in 20000 step is 4713.998764948993
the cost in 30000 step is 4698.930887031426
the cost in 40000 step is 4688.53318221257
the cost in 50000 step is 4680.67877827822
the cost in 60000 step is 4674.551664926897
the cost in 70000 step is 4669.680736350771
the cost in 80000 step is 4665.754818964564
the cost in 90000 step is 4662.55463788173
the cost in 100000 step is 4659.919566124383
the cost in 110000 step is 4657.728939952542
the cost in 120000 step is 4655.890558134034
the cost in 130000 step is 4654.333116809073
the cost in 140000 step is 4653.0009837541165
the cost in 150000 step is 4651.850457853401
the cost in 160000 step is 4650.847021825205
the cost in 170000 step is 4649.963286298036
the cost in 180000 step is 4649.177429841478
the cost in 190000 step is 4648.472002930839
the cost in 200000 step is 4647.833003566164
the cost in 0 step is 11174.437518526476
the cost in 10000 step is 3798.2646136405097
the cost in 20000 step is 3705.0244009201565
the cost in 30000 step is 3655.5708216293774
the cost in 40000 step is 3622.8815114186787
the cost in 50000 step is 3599.319775265019
the cost in 60000 step is 3581.4634712570905
the cost in 70000 step is 3567.4562005125076
the cost in 80000 step is 3556.180880429334
```

```
the cost in 90000 step is 3546.9189532149076
the cost in 100000 step is 3539.185338874341
the cost in 110000 step is 3532.6398782979672
the cost in 120000 step is 3527.0363876786523
the cost in 130000 step is 3522.191837814985
the cost in 140000 step is 3517.9669701581934
the cost in 150000 step is 3514.2537077720235
the cost in 160000 step is 3510.9667452725607
the cost in 170000 step is 3508.037783331035
the cost in 180000 step is 3505.4114792046084
the cost in 190000 step is 3503.042536846499
the cost in 200000 step is 3500.893570687847
the cost in 0 step is 10716.23967403196
the cost in 10000 step is 5436.415826873967
the cost in 20000 step is 5414.813068150808
the cost in 30000 step is 5406.710458184604
the cost in 40000 step is 5402.006825231792
the cost in 50000 step is 5398.800746295721
the cost in 60000 step is 5396.413353565801
the cost in 70000 step is 5394.531698486926
the cost in 80000 step is 5392.987838659084
the cost in 90000 step is 5391.682122950585
the cost in 100000 step is 5390.551079925108
the cost in 110000 step is 5389.552123917892
the cost in 120000 step is 5388.655542301073
the cost in 130000 step is 5387.839971913611
the cost in 140000 step is 5387.089685669847
the cost in 150000 step is 5386.392881964512
the cost in 160000 step is 5385.740561547407
the cost in 170000 step is 5385.1257654675055
the cost in 180000 step is 5384.543044173435
the cost in 190000 step is 5383.988079737385
the cost in 200000 step is 5383.457412405318
the cost in 0 step is 12859.308589492144
the cost in 10000 step is 3005.5981288791972
the cost in 20000 step is 2956.692565328443
the cost in 30000 step is 2943.685460772231
the cost in 40000 step is 2937.0087742005244
the cost in 50000 step is 2932.3870519743114
the cost in 60000 step is 2928.760658782249
the cost in 70000 step is 2925.738812734335
the cost in 80000 step is 2923.12993209388
the cost in 90000 step is 2920.8224317954337
the cost in 100000 step is 2918.7445519804164
the cost in 110000 step is 2916.847236335628
the cost in 120000 step is 2915.095503525564
the cost in 130000 step is 2913.4635959083494
the cost in 140000 step is 2911.9320490031478
```

```
the cost in 150000 step is 2910.485830055776
the cost in 160000 step is 2909.1131098477463
the cost in 170000 step is 2907.8044270487117
the cost in 180000 step is 2906.5521047408456
the cost in 190000 step is 2905.349833795178
the cost in 200000 step is 2904.19236948099
the cost in 0 step is 11515.196386206519
the cost in 10000 step is 3927.0088302984695
the cost in 20000 step is 3838.3086231997477
the cost in 30000 step is 3801.4069304543887
the cost in 40000 step is 3779.6211128270634
the cost in 50000 step is 3764.541929526753
the cost in 60000 step is 3753.120653481457
the cost in 70000 step is 3743.9754571195313
the cost in 80000 step is 3736.3835091469837
the cost in 90000 step is 3729.9232784664614
the cost in 100000 step is 3724.3271330031675
the cost in 110000 step is 3719.413658292306
the cost in 120000 step is 3715.0535652833064
the cost in 130000 step is 3711.151066742504
the cost in 140000 step is 3707.6329812625895
the cost in 150000 step is 3704.4419896256645
the cost in 160000 step is 3701.5322661376663
the cost in 170000 step is 3698.8665405674983
the cost in 180000 step is 3696.4140592813555
the cost in 190000 step is 3694.149131826925
the cost in 200000 step is 3692.0500702217582
the cost in 0 step is 11479.217210392377
the cost in 10000 step is 5342.85144675758
the cost in 20000 step is 5284.176358784428
the cost in 30000 step is 5262.125851555122
the cost in 40000 step is 5250.825193455512
the cost in 50000 step is 5243.926832598075
the cost in 60000 step is 5239.112025853081
the cost in 70000 step is 5235.411207281902
the cost in 80000 step is 5232.375465471756
the cost in 90000 step is 5229.7764376520345
the cost in 100000 step is 5227.487120984874
the cost in 110000 step is 5225.430862671986
the cost in 120000 step is 5223.557975287069
the cost in 130000 step is 5221.834276587008
the cost in 140000 step is 5220.235093559294
the cost in 150000 step is 5218.741921589191
the cost in 160000 step is 5217.340450753148
the cost in 170000 step is 5216.0193369679455
the cost in 180000 step is 5214.769401307368
the cost in 190000 step is 5213.583087844729
the cost in 200000 step is 5212.454084550456
```

