Advanced Pattern Recognition Homework 3

Name	DUID
Jian Sun	873397832

5.2

For i=1,...,c, $g_i(x) = w^t x + w_{i0}$.

When $x = x_1$ or $x = x_2$, we get $g_i(x_1) = w^t x_1 + w_{i0}$ and $g_i(x_2) = w^t x_2 + w_{i0}$.

For λx_1 and $(1-\lambda)x_2$, we have $g_i(\lambda x_1) = w^t \lambda x_1 + w_{i0}$ and $g_i((1-\lambda)x_2) = w^t x_2 + w_{i0}$.

$g_i(\lambda x_1) + g_i((1-\lambda)x_2) = w^t \lambda x_1 + w_{i0} + w^t x_2 + w_{i0}$

$= \lambda(w^t x_1 + w_{i0}) + (1-\lambda)(w^t x_2 + w_{i0})$
$= w^t(\lambda x_1 + (1-\lambda)x_2) + (\lambda w_{i0} + (1-\lambda)w_{i0})$
$= w^t(\lambda x_1 + (1-\lambda)x_2) + w_{i0}$
$= g_i(\lambda x_1 + (1-\lambda)x_2)$.

Since that $0 \leq \lambda \leq 1$, we get $min\{x_1, x_2\} \leq (\lambda x_1 + (1-\lambda)x_2) \leq max\{x_1, x_2\}$

Additionally, for any 2 points $x_1 \in R_i$ and $x_2 \in R_i$, we have that $(\lambda x_1 + (1-\lambda)x_2) \in R_i$.
Therefore, decision regions are convex.

5.4
(a)

Given that there is constraint $g(x) = 0$, so when x_a on the hyperplane, we have $g(x_a) = 0$.

It means that $||x - x_a||^2$ will be minimum, which 0, and let's check the formule.

The distance between point x_a and x is $|g(x_a) - g(x)|/||w|| = |0 - 0|/||w|| = 0$.

When x_a is not on the hyperplane, we view $||x - x_a||^2$ as function $f(x)$, so this problem becomes finding the minimum $||x - x_a||^2$ subject to the constraint $g(x) = 0$.

Referring to our class note, lagrange multiplier is a good helper to solve this kind of problem.
We build a new function, $L(x, \lambda) = f(x) + \lambda[g(x) - 0] = ||x - x_a||^2 + \lambda g(x)$

$= xx^t + x_a x_a^t - 2xx_a + \lambda(w^t x + w_0)$

$\frac{\partial L}{\partial \lambda} = w^t x + w_0 = g(x) = 0$

$\frac{\partial L}{\partial x} = 2x - 2x_a + \lambda w^t = 0$

$x = x_a - 0.5 * \lambda w^t \qquad (1)$

$w^t x + w0 = w^t(x_a - 0.5 * \lambda w^t) + w0 = 0$

$w^t x_a + w0 = 0.5 * \lambda w^t w^t$, so $\lambda = 2(w^t x_a + w0)/w^t w^t \qquad (2)$

Insert (2) into (1), we get $x = x_a - 0.5 * (2(w^t x_a + w0)/w^t w^t)w^t$, then

$x - x_a = x_a - 0.5 * (2(w^t x_a + w0)/w^t w^t)w^t - x_a$

$||x - x_a|| = ||x_a - 0.5 * (2(w^t x_a + w0)/w^t w^t)w^t - x_a|| = ||(w^t x_a + w0)w^t/(w^t w^t)|| = |g(x_a)|||w||/||w||^2 = |g(x_a)|/||w||$, this is the minimum $||x - x_a||$, also is the minimum $||x - x_a||^2$.

Hence we get what we want to prove, the distance from hyperplane $g(x) = w^t x + w0$ to the point x_a is $|g(x_a)|/||w||$ by minimizing $||x - x_a||^2$ subject to the constraint $g(x) = 0$.

(b)

According to 5.2.1, we notice that $x = x_p + r\frac{w}{||w||}$.

x_p is the normal projection of x (here x is x_a) onto hyperplane H, r is the desired algebraic distance.

$g(x_a) = g(x_p + r\frac{w}{||w||}) = w^t(x_p + r\frac{w}{||w||}) + w_0 = g(x_p) + w^t r\frac{w}{||w||} = g(x_p) + r||w||$

Since the hyperplance is equal to zero, $g(x) = 0$, $\therefore g(x_p) = 0$.

$w^t x_a + w_0 = 0 + r||w||$, then $r = \frac{g(x_a)}{||w||}$ (1).

Finally, inserting (1) back to $x_a = x_p + r\frac{w}{||w||}$, we get $x_a = x_p + \frac{g(x_a)}{||w||}\frac{w}{||w||} = x_p + \frac{g(x_a)w}{||w||^2}$.

$x_p = x_a - \frac{g(x_a)w}{||w||^2}$, we prove that this is the projection of x_a onto the hyperplane.

Methods Comparison via Handwritten Recognition Dataset

May 3, 2019

1 Content

2 Introduction

In this project, we utilize 5 machine learning models, Linear Discriminant Analysis(LDA), Support Vector Machine(SVM), Principal Component Analysis(PCA) + Bayesian Decision Rule(BDR), PCA + K-Nearest Neighbor(KNN) and PCA + SVM(Linear and RBF kernel), to recognize handwritten digits from MNIST dataset.
The dataset is downloaded from tensorflow keras package. It consists of 60000 images for training set and 10000 images for testing set. Additionally, it has 10 categories, from 0 to 9.
This project proposes to compare different methods, draw a conclusion and share using experience.

