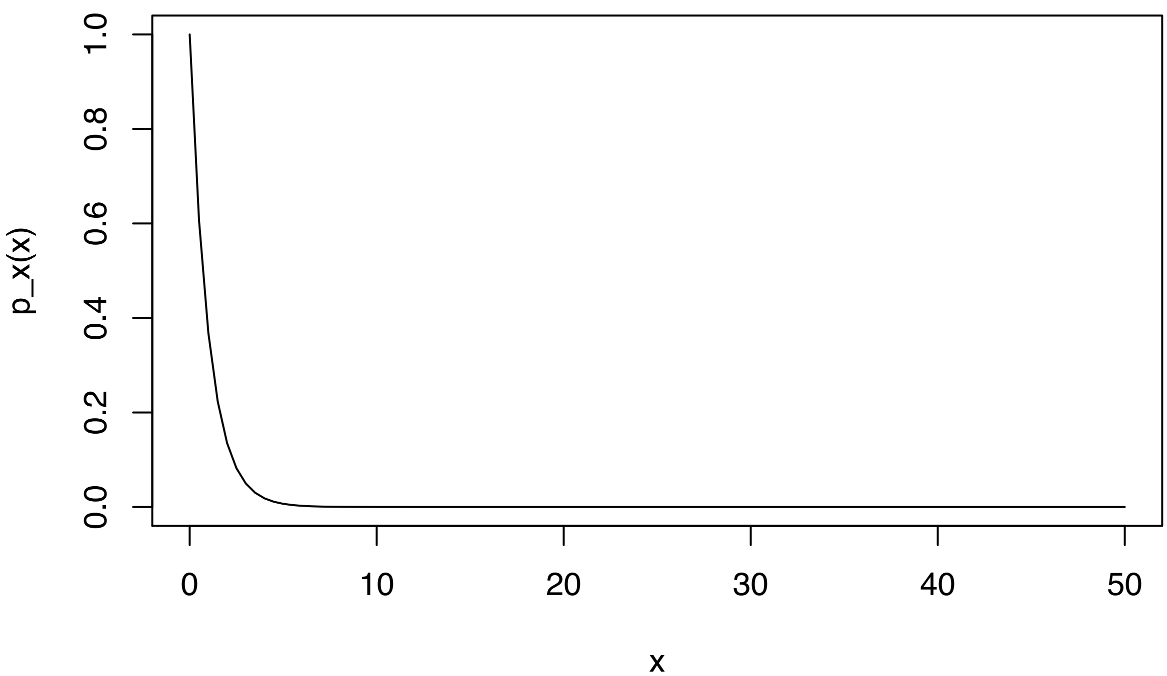
Advanced Pattern Recognition Homework 2

Name	DUID
Jian Sun	873397832

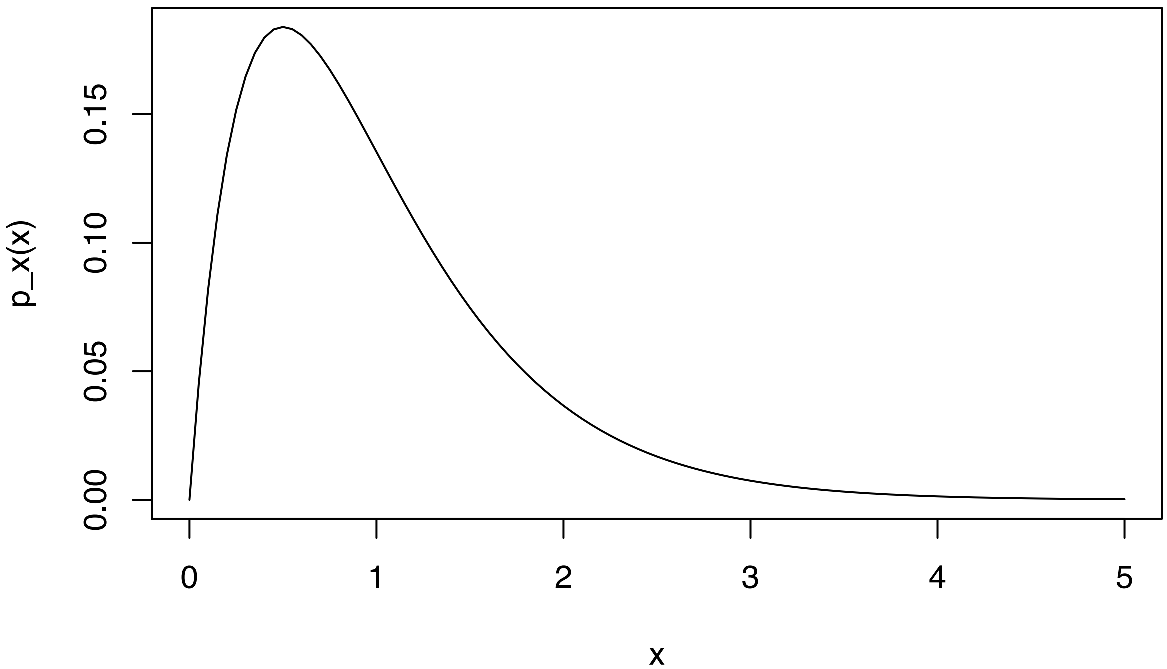
1.a
$p(x|\theta) = \theta e^{-\theta x}$, when $x \geq 0$; otherwise $p(x|\theta) = 0$.
When $\theta = 1$, $p(x|\theta = 1) = e^{-x}$, when $x \geq 0$; otherwise $p(x|\theta) = 0$. Then we plot it out.

```
p_x <- function(t) exp(-t)
curve(p_x, from = 0, to = 50)
```



When x = 2, $p(x = 2|\theta) = \theta e^{-2\theta}$.

```
p_x <- function(t) t*exp(-2*t)
curve(p_x, from = 0, to = 5)
```



1.b
$p(D|\theta) = \prod_{k=1}^{n} p(x_k|\theta) = \theta^n e^{-\theta \sum_{k=1}^{n} x_k}$

$ln(p(D|\theta)) = nln(\theta) - \theta \sum_{k=1}^{n} x_k$

$\frac{\partial ln(p(D|\theta))}{\partial \theta} = \frac{n}{\theta} - \sum_{k=1}^{n} x_k$.