```
the cost in 0 step is 11023.377253398028
the cost in 10000 step is 4876.590964464908
the cost in 20000 step is 4856.594228990089
the cost in 30000 step is 4847.622860851587
the cost in 40000 step is 4841.871770455712
the cost in 50000 step is 4837.698306952079
the cost in 60000 step is 4834.46388330204
the cost in 70000 step is 4831.84183116465
the cost in 80000 step is 4829.640220184657
the cost in 90000 step is 4827.737962687289
the cost in 100000 step is 4826.055439231063
the cost in 110000 step is 4824.538798048574
the cost in 120000 step is 4823.15080248661
the cost in 130000 step is 4821.865211698995
the cost in 140000 step is 4820.6632102301965
the cost in 150000 step is 4819.531081336323
the cost in 160000 step is 4818.458659884766
the cost in 170000 step is 4817.438286108525
the cost in 180000 step is 4816.464087732435
the cost in 190000 step is 4815.531481129755
the cost in 200000 step is 4814.636820770846
Iteration took 8354.994221 seconds
```

```
In [0]: # calculate probability for each category
        y_pred = []
        for i in range(np.shape(x_test)[0]):

            pred_0 = JS.get_pcx(x_test[i], w0)
            pred_1 = JS.get_pcx(x_test[i], w1)
            pred_2 = JS.get_pcx(x_test[i], w2)
            pred_3 = JS.get_pcx(x_test[i], w3)
            pred_4 = JS.get_pcx(x_test[i], w4)
            pred_5 = JS.get_pcx(x_test[i], w5)
            pred_6 = JS.get_pcx(x_test[i], w6)
            pred_7 = JS.get_pcx(x_test[i], w7)
            pred_8 = JS.get_pcx(x_test[i], w8)
            pred_9 = JS.get_pcx(x_test[i], w9)
            pred = [pred_0[0],pred_1[0],pred_2[0],pred_3[0],pred_4[0],
                    pred_5[0],pred_6[0],pred_7[0],pred_8[0],pred_9[0]]
            value = JS.predict_c(pred)
            y_pred.append(value)
```

```
In [0]: # calculate accuracy
        num_test = len(y_test)
        num_correct = np.sum(y_pred == y_test)
        print('Got %d / %d correct' % (num_correct, num_test))
        print('Accuracy = %f' % (np.mean(y_test == y_pred)))
```

```
Got 7197 / 10000 correct
Accuracy = 0.719700
```

```
In [0]: print(confusion_matrix(y_test, y_pred))
        print(classification_report(y_test, y_pred,
                                    target_names=list(label_dict.values()),digits=3))
```

```
[[ 819    0   14   37    4   28   51    2   22    3]
 [   1 1037   21   11   14    4    7    1   38    1]
 [  34   55  685   45   24   28   84    3   67    7]
 [  40   13   61  731   17   49   28   17   35   19]
 [   6   41   31   20  745   15   19   13   19   73]
 [  38   30   34  199   52  375   44    9   61   50]
 [  42   10   50    5   33    9  784    5   19    1]
 [   3   34   10   14   41    9    8  784   17  108]
 [  92   43   27   43   37   58   31   11  613   19]
 [  10    8    9   27   75   16    2  210   28  624]]
              precision    recall  f1-score   support

           0      0.755     0.836     0.793       980
           1      0.816     0.914     0.862      1135
           2      0.727     0.664     0.694      1032
           3      0.646     0.724     0.683      1010
           4      0.715     0.759     0.736       982
           5      0.635     0.420     0.506       892
           6      0.741     0.818     0.778       958
           7      0.743     0.763     0.753      1028
           8      0.667     0.629     0.648       974
           9      0.690     0.618     0.652      1009

   micro avg      0.720     0.720     0.720     10000
   macro avg      0.713     0.714     0.710     10000
weighted avg      0.716     0.720     0.714     10000
```

6.3 Retain 95% Eigenvalues