3 Results

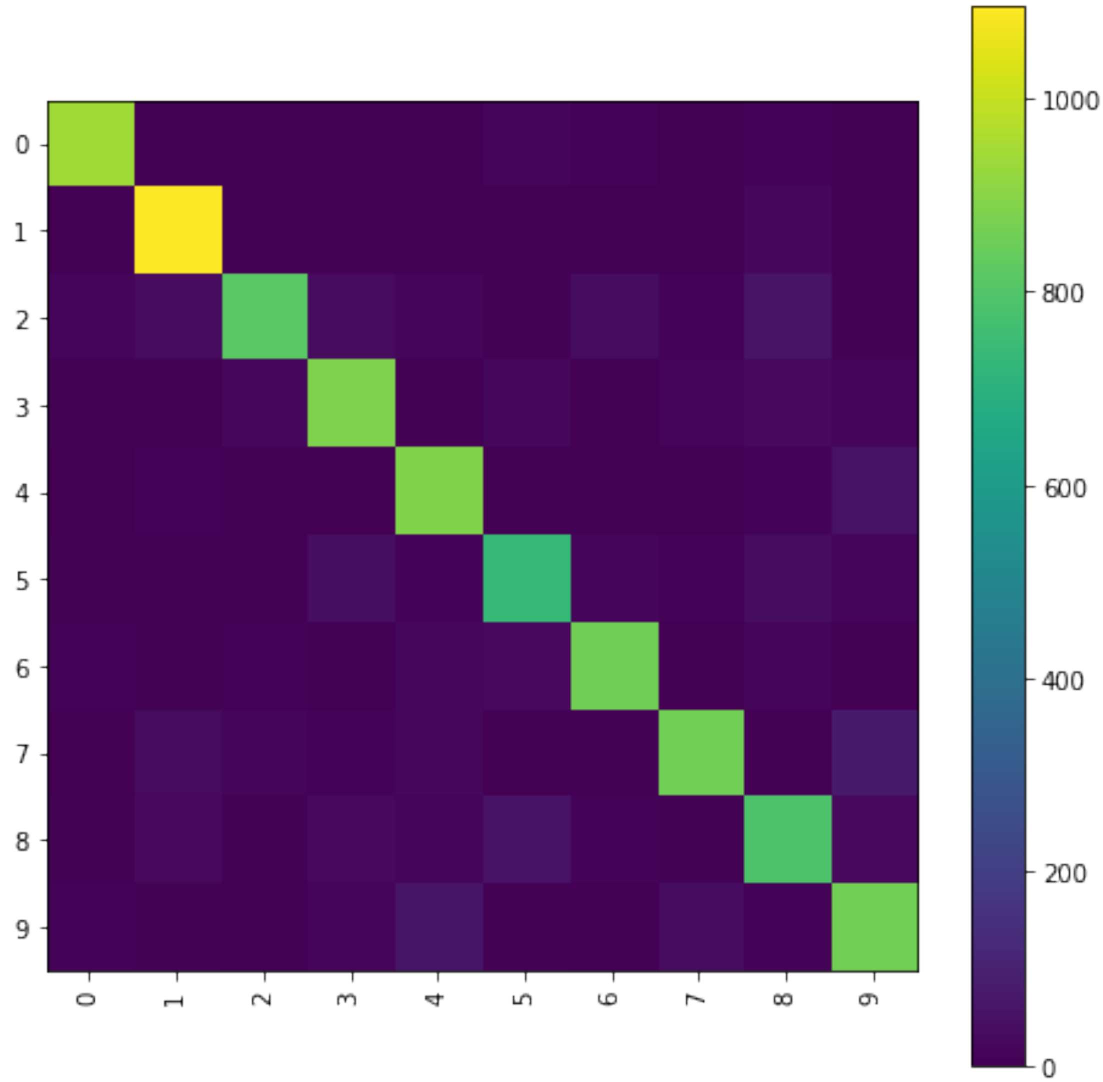
3.1 Results for LDA

```
In [10]: 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)
```

```
[[ 940    0    1    4    2   13    9    1    9    1]
 [   0 1096    4    3    2    2    3    0   25    0]
 [  15   32  816   34   21    5   37    9   57    6]
 [   5    5   25  883    4   25    3   16   29   15]
 [   0   12    6    0  888    4    7    2   10   53]
 [   8    8    4   44   12  735   15   10   38   18]
 [  12    8   11    0   25   29  857    0   16    0]
 [   2   30   15    9   22    2    0  864    4   80]
 [   7   27    8   27   20   53   10    6  790   26]
 [   9    7    1   13   63    6    0   37   12  861]]
              precision    recall  f1-score   support

           0      0.942     0.959     0.950       980
           1      0.895     0.966     0.929      1135
           2      0.916     0.791     0.849      1032
           3      0.868     0.874     0.871      1010
           4      0.839     0.904     0.870       982
           5      0.841     0.824     0.832       892
           6      0.911     0.895     0.903       958
           7      0.914     0.840     0.876      1028
           8      0.798     0.811     0.804       974
           9      0.812     0.853     0.832      1009

   micro avg      0.873     0.873     0.873     10000
   macro avg      0.874     0.872     0.872     10000
weighted avg      0.874     0.873     0.873     10000
```



3.2 Results for SVM

3.2.1 Linear Kernel

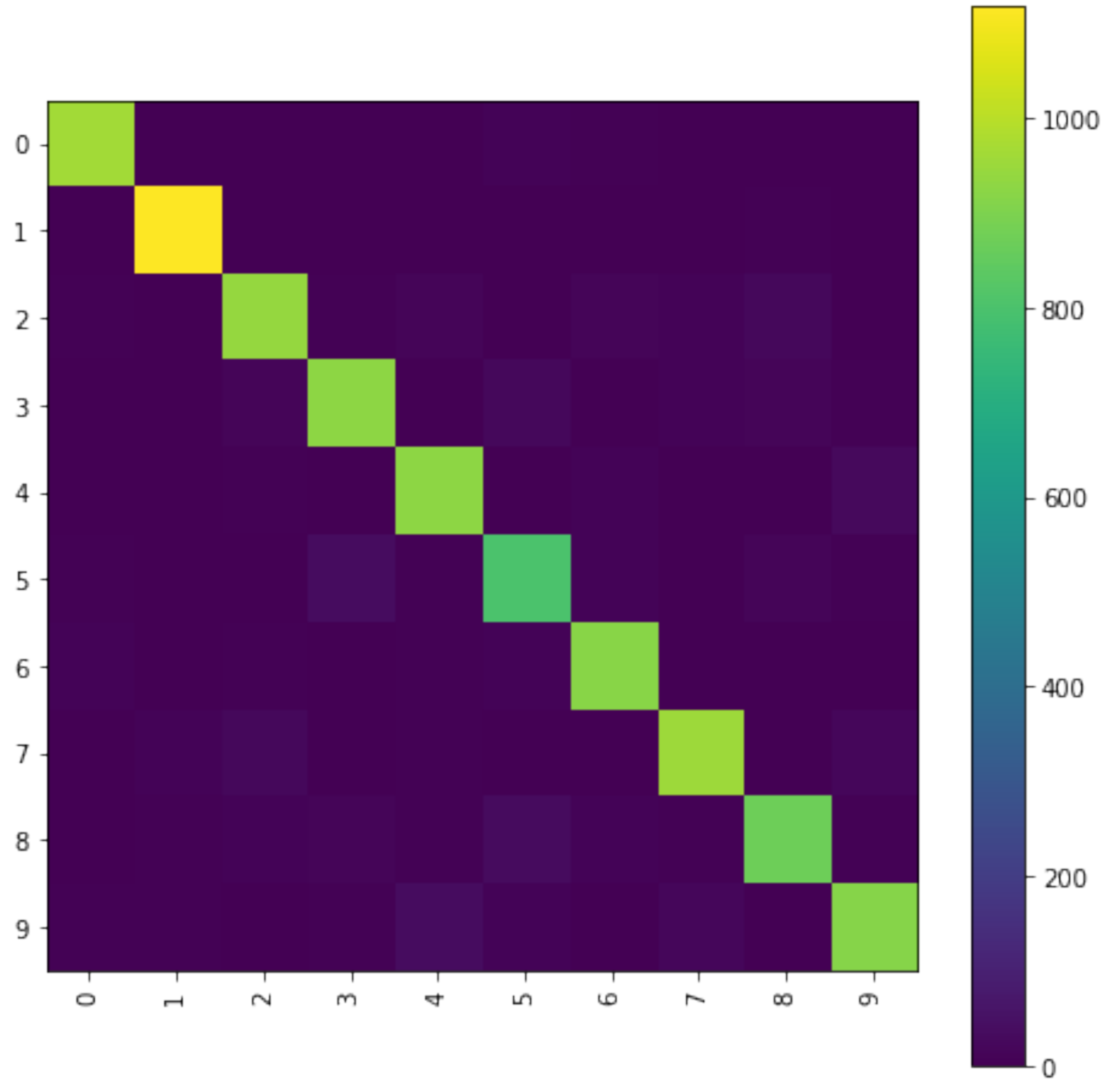
```
In [15]: print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, y_pred,
                                     target_names=list(label_dict.values()),digits=3))
         plt.figure(figsize=(8,8))
         cnf_matrix = confusion_matrix(y_test, y_pred)
         classes = 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)
```

```
[[ 962    0    0    1    0    9    5    1    1    1]
 [   0 1119    2    3    0    3    2    0    5    1]
 [   8    2  944    7   14    4   14   12   25    2]
 [   2    2   17  930    1   24    1   10   16    7]
 [   1    0    5    0  931    1   10    2    2   30]
 [   7    4    4   37    5  800   13    2   15    5]
 [  10    3    5    1    7    9  922    0    1    0]
 [   2   11   23    4    8    1    0  955    4   20]
 [   4    6   10   17    8   31   11    7  873    7]
 [   7    6    1    8   39    9    1   19    2  917]]
              precision    recall  f1-score   support

           0      0.959     0.982     0.970       980
           1      0.971     0.986     0.978      1135
           2      0.934     0.915     0.924      1032
           3      0.923     0.921     0.922      1010
           4      0.919     0.948     0.933       982
           5      0.898     0.897     0.897       892
           6      0.942     0.962     0.952       958
           7      0.947     0.929     0.938      1028
           8      0.925     0.896     0.910       974
           9      0.926     0.909     0.917      1009

   micro avg      0.935     0.935     0.935     10000
   macro avg      0.934     0.934     0.934     10000
weighted avg      0.935     0.935     0.935     10000
```