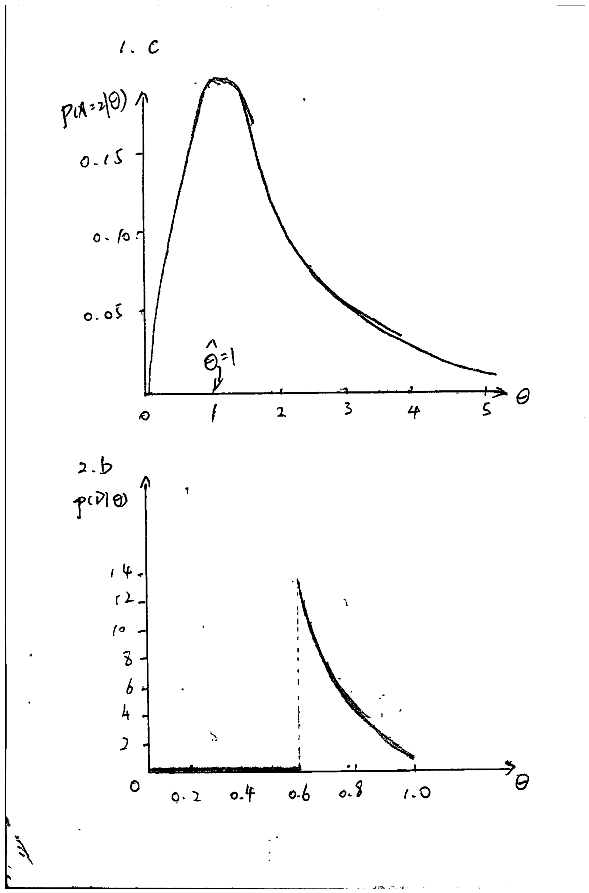
Let $\frac{\partial ln(p(D|\theta))}{\partial \theta} = 0$, then we get

$\frac{n}{\theta} - \sum_{k=1}^{n} x_k = 0$,

$\theta = \frac{1}{\frac{1}{n}\sum_{k=1}^{n} x_k}$.

So the maximum likelihood estimate for θ is given by $\hat{\theta} = \frac{1}{\frac{1}{n}\sum_{k=1}^{n} x_k}$.

1.c
The graph a is generated when $\theta = 1$, so 1 is its maximum likelihood estimate. We will mark $\hat{\theta}$ as 1 at the x_axis.



2.a
$p(x|\theta) \sim U(0, \theta) = 1/\theta$, when $0 \leq x \leq \theta$; otherwise $U(0, \theta) = 0$.

$p(D|\theta) = \prod_{k=1}^{n} p(x_k|\theta) = (1/\theta)^n$

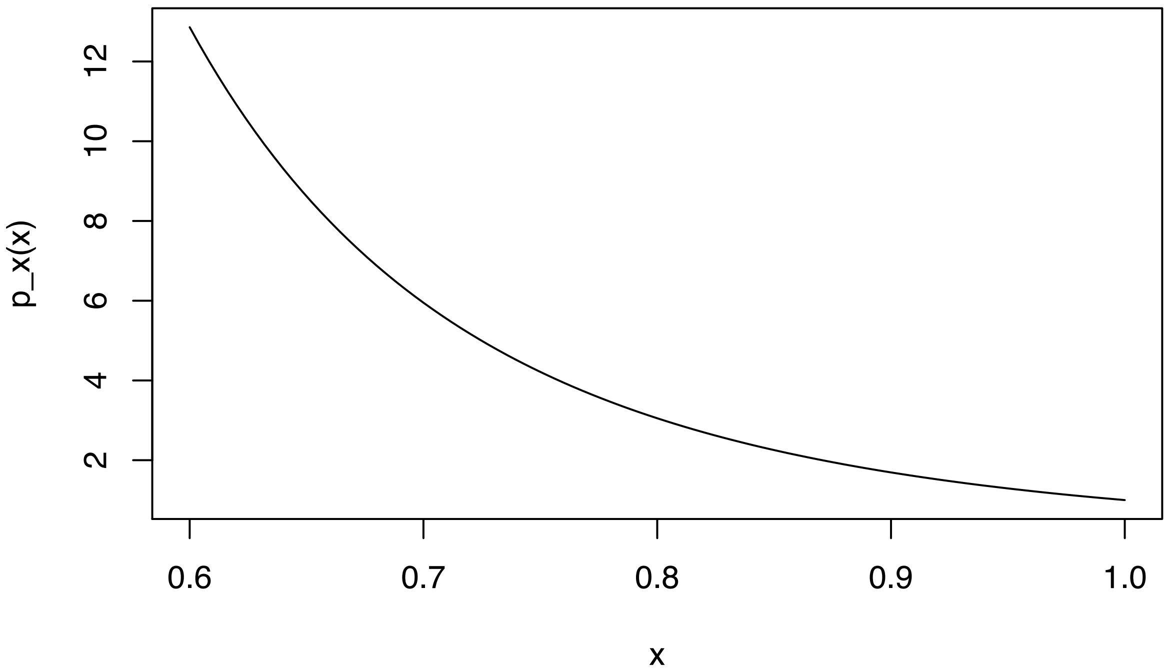
$ln(p(D|\theta)) = -nln(\theta)$

$\frac{\partial ln(p(D|\theta))}{\partial \theta} = -n/\theta,$

which means that $p(D|\theta)$ is a decreasing function. The value will decrease as θ increases. But I don't know how to solve the problem from here. So let's go back to initial condition, $0 \leq x \leq \theta$. So $\theta \geq x_{max}$, otherwise $p(x|\theta) = 0$, and 0 is less than any positive number. To keep $p(x|\theta)$ as positive number, we need to keep $\theta \geq x_{max}$, hence the maximum likelihood estimate for θ is $x_{max} = max[D]$.

2.b
Given that n = 5, $p(D|\theta) = (1/\theta)^5$. Meanwhile, $max_k x_k = 0.6$, $0 \leq \theta \leq 1$, so we have $p(D|\theta) = (1/\theta)^5$, when $0.6 \leq \theta \leq 1$; otherwise $p(D|\theta) = 0$, when $0 \leq \theta < 0.6$.
Next, we plot $p(D|\theta)$ out. By the way the predict variable is θ, not x

```
p_x <- function(t) {return((1/t)^5)
  }
curve(p_x, from = 0.6, to = 1)
```



For $0 \le \theta < 0.6$, $p(D|\theta)$ will be constant, 0.

When we know the max value of x, we know the scape of $p(D|\theta)$, and this is what we want to plot, which has no relationship with x, so we don't need to know the other four points.