```
In [0]: # Deduct Training Set
        SJ = SJPCA()
        SJ.train(big_X)
        xtr_m, tr_cov, tr_val, tr_vec = SJ.compute_mean_covar_eigen()
        keep_dim = SJ.get_comp_K(tr_val, 0.95)
        new_big_X = SJ.deduct_img(tr_vec, keep_dim)
        print(keep_dim)
```

```
153
```

```
In [0]: # resplit the dataset and normalize them with min-max normalization
        x_train = new_big_X[0:60000,:]
        x_test = new_big_X[60000:70000,:]
        tr_min = np.min(x_train,axis=1)
        tr_cha = np.max(x_train,axis=1)-np.min(x_train,axis=1)
        te_min = np.min(x_test,axis=1)
        te_cha = np.max(x_test,axis=1)-np.min(x_test,axis=1)
        for i in range(60000):
            x_train[i]=(x_train[i]-tr_min[i])/tr_cha[i]
        for i in range(10000):
            x_test[i]=(x_test[i]-te_min[i])/te_cha[i]
```

```
In [0]: # split the train as 10 categories
        x_train = x_train[0:25000,]
        y_train = y_train[0:25000,]
        JS = SJLogis_Regre()
        JS.train(x_train,y_train)
        y0_train = JS.split_category1(0)
        y1_train = JS.split_category1(1)
        y2_train = JS.split_category1(2)
        y3_train = JS.split_category1(3)
        y4_train = JS.split_category1(4)
        y5_train = JS.split_category1(5)
        y6_train = JS.split_category1(6)
        y7_train = JS.split_category1(7)
        y8_train = JS.split_category1(8)
        y9_train = JS.split_category1(9)
```

```
In [0]: import time
        # calculate weight seperately
        learning_rate = 0.00002
        iteration_time = 200001
        tic = time.time()
        w0 = JS.gradient_descent1(y0_train, learning_rate, iteration_time)
        w1 = JS.gradient_descent1(y1_train, learning_rate, iteration_time)
        w2 = JS.gradient_descent1(y2_train, learning_rate, iteration_time)
        w3 = JS.gradient_descent1(y3_train, learning_rate, iteration_time)
        w4 = JS.gradient_descent1(y4_train, learning_rate, iteration_time)
        w5 = JS.gradient_descent1(y5_train, learning_rate, iteration_time)
        w6 = JS.gradient_descent1(y6_train, learning_rate, iteration_time)
        w7 = JS.gradient_descent1(y7_train, learning_rate, iteration_time)
        w8 = JS.gradient_descent1(y8_train, learning_rate, iteration_time)
        w9 = JS.gradient_descent1(y9_train, learning_rate, iteration_time)
        toc = time.time()
        print('Iteration took %f seconds' %(toc - tic))
```

```
the cost in 0 step is 21481.39931363333
the cost in 10000 step is 1290.979156247232
```

```
the cost in 20000 step is 1179.9316720640436
the cost in 30000 step is 1136.8498801548822
the cost in 40000 step is 1113.610628108722
the cost in 50000 step is 1098.8877151910638
the cost in 60000 step is 1088.626968789076
the cost in 70000 step is 1081.0106065444509
the cost in 80000 step is 1075.0973584090684
the cost in 90000 step is 1070.3489266354475
the cost in 100000 step is 1066.434427246745
the cost in 110000 step is 1063.1389819309168
the cost in 120000 step is 1060.3168282585434
the cost in 130000 step is 1057.8654896608593
the cost in 140000 step is 1055.7107270994716
the cost in 150000 step is 1053.7973652048368
the cost in 160000 step is 1052.0834835329895
the cost in 170000 step is 1050.5366169529138
the cost in 180000 step is 1049.1311985537595
the cost in 190000 step is 1047.846794906433
the cost in 200000 step is 1046.6668605452846
the cost in 0 step is 24718.107437320756
the cost in 10000 step is 1151.1126723662248
the cost in 20000 step is 1066.7136659425819
the cost in 30000 step is 1027.6167318068062
the cost in 40000 step is 1003.9998753464371
the cost in 50000 step is 988.0094353940834
the cost in 60000 step is 976.4594199220651
the cost in 70000 step is 967.7520814664608
the cost in 80000 step is 960.979598913985
the cost in 90000 step is 955.5820962287655
the cost in 100000 step is 951.1937271701398
the cost in 110000 step is 947.5649766267343
the cost in 120000 step is 944.5199778282936
the cost in 130000 step is 941.9314899674977
the cost in 140000 step is 939.7054594937763
the cost in 150000 step is 937.7710981774508
the cost in 160000 step is 936.0742930364772
the cost in 170000 step is 934.5731091289713
the cost in 180000 step is 933.234650187493
the cost in 190000 step is 932.0328243210581
the cost in 200000 step is 930.9467269463917
the cost in 0 step is 21218.52266257611
the cost in 10000 step is 2608.734637799854
the cost in 20000 step is 2486.454143185559
the cost in 30000 step is 2437.877724271532
the cost in 40000 step is 2411.8854477581235
the cost in 50000 step is 2396.0875464441506
the cost in 60000 step is 2385.7768270080437
the cost in 70000 step is 2378.7126633745856
```