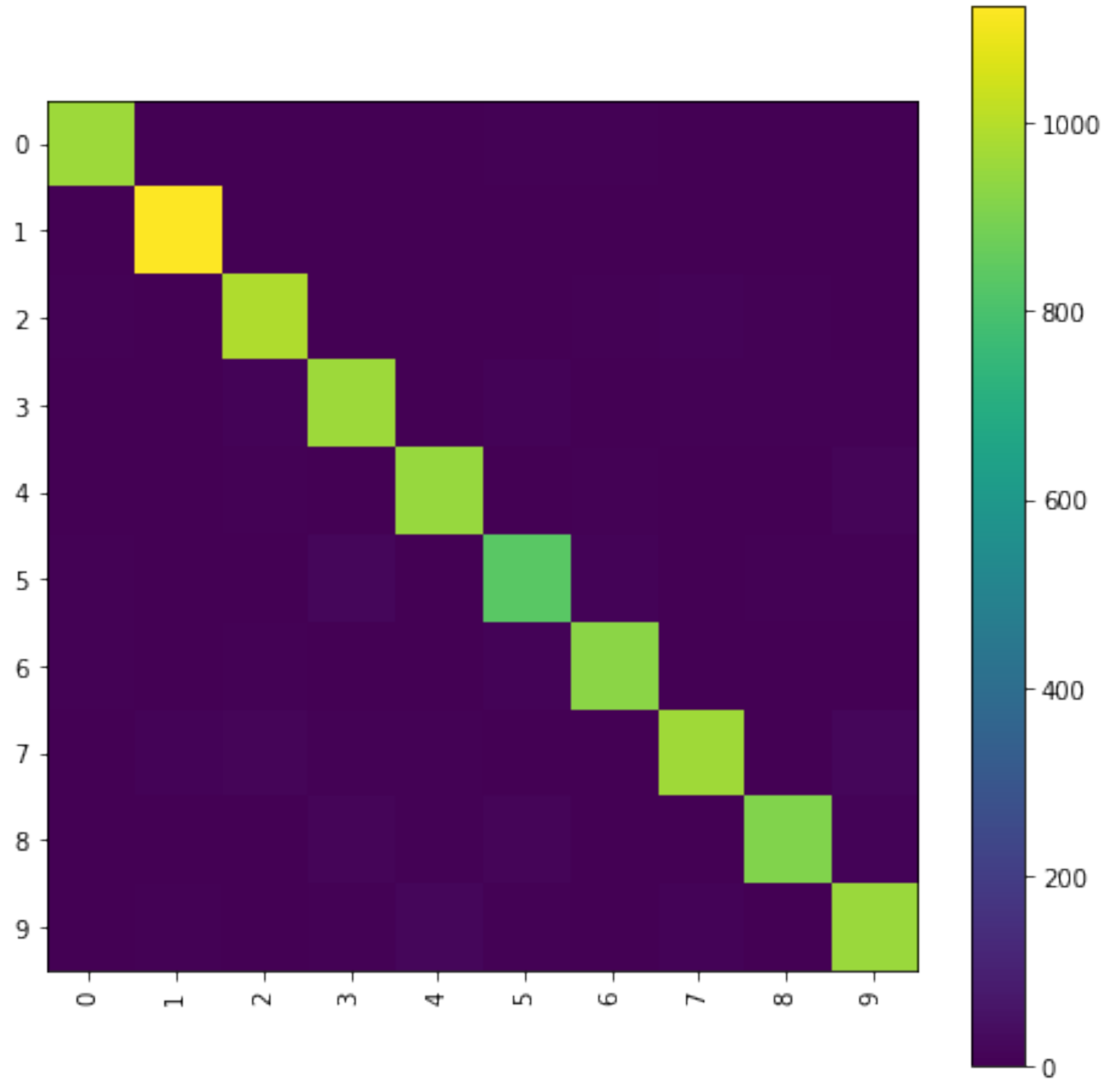
3.2.2 RBF Kernel

In [11]:
```
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)
```

```
[[ 961    0    3    1    0    7    6    1    1    0]
 [   0 1124    3    0    1    3    2    1    1    0]
 [   5    3  992    1    2    4    5   10    6    4]
 [   2    1   12  958    3   12    0    8    7    7]
 [   2    0    5    0  950    0    5    2    2   16]
 [   6    1    2   21    2  838    9    2    6    5]
 [   8    3    5    0    4   10  927    0    1    0]
 [   2   12   17    6    8    0    0  963    0   20]
 [   3    3    3   15    5   17    3    4  912    9]
 [   4    5    0    7   20    6    0   11    1  955]]
              precision    recall  f1-score   support

           0      0.968     0.981     0.974       980
           1      0.976     0.990     0.983      1135
           2      0.952     0.961     0.957      1032
           3      0.949     0.949     0.949      1010
           4      0.955     0.967     0.961       982
           5      0.934     0.939     0.937       892
           6      0.969     0.968     0.968       958
           7      0.961     0.937     0.949      1028
           8      0.973     0.936     0.954       974
           9      0.940     0.946     0.943      1009

   micro avg      0.958     0.958     0.958     10000
   macro avg      0.958     0.957     0.958     10000
weighted avg      0.958     0.958     0.958     10000
```