4

Here, $P(D|x) = \prod_{k=1}^{n} p(x_k|\theta) = \prod_{k=1}^{n} \prod_{i=1}^{d} \theta_i^{x_{ki}} (1-\theta_i)^{1-x_{ki}}$

$P(D|x) = \prod_{i=1}^{d} \theta_i^{\sum_{k=1}^{n} x_{ki}} (1-\theta_i)^{\sum_{k=1}^{n} 1-x_{ki}}$

$ln(P(D|x)) = ln(\prod_{i=1}^{d} \theta_i^{\sum_{k=1}^{n} x_{ki}} (1-\theta_i)^{\sum_{k=1}^{n} 1-x_{ki}}) = \sum_{i=1}^{d} ln(\theta_i^{\sum_{k=1}^{n} x_{ki}} (1-\theta_i)^{\sum_{k=1}^{n} 1-x_{ki}})$
$ln(P(D|x)) = \sum_{k=1}^{n} x_{ki} \sum_{i=1}^{d} ln(\theta_i) + \sum_{k=1}^{n} (1-x_{ki}) \sum_{i=1}^{d} ln(1-\theta_i)$

For every i here, we have
$\frac{\partial ln(p(D|\theta))}{\partial \theta_i} = (\sum_{k=1}^{n} x_{ki})/\theta_i - (\sum_{k=1}^{n} (1-x_{ki}))/(1-\theta_i) = 0$
$(1-\theta_i)\sum_{k=1}^{n} x_{ki} = \theta_i(\sum_{k=1}^{n}(1-x_{ki}))$
$\sum_{k=1}^{n} x_{ki} = \theta_i \sum_{k=1}^{n} 1$
$\theta_i = \sum_{k=1}^{n} x_{ki}/n.$

Therefore, for each θ_i, the maximum likelihood estimate is $\sum_{k=1}^{n} x_{ki}/n$. Generally, the maximum likelihood estimate for θ is $\hat{\theta} = \sum_{k=1}^{n} x_k/n$.

HW2 comprehen

April 20, 2019

1 Computer Exercise

In this part, we code Bayesian Decision Rule and K-Nearest Neighbor to recognize the handwritten digit from MNIST.
For Bayesian Decision Rule, we choose 2 methods to deal with it. One is purely utilizing Bayesian Decision Rule, the other one is using Principal Component Analysis to deduct dimension firstly, then using Bayesian Decision Rule with new dataset.
In general, the KNN performs best, then Bayesian Decision Rule ranks number 2, PCA + Bayes ranks third. But based on theoritical knowledge, PCA + Bayes should be better than purely Bayes, so PCA + Bayes has huge potential, which can be our future work.

1.1 Bayesian Decision Rule

The accuracy is around 60%. The micro average precision is 0.590, the macro average precision is 0.694, the weighted avg is 0.690.
The following is the detailed information.

```
In [12]: 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 5896 / 10000 correct
Accuracy = 0.589600
```

```
In [13]: dict_characters = {0: '0', 1: '1', 2: '2', 3: '3', 4: '4',
                            5: '5', 6: '6', 7: '7', 8: '8', 9: '9'}
         print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, y_pred,
                                     target_names=list(dict_characters.values()),digits=3))
```

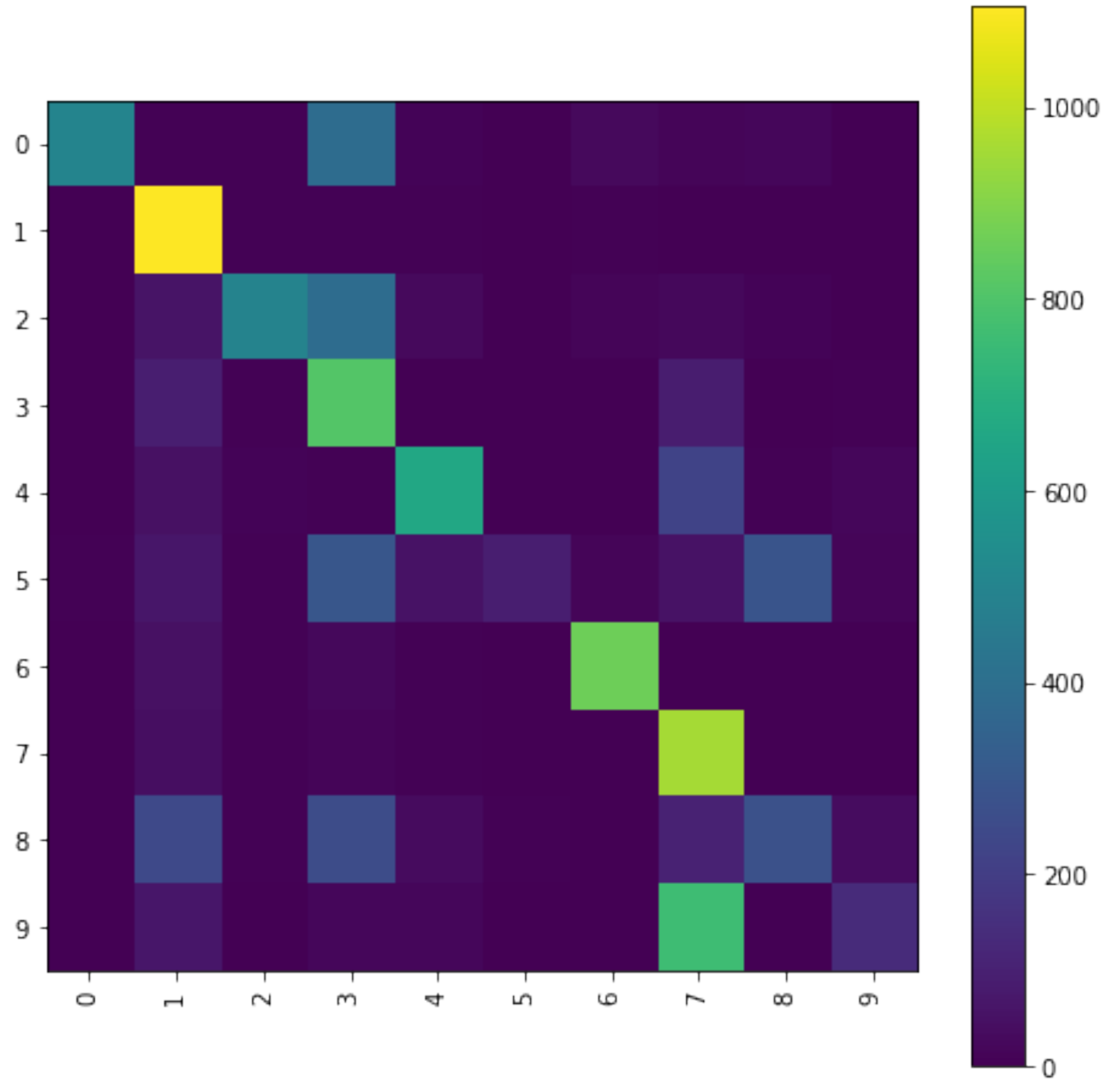
```
[[ 499    6    5  391   10    1   30   15   21    2]
 [    0 1106    5    8    8    0    5    2    0    1]
 [    3   60  496  393   27    0   16   23   10    4]
 [    1   95    6  809    3    2    0   88    1    5]
 [    0   50   12    5  664    1    3  221    7   19]
```

```
 [   5   65    5  297   52   93   16   56  287   16]
 [   4   49    5   23    7    3  863    1    3    0]
 [   0   43    5   13    7    0    0  955    1    4]
 [   2  245    5  259   33    6    4  107  275   38]
 [   1   65    4   20   19    0    0  764    0  136]]
             precision    recall  f1-score   support

           0      0.969     0.509     0.668       980
           1      0.620     0.974     0.758      1135
           2      0.905     0.481     0.628      1032
           3      0.365     0.801     0.501      1010
           4      0.800     0.676     0.733       982
           5      0.877     0.104     0.186       892
           6      0.921     0.901     0.911       958
           7      0.428     0.929     0.586      1028
           8      0.455     0.282     0.348       974
           9      0.604     0.135     0.220      1009

   micro avg      0.590     0.590     0.590     10000
   macro avg      0.694     0.579     0.554     10000
weighted avg      0.690     0.590     0.559     10000
```

```
In [14]: plt.figure(figsize=(8,8))
         cnf_matrix = confusion_matrix(y_test, y_pred)
         classes = list(dict_characters.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)
```