```
the cost in 80000 step is 2373.685525144285
the cost in 90000 step is 2369.990643722943
the cost in 100000 step is 2367.1958621303056
the cost in 110000 step is 2365.026052464628
the cost in 120000 step is 2363.3007313845874
the cost in 130000 step is 2361.898455859137
the cost in 140000 step is 2360.735657288192
the cost in 150000 step is 2359.7536535561653
the cost in 160000 step is 2358.9104681652975
the cost in 170000 step is 2358.1755592719614
the cost in 180000 step is 2357.526353823065
the cost in 190000 step is 2356.9459255534844
the cost in 200000 step is 2356.4214118910286
the cost in 0 step is 21446.709402156328
the cost in 10000 step is 2850.00157956573
the cost in 20000 step is 2772.696637781348
the cost in 30000 step is 2744.191092728533
the cost in 40000 step is 2729.400104717881
the cost in 50000 step is 2720.184920198146
the cost in 60000 step is 2713.767087775794
the cost in 70000 step is 2708.9654594612743
the cost in 80000 step is 2705.1960044236616
the cost in 90000 step is 2702.134881409138
the cost in 100000 step is 2699.585785238403
the cost in 110000 step is 2697.4212559585594
the cost in 120000 step is 2695.554086582456
the cost in 130000 step is 2693.9221847954063
the cost in 140000 step is 2692.4799592895497
the cost in 150000 step is 2691.193110456793
the cost in 160000 step is 2690.035317429438
the cost in 170000 step is 2688.986043480984
the cost in 180000 step is 2688.0290336786334
the cost in 190000 step is 2687.1512586001163
the cost in 200000 step is 2686.342155138763
the cost in 0 step is 20795.722369229996
the cost in 10000 step is 2070.384883555123
the cost in 20000 step is 1963.9199870251264
the cost in 30000 step is 1922.5514990173167
the cost in 40000 step is 1900.5117452659697
the cost in 50000 step is 1886.7223966820982
the cost in 60000 step is 1877.2139147898383
the cost in 70000 step is 1870.2263514889162
the cost in 80000 step is 1864.861562391915
the cost in 90000 step is 1860.6114038332978
the cost in 100000 step is 1857.164506701705
the cost in 110000 step is 1854.3176741517289
the cost in 120000 step is 1851.9315195784166
the cost in 130000 step is 1849.9065849012675
```

```
the cost in 140000 step is 1848.1696613729766
the cost in 150000 step is 1846.6655194813204
the cost in 160000 step is 1845.3516677710577
the cost in 170000 step is 1844.194892776376
the cost in 180000 step is 1843.1688933879636
the cost in 190000 step is 1842.2526147075337
the cost in 200000 step is 1841.4290448339698
the cost in 0 step is 19317.28222289082
the cost in 10000 step is 3575.185382029389
the cost in 20000 step is 3460.5829954966475
the cost in 30000 step is 3417.689616286633
the cost in 40000 step is 3396.6358994390694
the cost in 50000 step is 3384.7145562205437
the cost in 60000 step is 3377.190245329684
the cost in 70000 step is 3371.9941216118696
the cost in 80000 step is 3368.130168217673
the cost in 90000 step is 3365.0831494260724
the cost in 100000 step is 3362.570036685225
the cost in 110000 step is 3360.426715910553
the cost in 120000 step is 3358.5530994346514
the cost in 130000 step is 3356.8851793819013
the cost in 140000 step is 3355.38016591122
the cost in 150000 step is 3354.008260526704
the cost in 160000 step is 3352.747925925376
the cost in 170000 step is 3351.583066013601
the cost in 180000 step is 3350.50128616405
the cost in 190000 step is 3349.4927851182792
the cost in 200000 step is 3348.549628239625
the cost in 0 step is 20901.197291827262
the cost in 10000 step is 1519.5377431493357
the cost in 20000 step is 1444.7516444278635
the cost in 30000 step is 1414.820489172695
the cost in 40000 step is 1398.7687626530499
the cost in 50000 step is 1388.8322942594725
the cost in 60000 step is 1382.086372297032
the cost in 70000 step is 1377.1955498954237
the cost in 80000 step is 1373.4714706326677
the cost in 90000 step is 1370.5268377871892
the cost in 100000 step is 1368.128402444802
the cost in 110000 step is 1366.127763215047
the cost in 120000 step is 1364.4261340264209
the cost in 130000 step is 1362.955230591273
the cost in 140000 step is 1361.6663340576897
the cost in 150000 step is 1360.5237407143043
the cost in 160000 step is 1359.5006794835951
the cost in 170000 step is 1358.576677253663
the cost in 180000 step is 1357.7358057669546
the cost in 190000 step is 1356.9654833531415
```