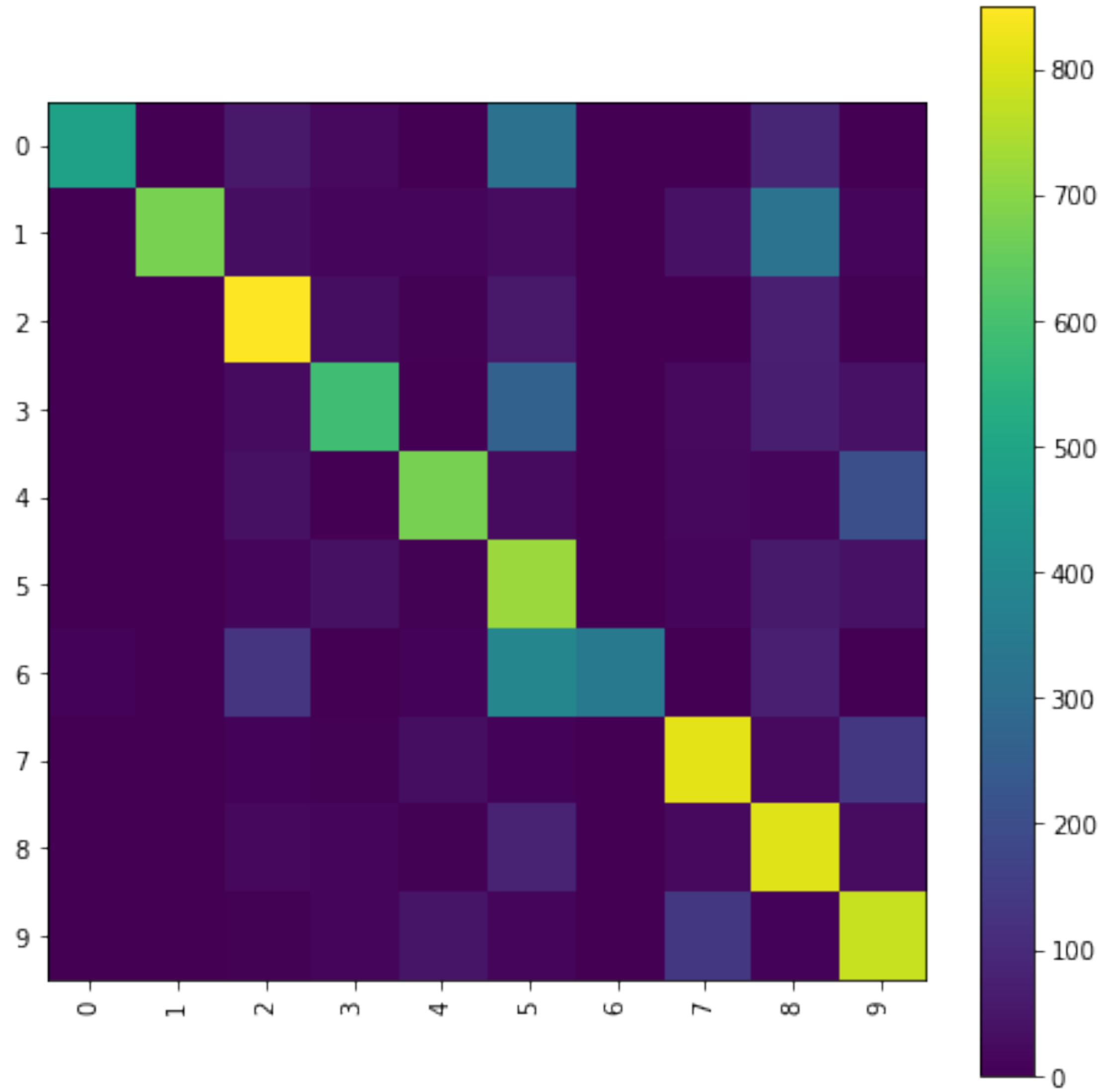
3.3 Result for PCA + BDR, 90% Eigenvalues

```
In [185]: 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)
```

```
[[485   0  57  22   2 320   1   1  92   0]
 [  0 678  32  10  10  28   0  43 323  11]
 [  3   0 850  31   6  59   0   2  75   6]
 [  0   0  24 588   1 265   0  20  70  42]
 [  0   0  37   1 675  26   0  19  15 209]
 [  0   0  11  39   4 722   0  13  60  43]
 [  7   0 131   2   8 392 344   0  74   0]
 [  0   0   7   6  33   8   0 816  20 138]
 [  0   0  18  11   4  81   0  23 810  27]
 [  0   0   4  16  49  15   0 139   7 779]]
              precision    recall  f1-score   support

           0      0.980     0.495     0.658       980
           1      1.000     0.597     0.748      1135
           2      0.726     0.824     0.772      1032
           3      0.810     0.582     0.677      1010
           4      0.852     0.687     0.761       982
           5      0.377     0.809     0.514       892
           6      0.997     0.359     0.528       958
           7      0.758     0.794     0.776      1028
           8      0.524     0.832     0.643       974
           9      0.621     0.772     0.688      1009

   micro avg      0.675     0.675     0.675     10000
   macro avg      0.764     0.675     0.676     10000
weighted avg      0.771     0.675     0.680     10000
```