1.2 Principal Component Analysis + Bayesian Decision Rule

The accuracy is around 40.8%. The micro average precision is 0.408, the macro average precision is 0.507, the weighted avg is 0.512. And this result is not constant, sometimes the macro average precision is 0.595.
The following is the detailed information.

```
In [62]: 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 4077 / 10000 correct
```

```
Accuracy = 0.407700
```

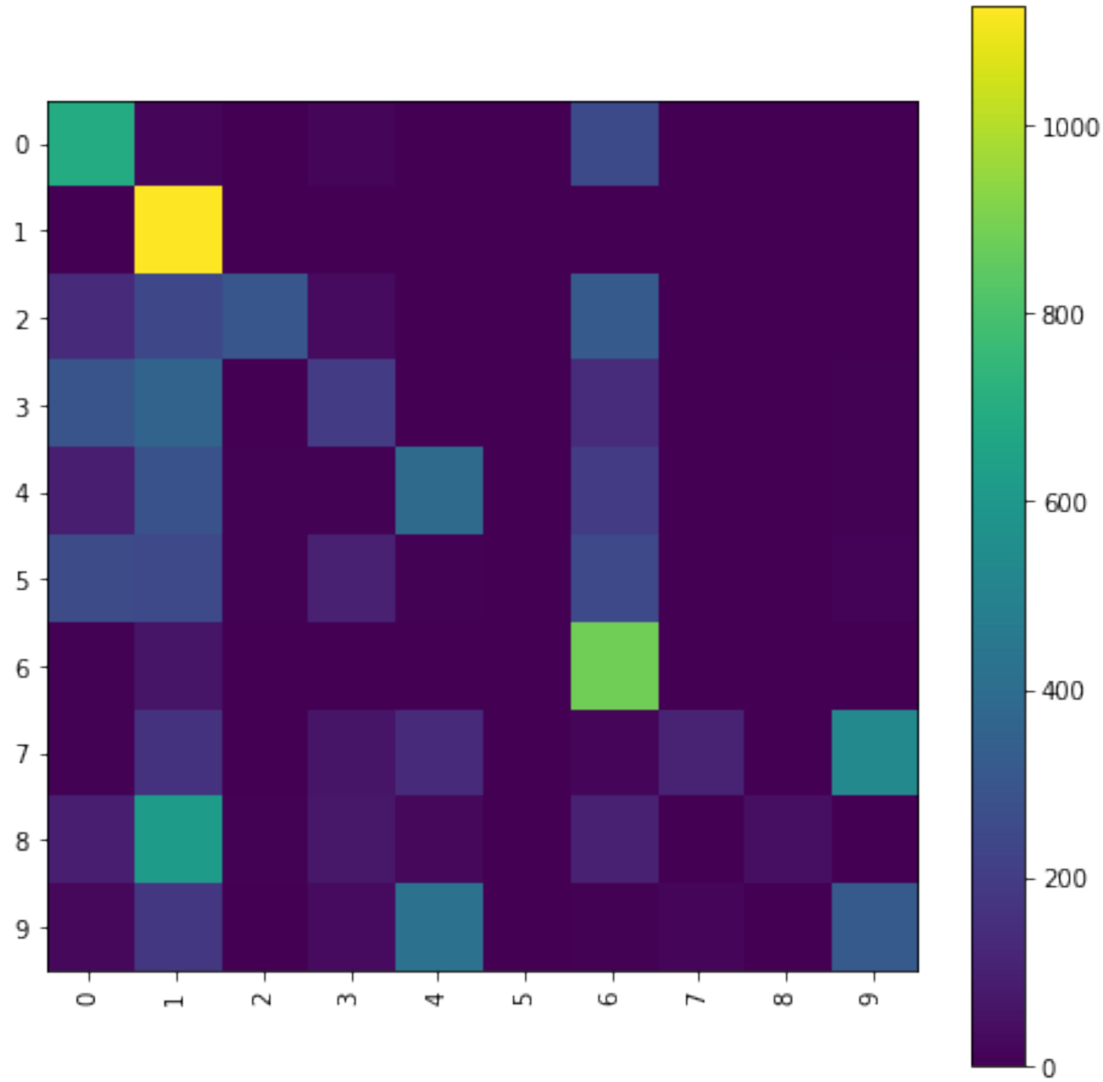
```
In [63]: dict_characters = {0: '0', 1: '1', 2: '2', 3: '3', 4: '4',
                            5: '5', 6: '6', 7: '7', 8: '8', 9: '9'}
         print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, y_pred,
                                     target_names=list(dict_characters.values()),digits=3))
```

```
[[ 694   18    0   14    0    0  254    0    0    0]
 [    1 1128    2    0    1    0    3    0    0    0]
 [ 133  244  302   31    4    0  318    0    0    0]
 [ 293  359    2  201    4    0  145    1    0    5]
 [  96  284    5    6  392    0  194    0    0    5]
 [ 263  251    6  105    5    0  249    1    2   10]
 [    8   65    2    0    0    0  883    0    0    0]
 [    8  164    1   64  137    0   15  110    0  529]
 [  95  623    5   73   26    0  102    0   47    3]
 [  25  184    3   34  424    0    5   14    0  320]]
              precision    recall  f1-score   support

           0      0.429     0.708     0.535       980
           1      0.340     0.994     0.506      1135
           2      0.921     0.293     0.444      1032
           3      0.381     0.199     0.261      1010
           4      0.395     0.399     0.397       982
           5      0.000     0.000     0.000       892
           6      0.407     0.922     0.565       958
           7      0.873     0.107     0.191      1028
           8      0.959     0.048     0.092       974
           9      0.367     0.317     0.340      1009

   micro avg      0.408     0.408     0.408     10000
   macro avg      0.507     0.399     0.333     10000
weighted avg      0.512     0.408     0.338     10000
```

```
In [64]: plt.figure(figsize=(8,8))
         cnf_matrix = sklearn.metrics.confusion_matrix(y_test, y_pred)
         classes = list(dict_characters.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)
```