```
the cost in 200000 step is 1356.2556363871279
the cost in 0 step is 22035.56148843189
the cost in 10000 step is 1977.5659705585051
the cost in 20000 step is 1889.0166862059818
the cost in 30000 step is 1853.2871074041734
the cost in 40000 step is 1833.672836196149
the cost in 50000 step is 1821.1783931912528
the cost in 60000 step is 1812.4340081164917
the cost in 70000 step is 1805.896393868115
the cost in 80000 step is 1800.765827619451
the cost in 90000 step is 1796.5900928049737
the cost in 100000 step is 1793.0956220990286
the cost in 110000 step is 1790.1075074530365
the cost in 120000 step is 1787.5085474568584
the cost in 130000 step is 1785.216968812816
the cost in 140000 step is 1783.1736862886985
the cost in 150000 step is 1781.334698726024
the cost in 160000 step is 1779.6663757001656
the cost in 170000 step is 1778.1424347969544
the cost in 180000 step is 1776.7419420831586
the cost in 190000 step is 1775.447951354875
the cost in 200000 step is 1774.2465536903944
the cost in 0 step is 20576.308660686824
the cost in 10000 step is 3444.047551910203
the cost in 20000 step is 3377.4839662842332
the cost in 30000 step is 3350.8809413137596
the cost in 40000 step is 3335.833799625315
the cost in 50000 step is 3326.089131294426
the cost in 60000 step is 3319.2895856233527
the cost in 70000 step is 3314.2954441476877
the cost in 80000 step is 3310.478970401891
the cost in 90000 step is 3307.4659237536894
the cost in 100000 step is 3305.0206915983354
the cost in 110000 step is 3302.9887299299826
the cost in 120000 step is 3301.2652894855755
the cost in 130000 step is 3299.7773979977815
the cost in 140000 step is 3298.4730033611722
the cost in 150000 step is 3297.314192991264
the cost in 160000 step is 3296.272831530966
the cost in 170000 step is 3295.3276825717717
the cost in 180000 step is 3294.4624671361757
the cost in 190000 step is 3293.664527990388
the cost in 200000 step is 3292.923894195433
the cost in 0 step is 21137.078491081204
the cost in 10000 step is 3537.109339428206
the cost in 20000 step is 3436.0548331330574
the cost in 30000 step is 3398.3005635758823
the cost in 40000 step is 3378.225014121994
```

```
the cost in 50000 step is 3365.338215687117
the cost in 60000 step is 3356.115688711854
the cost in 70000 step is 3349.057324853924
the cost in 80000 step is 3343.4112057817606
the cost in 90000 step is 3338.7535731521043
the cost in 100000 step is 3334.8239112069236
the cost in 110000 step is 3331.451042581434
the cost in 120000 step is 3328.516487313927
the cost in 130000 step is 3325.934791665879
the cost in 140000 step is 3323.642260327631
the cost in 150000 step is 3321.590141850532
the cost in 160000 step is 3319.740311377656
the cost in 170000 step is 3318.062423498491
the cost in 180000 step is 3316.5319681548644
the cost in 190000 step is 3315.128902596308
the cost in 200000 step is 3313.8366633633846
Iteration took 4603.677029 seconds
```

```
In [0]: # calculate probability for each category
        y_pred = []
        for i in range(np.shape(x_test)[0]):

            pred_0 = JS.get_pcx(x_test[i], w0)
            pred_1 = JS.get_pcx(x_test[i], w1)
            pred_2 = JS.get_pcx(x_test[i], w2)
            pred_3 = JS.get_pcx(x_test[i], w3)
            pred_4 = JS.get_pcx(x_test[i], w4)
            pred_5 = JS.get_pcx(x_test[i], w5)
            pred_6 = JS.get_pcx(x_test[i], w6)
            pred_7 = JS.get_pcx(x_test[i], w7)
            pred_8 = JS.get_pcx(x_test[i], w8)
            pred_9 = JS.get_pcx(x_test[i], w9)
            pred = [pred_0[0],pred_1[0],pred_2[0],pred_3[0],pred_4[0],
                    pred_5[0],pred_6[0],pred_7[0],pred_8[0],pred_9[0]]
            value = JS.predict_c(pred)
            y_pred.append(value)
```

```
In [0]: # calculate accuracy
        num_test = len(y_test)
        num_correct = np.sum(y_pred == y_test)
        print('Got %d / %d correct' % (num_correct, num_test))
        print('Accuracy = %f' % (np.mean(y_test == y_pred)))
```

```
Got 8775 / 10000 correct
Accuracy = 0.877500
```

```
In [0]: print(confusion_matrix(y_test, y_pred))
```

```
        print(classification_report(y_test, y_pred,
                                    target_names=list(label_dict.values()),digits=3))
```