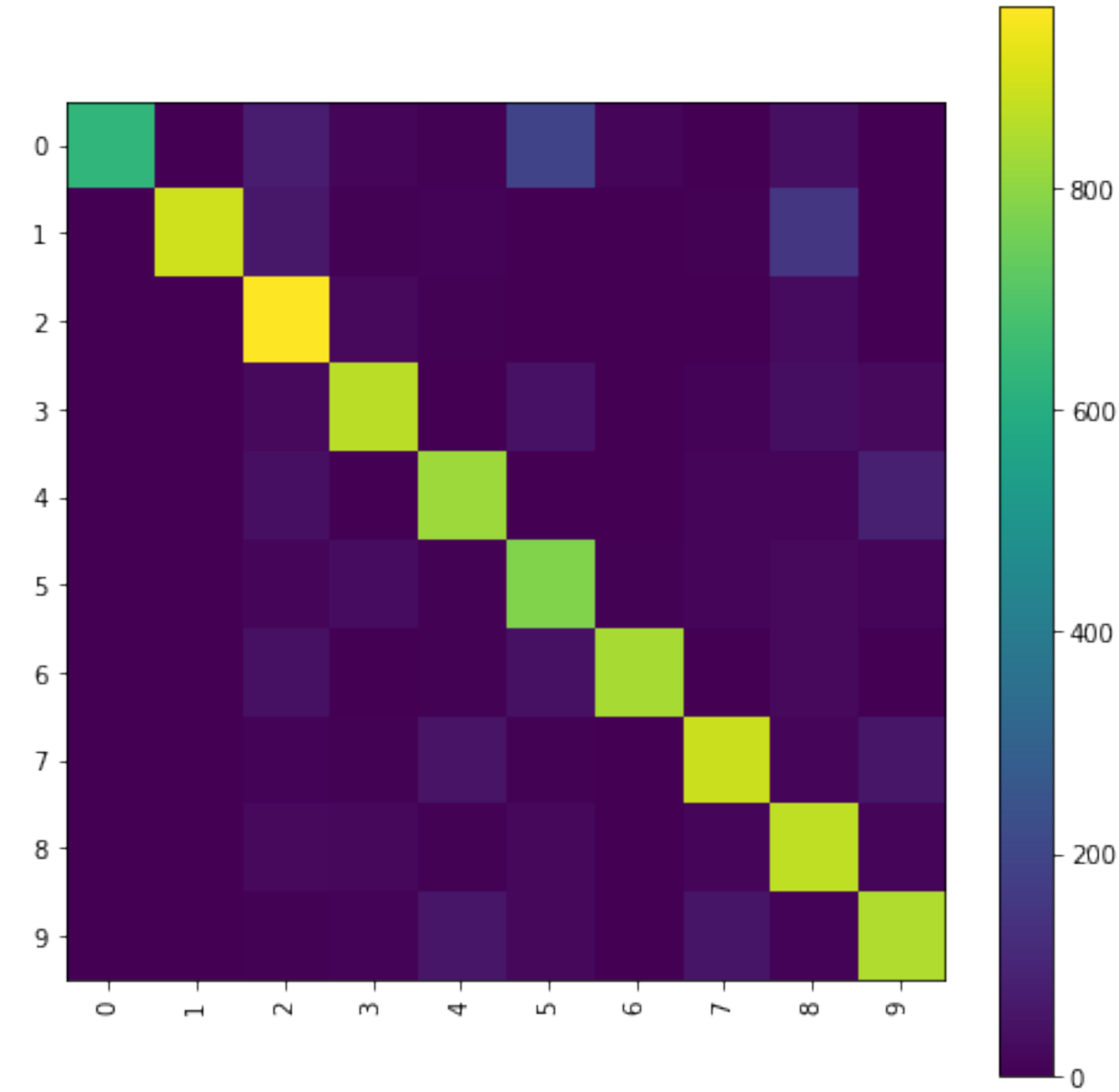
3.4 Result for PCA + BDR, 95% Eigenvalues

```
In [163]: 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)
```

```
[[636   0  74  15   6 195  12   1  41   0]
 [  0 895  62   7  11   0   0   6 154   0]
 [  0   0 964  24   7   3   0   3  28   3]
 [  0   0  23 865   1  48   1  11  36  25]
 [  0   0  41   2 821   1   0  18  13  86]
 [  1   0  16  31   4 783   4  14  25  14]
 [  0   0  43   1   7  45 836   0  26   0]
 [  0   0   8   7  52   5   0 886  13  57]
 [  1   0  26  20   5  19   3  14 873  13]
 [  0   0   7  10  57  19   0  55  11 850]]
              precision    recall  f1-score   support

           0      0.997     0.649     0.786       980
           1      1.000     0.789     0.882      1135
           2      0.763     0.934     0.840      1032
           3      0.881     0.856     0.868      1010
           4      0.846     0.836     0.841       982
           5      0.700     0.878     0.779       892
           6      0.977     0.873     0.922       958
           7      0.879     0.862     0.870      1028
           8      0.716     0.896     0.796       974
           9      0.811     0.842     0.826      1009

   micro avg      0.841     0.841     0.841     10000
   macro avg      0.857     0.842     0.841     10000
weighted avg      0.860     0.841     0.842     10000
```



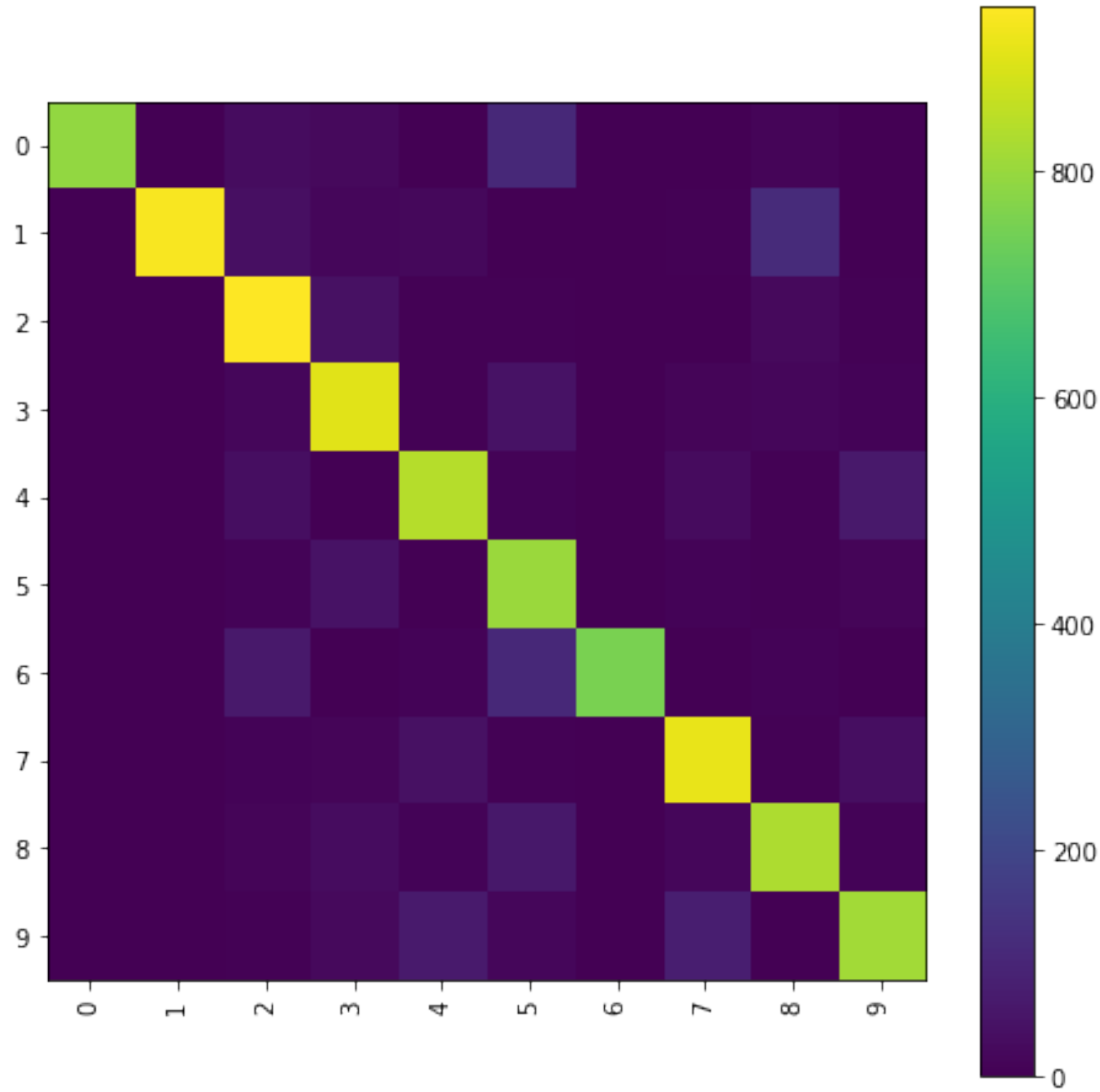
3.5 Results for PCA + KNN, 90% Eigenvalues

```
In [60]: print(confusion_matrix(y_test, Y_test_pred))
         print(classification_report(y_test, Y_test_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)
```

```
[[ 892    0    3   20    1   13   38    0   13    0]
 [   0 1110    7    2    1    1    6    4    2    2]
 [  25   32  824   29   11   16   49    3   41    2]
 [  36    1   19  778    5  106   16    6   32   11]
 [   2   16   22    2  805    5   10   19    5   96]
 [  34   10    8  205   12  541   18   13   33   18]
 [  43    5   12    6    5    3  876    0    8    0]
 [   1   30    6    5   44    5    2  780    6  149]
 [  50   27   18   31    7   33   33   12  751   12]
 [   3   13    6   25   87    6    4  183    4  678]]
              precision    recall  f1-score   support

           0      0.821     0.910     0.864       980
           1      0.892     0.978     0.933      1135
           2      0.891     0.798     0.842      1032
           3      0.705     0.770     0.736      1010
           4      0.823     0.820     0.821       982
           5      0.742     0.607     0.667       892
           6      0.833     0.914     0.872       958
           7      0.765     0.759     0.762      1028
           8      0.839     0.771     0.804       974
           9      0.700     0.672     0.686      1009

   micro avg      0.803     0.803     0.803     10000
   macro avg      0.801     0.800     0.799     10000
weighted avg      0.803     0.803     0.801     10000
```