1.3 K-Nearest Neighbor

There are 3 K waiting for selection, K=1, K=3 and K=5. And we find that
k = 1, Accuracy = 0.969100
k = 3, Accuracy = 0.971700
k = 5, Accuracy = 0.969300.
Hence, we set K as 3, since it will contribute to highest accuracy. Next, by running the code, the accuracy is 97%. The micro average precision is 0.972, the macro average precision is 0.972, the weighted avg is 0.972. The following is the detailed information.

```
In [34]: num_test = len(y_test)
         num_correct = np.sum(Y_test_pred == y_test)
         print('Got %d / %d correct' % (num_correct, num_test))
         print('Accuracy = %f' % (np.mean(y_test == Y_test_pred)))
```

```
Got 9717 / 10000 correct
Accuracy = 0.971700
```

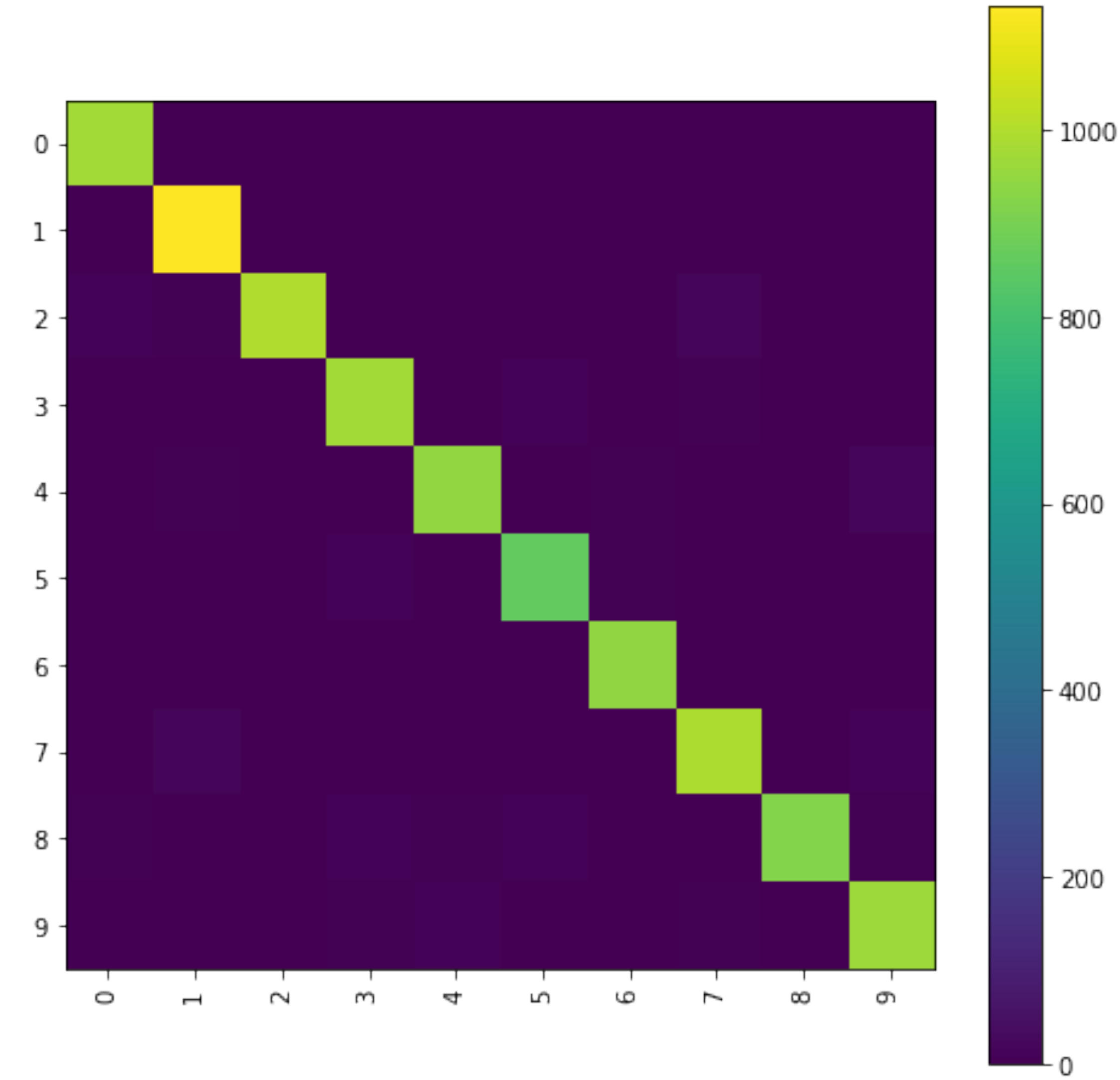
```
In [35]: dict_characters = {0: '0', 1: '1', 2: '2', 3: '3', 4: '4',
                            5: '5', 6: '6', 7: '7', 8: '8', 9: '9'}
         print(confusion_matrix(y_test, Y_test_pred))
         print(classification_report(y_test, Y_test_pred,
                                     target_names=list(dict_characters.values()),digits=3))
```

```
[[ 974    1    1    0    0    1    2    1    0    0]
 [   0 1133    2    0    0    0    0    0    0    0]
 [   9    7  997    2    0    0    1   14    2    0]
 [   0    1    4  975    1   13    1    7    4    4]
 [   0    5    0    0  948    0    5    4    1   19]
 [   4    1    0   12    2  860    5    1    3    4]
 [   4    3    0    0    4    3  944    0    0    0]
 [   0   18    4    0    2    0    0  994    0   10]
 [   7    0    3   13    5   11    3    4  923    5]
 [   3    4    2    7    9    4    1    8    2  969]]
              precision    recall  f1-score   support

           0      0.973     0.994     0.983       980
           1      0.966     0.998     0.982      1135
           2      0.984     0.966     0.975      1032
           3      0.966     0.965     0.966      1010
           4      0.976     0.965     0.971       982
           5      0.964     0.964     0.964       892
           6      0.981     0.985     0.983       958
           7      0.962     0.967     0.965      1028
           8      0.987     0.948     0.967       974
           9      0.958     0.960     0.959      1009

   micro avg      0.972     0.972     0.972     10000
   macro avg      0.972     0.971     0.972     10000
weighted avg      0.972     0.972     0.972     10000
```

```
In [37]: plt.figure(figsize=(8,8))
         cnf_matrix = confusion_matrix(y_test, Y_test_pred)
         classes = list(dict_characters.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)
```