```
[[ 932    0    2    5    1   13   17    1    8    1]
 [   0 1098    5    2    0    3    4    3   20    0]
 [  11   15  877   22   18    9   15    9   53    3]
 [  13    2   36  854    1   45   11   10   24   14]
 [   3    4   16    1  888    4   14    8    9   35]
 [  19    6   11   59   22  674   23   12   53   13]
 [  20    3    6    2   13   22  879    3   10    0]
 [   2   14   23    7   13    4    0  915    6   44]
 [  15   11   17   27   16   32   14   14  818   10]
 [   8    7    1   14   52    9    4   58   16  840]]
              precision    recall  f1-score   support

           0      0.911     0.951     0.931       980
           1      0.947     0.967     0.957      1135
           2      0.882     0.850     0.866      1032
           3      0.860     0.846     0.853      1010
           4      0.867     0.904     0.885       982
           5      0.827     0.756     0.790       892
           6      0.896     0.918     0.907       958
           7      0.886     0.890     0.888      1028
           8      0.804     0.840     0.822       974
           9      0.875     0.833     0.853      1009

   micro avg      0.877     0.877     0.877     10000
   macro avg      0.876     0.875     0.875     10000
weighted avg      0.877     0.877     0.877     10000
```

7 Random Forest

```
In [0]: # download MNIST dataset from keras
        (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
        # convert data type to float 32
        x_train=np.float32(x_train)[0:20000]
        y_train=y_train[0:20000]
        x_test=np.float32(x_test)
        x_train = x_train.reshape(np.shape(x_train)[0], 28*28)
        x_test = x_test.reshape(np.shape(x_test)[0], 28*28)
```

```
In [0]: # Feature Scaling
        scaler = StandardScaler()
        x_train = scaler.fit_transform(x_train)
        x_test = scaler.transform(x_test)
```

```
In [0]: rf = RandomForestClassifier(n_estimators = 200,criterion = 'entropy')

In [0]: # Train the model on training data
        rf.fit(x_train, y_train);

In [30]: from sklearn import metrics
         # Use the forest's predict method on the test data
         y_pred = rf.predict(x_test)

         # Print out the mean square error (mse)
         print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))

Mean Squared Error: 0.7315


In [31]: # calculate accuracy
         acc=rf.score(x_test, y_test)
         num_test = len(y_test)
         num_correct = np.sum(y_pred == y_test)
         print('Got %d / %d correct' % (num_correct, num_test))
         print('Accuracy = %f' % (acc))

Got 9608 / 10000 correct
Accuracy = 0.960800


In [25]: md = [3,7,10,13,16,20]
         for i in range(len(md)):
             rf1 = RandomForestClassifier(max_depth=md[i],
                                          n_estimators=200,criterion = 'entropy')
             rf1.fit(x_train, y_train)
             print("Accuracy on training set: {:.4f}".format(
                 rf1.score(x_train, y_train)))
             print("Accuracy on test set: {:.4f}".format(rf1.score(x_test, y_test)))

Accuracy on training set: 0.7432
Accuracy on test set: 0.7471
Accuracy on training set: 0.9260
Accuracy on test set: 0.9168
Accuracy on training set: 0.9841
Accuracy on test set: 0.9469
Accuracy on training set: 0.9988
Accuracy on test set: 0.9580
Accuracy on training set: 0.9999
Accuracy on test set: 0.9573
Accuracy on training set: 1.0000
Accuracy on test set: 0.9593
```

```
In [0]: rf2 = RandomForestClassifier(max_depth=20, n_estimators=200,criterion = 'entropy')
        rf2.fit(x_train, y_train)
        y_pred = rf2.predict(x_test)
        acc=rf2.score(x_test, y_test)
        print('Accuracy = %f' % (acc))
```

```
Accuracy = 0.960000
```

```
In [32]: print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, y_pred,
                                     target_names=list(label_dict.values()),digits=3))
```

```
[[ 968    0    0    0    0    4    4    1    3    0]
 [   0 1122    3    3    0    2    2    1    2    0]
 [   6    0  987   10    3    0    7   11    7    1]
 [   1    0   11  968    0   10    0    9    7    4]
 [   1    0    2    0  944    0    8    0    3   24]
 [   5    1    0   22    3  843    8    2    6    2]
 [  11    3    0    0    7    4  931    0    2    0]
 [   1    6   24    1    3    0    0  979    4   10]
 [   5    1    6    6    5    6    9    5  917   14]
 [   5    6    3   14   15    4    2    3    8  949]]
              precision    recall  f1-score   support