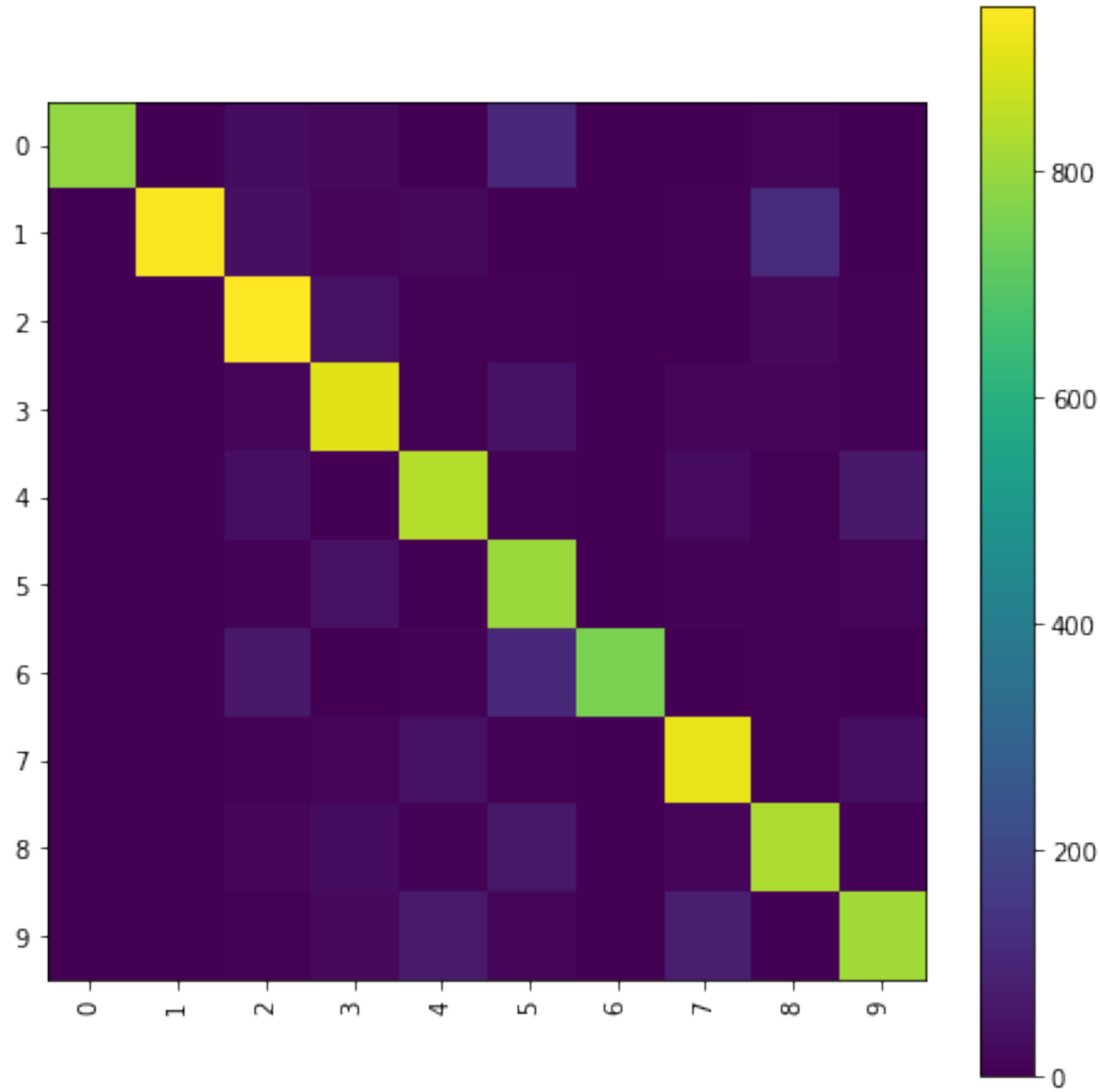
3.6 Results for PCA + KNN, 95% Eigenvalues

```
In [67]: print(confusion_matrix(y_test, Y_test_pred))
         print(classification_report(y_test, Y_test_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)
```

```
[[ 960    0    0    4    0    0   12    0    4    0]
 [   0 1125    4    2    0    1    1    1    1    0]
 [  17   11  952   11    5    4    9    7   13    3]
 [  14    2   10  899    2   55    1    4   13   10]
 [   1    9    5    0  887    0    7   10    0   63]
 [  14    5    1   92    7  731   12    4   16   10]
 [  20    4    2    1    6    3  920    0    2    0]
 [   0   27    9    0   18    0    1  925    1   47]
 [  26    8    6   13    4   20   10    6  872    9]
 [   4   10    4   18   34    6    2   54    2  875]]
              precision    recall  f1-score   support

           0      0.909     0.980     0.943       980
           1      0.937     0.991     0.963      1135
           2      0.959     0.922     0.940      1032
           3      0.864     0.890     0.877      1010
           4      0.921     0.903     0.912       982
           5      0.891     0.820     0.854       892
           6      0.944     0.960     0.952       958
           7      0.915     0.900     0.907      1028
           8      0.944     0.895     0.919       974
           9      0.860     0.867     0.864      1009

   micro avg      0.915     0.915     0.915     10000
   macro avg      0.914     0.913     0.913     10000
weighted avg      0.915     0.915     0.914     10000
```