2 Appendix

```
In [1]: # import dataset and seperate them as train set and test set
        # index x represents image, index y represents label
        import os
        import cv2
        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.metrics import classification_report, confusion_matrix
```

In [2]:
```
# 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)
```

2.1 Bayes Part

In [3]:
```
# add noise to images
def add_noisy(image):
    ch,row,col = np.shape(image)
    mean = 0
    var = 0.01
    sigma = var**0.5
    gauss = np.random.normal(mean,sigma,(ch,row,col))
    gauss = np.reshape(gauss,(ch,row,col))
    noisy = image + gauss
    return noisy
```

In [4]:
```
# normalize and reconstruct the dataset
X_train = add_noisy(x_train) / 255.0
X_test = add_noisy(x_test) / 255.0
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 [6]:
```
# construct a Bayesian Decision Rule class
class SJBAYES(object):
    def __init__(self):
        pass

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

    def split_category(self, category_name):
        xx_train=[]
        yy_train=[]
        for i in range(len(y_train)):
            if (self.y_train[i]==category_name):
                xx_train.append(self.x_train[i])
                yy_train.append(self.y_train[i])
        return xx_train, yy_train

    def MLE_miu_sigma(self, img_col, data):
        wait_mean = np.reshape(data,(len(data),img_col))
        cate_miu = np.mean(wait_mean, axis=0)
```

```
            cm=np.reshape(cate_miu,(1,img_col))
            b=data-cm
            a=np.transpose(b)
            sgm=np.dot(a,b)
            return cate_miu, sgm

        def para_for_case3(self, cate_miu, sgm, data):
            cm=np.reshape(cate_miu,(1,img_col))
            W=-0.5*pinv(sgm)
            w=np.transpose(np.dot(pinv(sgm),np.transpose(cm)))
            P_w=len(data)/len(self.x_train)
            #det_=np.exp(np.trace(np.log(sgm)))
            det_=np.trace(sgm)
            #det_=det(sgm)
            sigdet=-0.5*np.log(det_)
            msm=np.dot(np.dot(cm,pinv(sgm)),np.transpose(cm))
            www=-0.5*msm[0][0]+sigdet+np.log(P_w)
            #print(www)
            return W, w, www

        def discri_fun(self, img_col, x_test, W, w, www):
            x_test = np.reshape(x_test,(1,img_col))
            g=np.dot(np.dot(x_test,W),x_test.T)+np.dot(w,x_test.T)+www
            return g
```

In [7]:
```
# split the train as 10 categories
JS = SJBAYES()
JS.train(x_train,y_train)
x0_train, y0_train = JS.split_category(0)
x1_train, y1_train = JS.split_category(1)
x2_train, y2_train = JS.split_category(2)
x3_train, y3_train = JS.split_category(3)
x4_train, y4_train = JS.split_category(4)
x5_train, y5_train = JS.split_category(5)
x6_train, y6_train = JS.split_category(6)
x7_train, y7_train = JS.split_category(7)
x8_train, y8_train = JS.split_category(8)
x9_train, y9_train = JS.split_category(9)

# get mean and variance matrix for training set
img_col = 784
miu0,sig0=JS.MLE_miu_sigma(img_col, x0_train)
miu1,sig1=JS.MLE_miu_sigma(img_col, x1_train)
miu2,sig2=JS.MLE_miu_sigma(img_col, x2_train)
miu3,sig3=JS.MLE_miu_sigma(img_col, x3_train)
miu4,sig4=JS.MLE_miu_sigma(img_col, x4_train)
miu5,sig5=JS.MLE_miu_sigma(img_col, x5_train)
miu6,sig6=JS.MLE_miu_sigma(img_col, x6_train)
```

```
miu7,sig7=JS.MLE_miu_sigma(img_col, x7_train)
miu8,sig8=JS.MLE_miu_sigma(img_col, x8_train)
miu9,sig9=JS.MLE_miu_sigma(img_col, x9_train)
```

In [8]:
```
W0, w0, w00 = JS.para_for_case3(miu0, sig0, x0_train)
W1, w1, w11 = JS.para_for_case3(miu1, sig1, x1_train)
W2, w2, w22 = JS.para_for_case3(miu2, sig2, x2_train)
W3, w3, w33 = JS.para_for_case3(miu3, sig3, x3_train)
W4, w4, w44 = JS.para_for_case3(miu4, sig4, x4_train)
W5, w5, w55 = JS.para_for_case3(miu5, sig5, x5_train)
W6, w6, w66 = JS.para_for_case3(miu6, sig6, x6_train)
W7, w7, w77 = JS.para_for_case3(miu7, sig7, x7_train)
W8, w8, w88 = JS.para_for_case3(miu8, sig8, x8_train)
W9, w9, w99 = JS.para_for_case3(miu9, sig9, x9_train)
```

In [9]:
```
# calculate discriminant function and predict
y_pred=[]
x_test=np.reshape(x_test,(10000,img_col))
for i in range(len(x_test)):
    g0=JS.discri_fun(img_col, x_test[i], W0, w0, w00)
    g1=JS.discri_fun(img_col, x_test[i], W1, w1, w11)
    g2=JS.discri_fun(img_col, x_test[i], W2, w2, w22)
    g3=JS.discri_fun(img_col, x_test[i], W3, w3, w33)
    g4=JS.discri_fun(img_col, x_test[i], W4, w4, w44)
    g5=JS.discri_fun(img_col, x_test[i], W5, w5, w55)
    g6=JS.discri_fun(img_col, x_test[i], W6, w6, w66)
    g7=JS.discri_fun(img_col, x_test[i], W7, w7, w77)
    g8=JS.discri_fun(img_col, x_test[i], W8, w8, w88)
    g9=JS.discri_fun(img_col, x_test[i], W9, w9, w99)

    g=[g0[0][0],g1[0][0],g2[0][0],g3[0][0],g4[0][0],
       g5[0][0],g6[0][0],g7[0][0],g8[0][0],g9[0][0]]
    ind=np.where(g==np.max(g))
    y_pred.append(ind[0][0])
```

In [10]:
```
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 5896 / 10000 correct
Accuracy = 0.589600
```

In [11]:
```
dict_characters = {0: '0', 1: '1', 2: '2', 3: '3', 4: '4',
                   5: '5', 6: '6', 7: '7', 8: '8', 9: '9'}
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred,
                            target_names=list(dict_characters.values()),digits=3))
```