           0      0.965     0.988     0.976       980
           1      0.985     0.989     0.987      1135
           2      0.953     0.956     0.955      1032
           3      0.945     0.958     0.952      1010
           4      0.963     0.961     0.962       982
           5      0.966     0.945     0.955       892
           6      0.959     0.972     0.965       958
           7      0.968     0.952     0.960      1028
           8      0.956     0.941     0.949       974
           9      0.945     0.941     0.943      1009

   micro avg      0.961     0.961     0.961     10000
   macro avg      0.961     0.960     0.960     10000
weighted avg      0.961     0.961     0.961     10000
```

8 PCA + Random Forest

8.1 PCA Class

```
In [0]: # download MNIST dataset from keras
        (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

```
        # convert data type to float 32
        x_train=np.float32(x_train)
        x_test=np.float32(x_test)
        x_train = x_train.reshape(np.shape(x_train)[0], 28*28)/255.0
        x_test = x_test.reshape(np.shape(x_test)[0], 28*28)/255.0

In [0]: # stack them as a big one for dimension deduction
        big_X=np.vstack((x_train,x_test))

In [0]: # build PCA class
        class SJPCA(object):
            def __init__(self):
                pass

            def train(self, X):
                self.x_train = X

            def compute_mean_covar_eigen(self):
                # get average image and get mean image by summing each row
                tr_mean = np.mean(self.x_train, axis=0)
                tr_mean = np.reshape(tr_mean,(1,np.shape(tr_mean)[0]))

                # subtract the mean
                xtr_m = self.x_train - tr_mean

                # calculate covariance matrix
                tr_cov = np.dot(xtr_m.T,xtr_m)

                # get eigenvalue and eigenvector
                tr_val, tr_vec = eig(tr_cov)

                return xtr_m, tr_cov, tr_val, tr_vec

            def get_comp_K(self,tr_val, threshold):
                cum_lambda = np.cumsum(tr_val)
                total_lamda = cum_lambda[-1]

                # get the principal component number that we want to keep
                for keep_dim in range(len(tr_val)):
                    rate = cum_lambda[keep_dim]/total_lamda
                    if rate >= threshold:
                        return keep_dim
                        break
                    else: continue

            def deduct_img(self, tr_vec, keep_dim):
                x_proj= np.dot(self.x_train, tr_vec.T[:,0:keep_dim])
                return x_proj
```

8.2 Retain 90% Eigenvalues

```
In [8]: # Deduct Training Set
        SJ = SJPCA()
        SJ.train(big_X)
        xtr_m, tr_cov, tr_val, tr_vec = SJ.compute_mean_covar_eigen()
        keep_dim = SJ.get_comp_K(tr_val, 0.90)
        new_big_X = SJ.deduct_img(tr_vec, keep_dim)
        print(keep_dim)
```

```
86
```

```
In [0]: # resplit the dataset and normalize them with min-max normalization
        x_train = new_big_X[0:60000,:]
        x_test = new_big_X[60000:70000,:]
        scaler = StandardScaler()
        x_train = scaler.fit_transform(x_train)
        x_test = scaler.transform(x_test)
```

```
In [10]: rf = RandomForestClassifier(n_estimators = 200,criterion = 'entropy')
         # Train the model on training data
         rf.fit(x_train[0:20000,], y_train[0:20000,])
         from sklearn import metrics
         # Use the forest's predict method on the test data
         y_pred = rf.predict(x_test[0:20000,])

         # Print out the mean square error (mse)
         print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
```

```
Mean Squared Error: 3.2954
```

```
In [11]:  # calculate accuracy
         acc=rf.score(x_test, y_test)
         num_test = len(y_test)
         num_correct = np.sum(y_pred == y_test)
         print('Got %d / %d correct' % (num_correct, num_test))
         print('Accuracy = %f' % (acc))
```

```
Got 7912 / 10000 correct
Accuracy = 0.791200
```

```
In [12]: md = [3,7,10,13,16,20]
         for i in range(len(md)):
             rf1 = RandomForestClassifier(max_depth=md[i],
                                          n_estimators=200,criterion = 'entropy')
             rf1.fit(x_train[0:20000,], y_train[0:20000,])
```

```
            print("Accuracy on training set: {:.3f}".format(rf1.score(
                  x_train[0:20000,], y_train[0:20000,])))
            print("Accuracy on test set: {:.3f}".format(rf1.score(x_test, y_test)))
```

```
Accuracy on training set: 0.506
Accuracy on test set: 0.503
Accuracy on training set: 0.700
Accuracy on test set: 0.670
Accuracy on training set: 0.859
Accuracy on test set: 0.737
Accuracy on training set: 0.979
Accuracy on test set: 0.773
Accuracy on training set: 0.998
Accuracy on test set: 0.787
Accuracy on training set: 1.000
Accuracy on test set: 0.788
```

```
In [0]: print(confusion_matrix(y_test, y_pred))
        print(classification_report(y_test, y_pred,
                                    target_names=list(label_dict.values()),digits=3))
```

```
[[ 848    0   16   13    2   12   56    0   33    0]
 [   0 1089   13    2    4    0   10    1   16    0]
 [  34    9  817   24    5    6   67    4   66    0]
 [  27    1   36  778   13   57   27   12   49   10]
 [   0   20   19    3  819    1   16   18    5   81]
 [  33    5   33  241   18  439   30    6   78    9]
 [  20    4   37    0   15   11  846    0   25    0]
 [   0   26    4    2   54    0    5  818   10  109]
 [  62   14   16   38    7   24   21    9  777    6]
 [   4    6   10   17  113    9    0  170    6  674]]
              precision    recall  f1-score   support

           0      0.825     0.865     0.845       980
           1      0.928     0.959     0.943      1135
           2      0.816     0.792     0.804      1032
           3      0.696     0.770     0.731      1010
           4      0.780     0.834     0.806       982
           5      0.785     0.492     0.605       892
           6      0.785     0.883     0.831       958
           7      0.788     0.796     0.792      1028
           8      0.730     0.798     0.762       974
           9      0.758     0.668     0.710      1009

   micro avg      0.790     0.790     0.790     10000
   macro avg      0.789     0.786     0.783     10000
weighted avg      0.791     0.790     0.787     10000
```