3.7 Results for PCA + SVM, 90% Eigenvalues

3.7.1 Linear Kernel

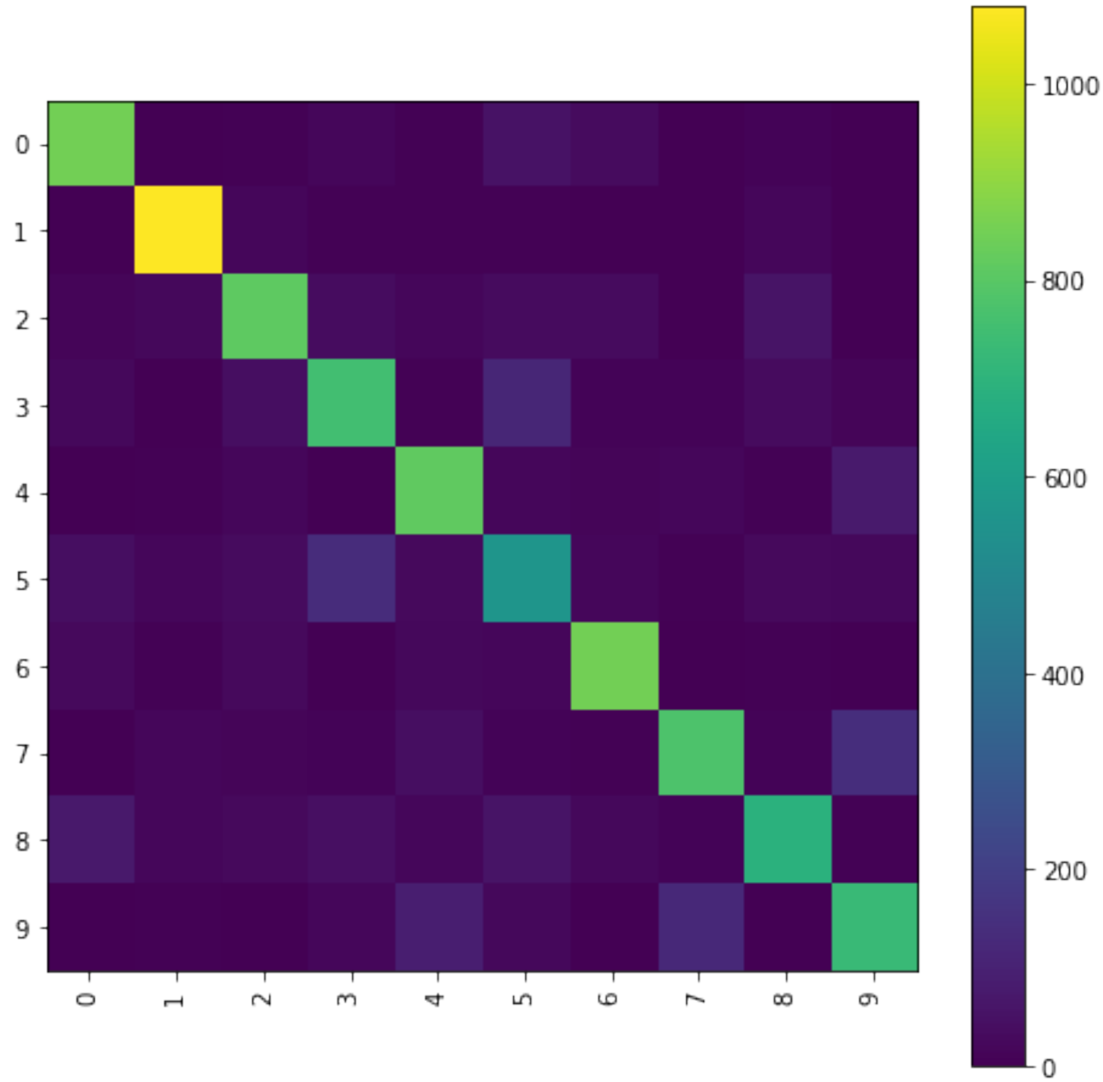
```
In [66]: print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, y_pred,
                                     target_names=list(label_dict.values()),digits=3))
         plt.figure(figsize=(8,8))
         cnf_matrix = confusion_matrix(y_test, y_pred)
         classes = 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)
```

```
[[ 850    1    7   18    6   54   32    0   12    0]
 [   0 1080   17    5    5    6    3    2   17    0]
 [  15   23  814   35   19   32   33    3   55    3]
 [  25    2   38  755    6  116   10    9   33   16]
 [   4    7   18    4  816   18   13   20    6   76]
 [  38   20   33  138   26  562   18    6   29   22]
 [  27    7   26    1   22   19  851    0    5    0]
 [   0   18   14   10   39   11    5  780    9  142]
 [  75   21   26   45   20   55   25   11  691    5]
 [   4    7    3   20   90   23    1  126    4  731]]
              precision    recall  f1-score   support

           0      0.819     0.867     0.842       980
           1      0.911     0.952     0.931      1135
           2      0.817     0.789     0.803      1032
           3      0.732     0.748     0.740      1010
           4      0.778     0.831     0.804       982
           5      0.627     0.630     0.629       892
           6      0.859     0.888     0.873       958
           7      0.815     0.759     0.786      1028
           8      0.803     0.709     0.753       974
           9      0.735     0.724     0.730      1009

   micro avg      0.793     0.793     0.793     10000
   macro avg      0.790     0.790     0.789     10000
weighted avg      0.793     0.793     0.792     10000
```