```
[[ 499    6    5  391   10    1   30   15   21    2]
 [   0 1106    5    8    8    0    5    2    0    1]
 [   3   60  496  393   27    0   16   23   10    4]
 [   1   95    6  809    3    2    0   88    1    5]
 [   0   50   12    5  664    1    3  221    7   19]
 [   5   65    5  297   52   93   16   56  287   16]
 [   4   49    5   23    7    3  863    1    3    0]
 [   0   43    5   13    7    0    0  955    1    4]
 [   2  245    5  259   33    6    4  107  275   38]
 [   1   65    4   20   19    0    0  764    0  136]]
              precision    recall  f1-score   support

           0      0.969     0.509     0.668       980
           1      0.620     0.974     0.758      1135
           2      0.905     0.481     0.628      1032
           3      0.365     0.801     0.501      1010
           4      0.800     0.676     0.733       982
           5      0.877     0.104     0.186       892
           6      0.921     0.901     0.911       958
           7      0.428     0.929     0.586      1028
           8      0.455     0.282     0.348       974
           9      0.604     0.135     0.220      1009

   micro avg      0.590     0.590     0.590     10000
   macro avg      0.694     0.579     0.554     10000
weighted avg      0.690     0.590     0.559     10000
```

2.2 PCA + Bayes Part

In [50]:
```python
# 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)
```

In [51]:
```python
# add noise to images
def add_noisy(image):
    ch,row,col = np.shape(image)
    mean = 0
    var = 0.01
    sigma = var**0.5
    gauss = np.random.normal(mean,sigma,(ch,row,col))
    gauss = np.reshape(gauss,(ch,row,col))
    noisy = image + gauss
    return noisy
```

In [52]:
```python
# add noise and reconstruct dataset and stack them as a big one for dimension deducti
```

```
X_train = add_noisy(x_train)
X_test = add_noisy(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)
big_X=np.vstack((x_train,x_test))
```

In [53]:
```
# 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 tr_mean, 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
```

In [54]:
```
# Deduct Training Set
SJ = SJPCA()
```

```
        SJ.train(big_X)
        tr_mean, 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)
```

```
In [55]: # 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])/tr_cha[i]
```

```
In [56]: # build a Bayes class
         class SJBAYES(object):
             def __init__(self):
                 pass

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

             def split_category(self, category_name):
                 xx_train=[]
                 yy_train=[]
                 for i in range(len(y_train)):
                     if (self.y_train[i]==category_name):
                         xx_train.append(self.x_train[i])
                         yy_train.append(self.y_train[i])
                 return xx_train, yy_train

             def MLE_miu_sigma(self, img_col, data):
                 wait_mean = np.reshape(data,(len(data),img_col))
                 cate_miu = np.mean(wait_mean, axis=0)
                 cm=np.reshape(cate_miu,(1,img_col))
                 b=data-cm
                 a=np.transpose(b)
                 sgm=np.dot(a,b)
                 return cate_miu, sgm

             def para_for_case3(self, cate_miu, sgm, data):
                 cm=np.reshape(cate_miu,(1,img_col))
                 W=-0.5*inv(sgm)
                 w=np.transpose(np.dot(inv(sgm),np.transpose(cm)))
```

```
        P_w=len(data)/len(self.x_train)
        #det_=np.exp(np.trace(np.log(sgm)))
        det_=np.trace(sgm)
        sigdet=-0.5*np.log(det_)
        msm=np.dot(np.dot(cm,inv(sgm)),np.transpose(cm))
        www=-0.5*msm[0][0]+sigdet+np.log(P_w)
        return W, w, www

    def discri_fun(self, img_col, x_test, W, w, www):
        x_test = np.reshape(x_test,(1,img_col))
        g=np.dot(np.dot(x_test,W),x_test.T)+np.dot(w,x_test.T)+www
        return g
```

In [57]:
```
# split the train as 10 categories
JS = SJBAYES()
JS.train(x_train,y_train)
x0_train, y0_train = JS.split_category(0)
x1_train, y1_train = JS.split_category(1)
x2_train, y2_train = JS.split_category(2)
x3_train, y3_train = JS.split_category(3)
x4_train, y4_train = JS.split_category(4)
x5_train, y5_train = JS.split_category(5)
x6_train, y6_train = JS.split_category(6)
x7_train, y7_train = JS.split_category(7)
x8_train, y8_train = JS.split_category(8)
x9_train, y9_train = JS.split_category(9)

# get mean and variance matrix for training set
img_col = keep_dim
miu0,sig0=JS.MLE_miu_sigma(img_col, x0_train)
miu1,sig1=JS.MLE_miu_sigma(img_col, x1_train)
miu2,sig2=JS.MLE_miu_sigma(img_col, x2_train)
miu3,sig3=JS.MLE_miu_sigma(img_col, x3_train)
miu4,sig4=JS.MLE_miu_sigma(img_col, x4_train)
miu5,sig5=JS.MLE_miu_sigma(img_col, x5_train)
miu6,sig6=JS.MLE_miu_sigma(img_col, x6_train)
miu7,sig7=JS.MLE_miu_sigma(img_col, x7_train)
miu8,sig8=JS.MLE_miu_sigma(img_col, x8_train)
miu9,sig9=JS.MLE_miu_sigma(img_col, x9_train)
```

In [58]:
```
W0, w0, w00 = JS.para_for_case3(miu0, sig0, x0_train)
W1, w1, w11 = JS.para_for_case3(miu1, sig1, x1_train)
W2, w2, w22 = JS.para_for_case3(miu2, sig2, x2_train)
W3, w3, w33 = JS.para_for_case3(miu3, sig3, x3_train)
W4, w4, w44 = JS.para_for_case3(miu4, sig4, x4_train)
W5, w5, w55 = JS.para_for_case3(miu5, sig5, x5_train)
W6, w6, w66 = JS.para_for_case3(miu6, sig6, x6_train)
W7, w7, w77 = JS.para_for_case3(miu7, sig7, x7_train)
```

```
W8, w8, w88 = JS.para_for_case3(miu8, sig8, x8_train)
W9, w9, w99 = JS.para_for_case3(miu9, sig9, x9_train)
```

In [59]:
```
# calculate discriminant function
y_pred=[]
x_test=np.reshape(x_test,(10000,img_col))
for i in range(len(x_test)):
    g0=JS.discri_fun(img_col, x_test[i], W0, w0, w00)
    g1=JS.discri_fun(img_col, x_test[i], W1, w1, w11)
    g2=JS.discri_fun(img_col, x_test[i], W2, w2, w22)
    g3=JS.discri_fun(img_col, x_test[i], W3, w3, w33)
    g4=JS.discri_fun(img_col, x_test[i], W4, w4, w44)
    g5=JS.discri_fun(img_col, x_test[i], W5, w5, w55)
    g6=JS.discri_fun(img_col, x_test[i], W6, w6, w66)
    g7=JS.discri_fun(img_col, x_test[i], W7, w7, w77)
    g8=JS.discri_fun(img_col, x_test[i], W8, w8, w88)
    g9=JS.discri_fun(img_col, x_test[i], W9, w9, w99)

    g=[g0[0][0],g1[0][0],g2[0][0],g3[0][0],g4[0][0],
       g5[0][0],g6[0][0],g7[0][0],g8[0][0],g9[0][0]]
    #print(g,y_test[i])
    ind=np.where(g==np.max(g))
    y_pred.append(ind[0][0])
```

In [60]:
```
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 4077 / 10000 correct
Accuracy = 0.407700
```

In [61]:
```
dict_characters = {0: '0', 1: '1', 2: '2', 3: '3', 4: '4',
                   5: '5', 6: '6', 7: '7', 8: '8', 9: '9'}
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred,
                            target_names=list(dict_characters.values()),digits=3))
```