8.3 Retain 95% Eigenvalues

```
In [10]: # Deduct Training Set
         SJ = SJPCA()
         SJ.train(big_X)
         xtr_m, tr_cov, tr_val, tr_vec = SJ.compute_mean_covar_eigen()
         keep_dim = SJ.get_comp_K(tr_val, 0.95)
         new_big_X = SJ.deduct_img(tr_vec, keep_dim)
         print(keep_dim)
```

```
153
```

```
In [0]: # resplit the dataset and normalize them with min-max normalization
        x_train = new_big_X[0:60000,:]
        x_test = new_big_X[60000:70000,:]

        scaler = StandardScaler()
        x_train = scaler.fit_transform(x_train)
        x_test = scaler.transform(x_test)
```

```
In [12]: rf = RandomForestClassifier(n_estimators = 200,criterion = 'entropy')
         # Train the model on training data
         rf.fit(x_train[0:20000,], y_train[0:20000,])
         from sklearn import metrics
         # Use the forest's predict method on the test data
         y_pred = rf.predict(x_test[0:20000,])

         # Print out the mean square error (mse)
         print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
```

```
Mean Squared Error: 1.7849
```

```
In [14]:  # calculate accuracy
          acc=rf.score(x_test, y_test)
          num_test = len(y_test)
          num_correct = np.sum(y_pred == y_test)
          print('Got %d / %d correct' % (num_correct, num_test))
          print('Accuracy = %f' % (acc))
```

```
Got 8928 / 10000 correct
Accuracy = 0.892800
```

```
In [16]: md = [3,7,10,13,16,20]
         for i in range(len(md)):
             rf1 = RandomForestClassifier(max_depth=md[i],
                                          n_estimators=200,criterion = 'entropy')
             rf1.fit(x_train[0:20000,], y_train[0:20000,])
             print("Accuracy on training set: {:.3f}".format(rf1.score(
                  x_train[0:20000,], y_train[0:20000,])))
             print("Accuracy on test set: {:.3f}".format(rf1.score(x_test, y_test)))
```

```
Accuracy on training set: 0.628
Accuracy on test set: 0.621
Accuracy on training set: 0.822
Accuracy on test set: 0.798
Accuracy on training set: 0.956
Accuracy on test set: 0.864
Accuracy on training set: 0.999
Accuracy on test set: 0.885
Accuracy on training set: 1.000
Accuracy on test set: 0.892
Accuracy on training set: 1.000
Accuracy on test set: 0.891
```

```
In [15]: print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, y_pred,
                                     target_names=list(label_dict.values()),digits=3))
```

```
[[ 908    0    7    3    1    5   44    1   11    0]
 [   0 1111    5    3    2    1    5    0    8    0]
 [  11    3  923   19   10    1   27    7   31    0]
 [  12    0   21  884    5   37   19   11   16    5]
 [   0    4   10    0  899    1    8   10    4   46]
 [  25    3    6   73   18  688   26    5   34   14]
 [  21    1   15    1   12    7  879    1   21    0]
 [   0   12   12    0   32    0    3  912   10   47]
 [  18    2   12   21    6   30   16    5  857    7]
 [   4    7    2   21   73    4    1   27    3  867]]
              precision    recall  f1-score   support

           0      0.909     0.927     0.918       980
           1      0.972     0.979     0.975      1135
           2      0.911     0.894     0.903      1032
           3      0.862     0.875     0.869      1010
           4      0.850     0.915     0.881       982
           5      0.889     0.771     0.826       892
           6      0.855     0.918     0.885       958
           7      0.932     0.887     0.909      1028
           8      0.861     0.880     0.870       974
```

9	0.879	0.859	0.869	1009
micro avg	0.893	0.893	0.893	10000
macro avg	0.892	0.891	0.891	10000
weighted avg	0.894	0.893	0.892	10000