```
[[ 694   18    0   14    0    0  254    0    0    0]
 [   1 1128    2    0    1    0    3    0    0    0]
 [ 133  244  302   31    4    0  318    0    0    0]
 [ 293  359    2  201    4    0  145    1    0    5]
 [  96  284    5    6  392    0  194    0    0    5]
 [ 263  251    6  105    5    0  249    1    2   10]
 [   8   65    2    0    0    0  883    0    0    0]
 [   8  164    1   64  137    0   15  110    0  529]
 [  95  623    5   73   26    0  102    0   47    3]
 [  25  184    3   34  424    0    5   14    0  320]]
```

```
              precision    recall  f1-score   support

           0      0.429     0.708     0.535       980
           1      0.340     0.994     0.506      1135
           2      0.921     0.293     0.444      1032
           3      0.381     0.199     0.261      1010
           4      0.395     0.399     0.397       982
           5      0.000     0.000     0.000       892
           6      0.407     0.922     0.565       958
           7      0.873     0.107     0.191      1028
           8      0.959     0.048     0.092       974
           9      0.367     0.317     0.340      1009

   micro avg      0.408     0.408     0.408     10000
   macro avg      0.507     0.399     0.333     10000
weighted avg      0.512     0.408     0.338     10000
```

2.3 KNN Part

```
In [27]: (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)
         # Normalize the data
         X_train = x_train / 255.0
         X_test = x_test / 255.0
```

```
In [28]: # reconstruct dataset
         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 [29]: # build KNN class
         class SJKNN(object):
             def __init__(self):
                 pass

             def train(self, X, Y):
             # the nearest neighbor classifier simply remembers all the training data
                 self.X_train = X
                 self.Y_train = Y

             def compute_distances_no_loops(self, X_test):
                 num_test = np.shape(X_test)[0]
                 num_train = np.shape(self.X_train)[0]
                 dists = np.zeros((num_test, num_train))
                 dists = np.sqrt(self.getNormMatrix(X_test, num_train).T +
```

```
                            self.getNormMatrix(self.X_train, num_test) -
                            2 * np.dot(X_test, self.X_train.T))
            pass
            return(dists)

        def getNormMatrix(self, x, lines_num):
            return(np.ones((lines_num, 1)) * np.sum(np.square(x), axis = 1))

        def predict_labels(self, dists, k):
            num_test = np.shape(dists)[0]
            Y_pred = np.zeros(num_test)
            for i in range(num_test):
                closest_y = []
                kids = np.argsort(dists[i])
                closest_y = self.Y_train[kids[:k]]
                count = 0
                label = 0
                for j in closest_y:
                    tmp = 0
                    for kk in closest_y:
                        tmp += (kk == j)
                    if tmp > count:
                        count = tmp
                        label = j
                Y_pred[i] = label
            return Y_pred

        def predict(self, X_test, k):
            num_test = X_test.shape[0]
            # lets make sure that the output type matches the input type
            ypred = np.zeros(num_test, dtype = self.Y_train.dtype)
            dists = self.compute_distances_no_loops(X_test)
            return self.predict_labels(dists, k=k)
```

```
In [30]: def time_function(f, *args):
             import time
             tic = time.time()
             f(*args)
             toc = time.time()
             return toc - tic
         classifier = SJKNN()
         classifier.train(X_train, y_train)

         no_loop_time = time_function(classifier.compute_distances_no_loops, X_test)
         print('No loop version took %f seconds' % no_loop_time)
```

```
No loop version took 61.355703 seconds
```

```
In [31]: # select best k
         K=[1, 3, 5]
         classifier = SJKNN()
         classifier.train(X_train, y_train)
         num_test = len(y_test)
         for i in K:
             Y_test_pred=classifier.predict(X_test, k=i)
             num_correct = np.sum(Y_test_pred == y_test)
             print('Got %d / %d correct' % (num_correct, num_test))
             print('k = %s, Accuracy = %f' % (i, np.mean(y_test == Y_test_pred)))
```

```
Got 9691 / 10000 correct
k = 1, Accuracy = 0.969100
Got 9717 / 10000 correct
k = 3, Accuracy = 0.971700
Got 9693 / 10000 correct
k = 5, Accuracy = 0.969300
```

```
In [33]: # doing prediction
         dict_characters = {0: '0', 1: '1', 2: '2', 3: '3', 4: '4',
                            5: '5', 6: '6', 7: '7', 8: '8', 9: '9'}
         Y_test_pred=classifier.predict(X_test, k=3)
         print(confusion_matrix(y_test, Y_test_pred))
         print(classification_report(y_test, Y_test_pred,
                                     target_names=list(dict_characters.values()),digits=3))
```

```
[[ 974    1    1    0    0    1    2    1    0    0]
 [   0 1133    2    0    0    0    0    0    0    0]
 [   9    7  997    2    0    0    1   14    2    0]
 [   0    1    4  975    1   13    1    7    4    4]
 [   0    5    0    0  948    0    5    4    1   19]
 [   4    1    0   12    2  860    5    1    3    4]
 [   4    3    0    0    4    3  944    0    0    0]
 [   0   18    4    0    2    0    0  994    0   10]
 [   7    0    3   13    5   11    3    4  923    5]
 [   3    4    2    7    9    4    1    8    2  969]]
              precision    recall  f1-score   support

           0      0.973     0.994     0.983       980
           1      0.966     0.998     0.982      1135
           2      0.984     0.966     0.975      1032
           3      0.966     0.965     0.966      1010
           4      0.976     0.965     0.971       982
           5      0.964     0.964     0.964       892
           6      0.981     0.985     0.983       958
           7      0.962     0.967     0.965      1028
           8      0.987     0.948     0.967       974
```

```
           9      0.958     0.960     0.959      1009

   micro avg      0.972     0.972     0.972     10000
   macro avg      0.972     0.971     0.972     10000
weighted avg      0.972     0.972     0.972     10000
```