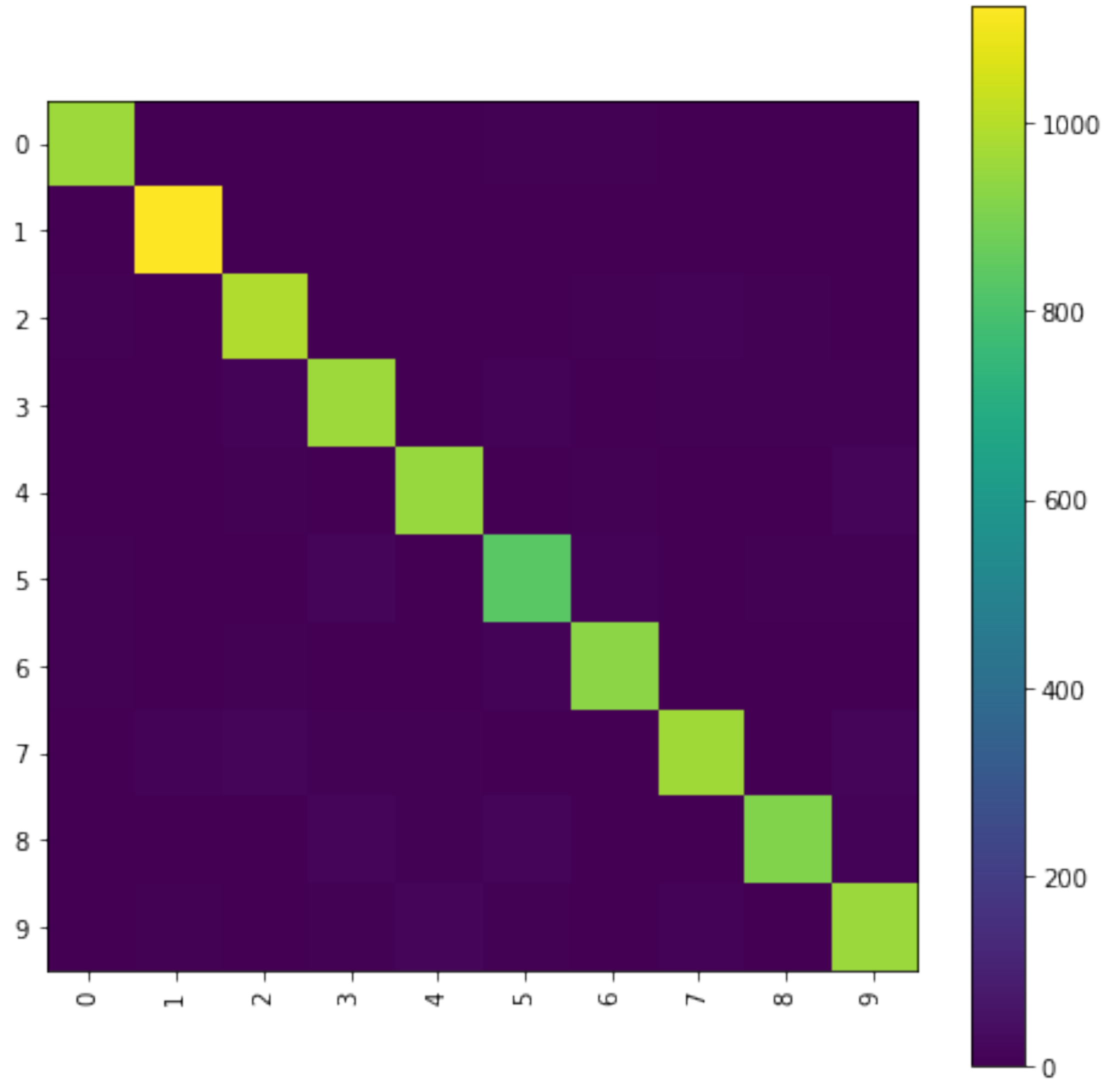
3.7.2 RBF Kernel

```
In [78]: 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)
```

```
[[ 961    0    3    1    0    7    6    1    1    0]
 [   0 1124    3    0    1    3    2    1    1    0]
 [   5    3  992    1    2    4    5   10    6    4]
 [   2    1   12  958    3   12    0    8    7    7]
 [   2    0    5    0  950    0    5    2    2   16]
 [   6    1    2   21    2  838    9    2    6    5]
 [   8    3    5    0    4   10  927    0    1    0]
 [   2   12   17    6    8    0    0  963    0   20]
 [   3    3    3   15    5   17    3    4  912    9]
 [   4    5    0    7   20    6    0   11    1  955]]
              precision    recall  f1-score   support

           0      0.968     0.981     0.974       980
           1      0.976     0.990     0.983      1135
           2      0.952     0.961     0.957      1032
           3      0.949     0.949     0.949      1010
           4      0.955     0.967     0.961       982
           5      0.934     0.939     0.937       892
           6      0.969     0.968     0.968       958
           7      0.961     0.937     0.949      1028
           8      0.973     0.936     0.954       974
           9      0.940     0.946     0.943      1009

   micro avg      0.958     0.958     0.958     10000
   macro avg      0.958     0.957     0.958     10000
weighted avg      0.958     0.958     0.958     10000
```



3.8 Results for PCA + SVM, 95% Eigenvalues

3.8.1 Linear Kernel

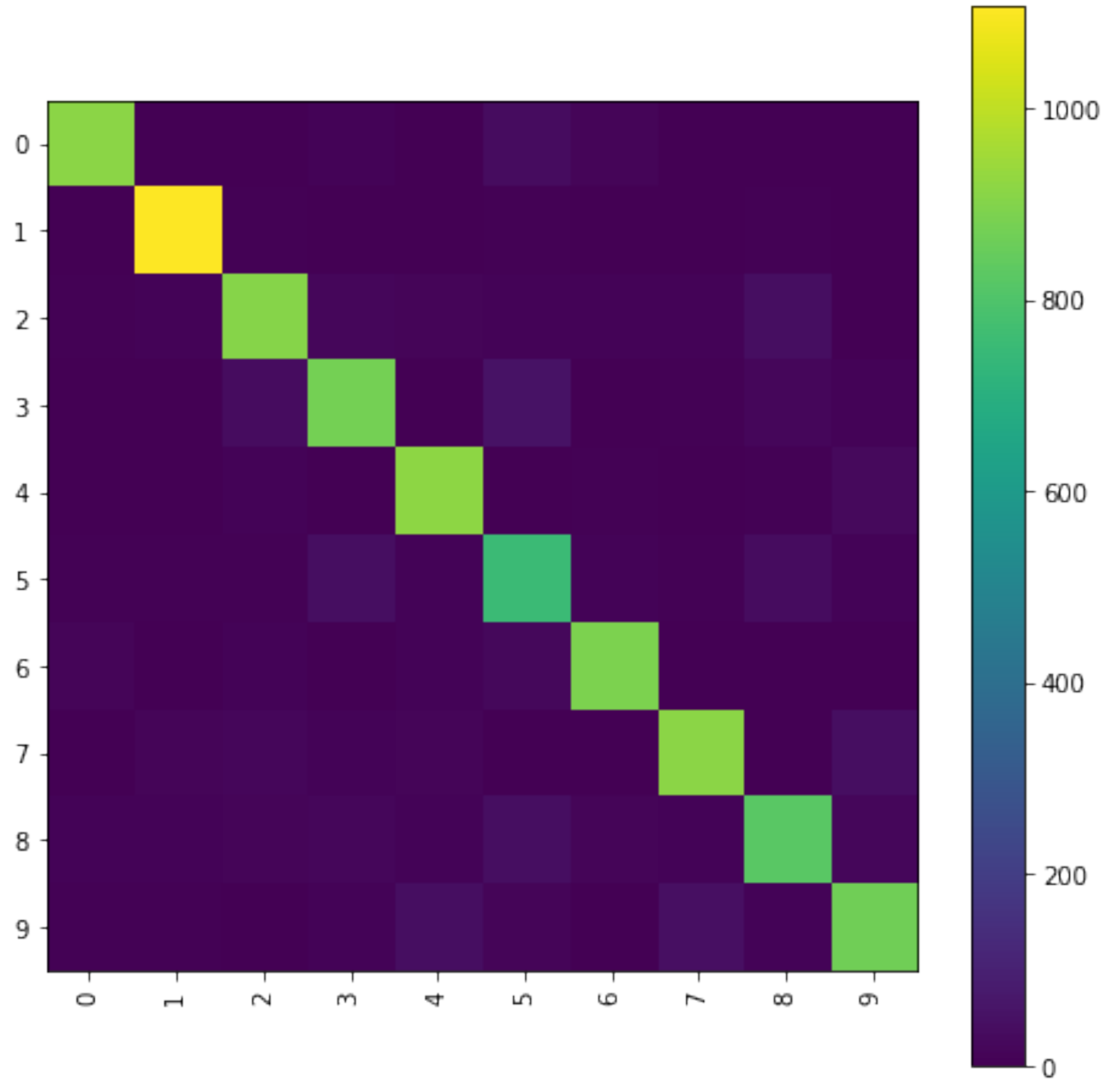
```
In [51]: print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, y_pred,
                                     target_names=list(label_dict.values()),digits=3))
         plt.figure(figsize=(8,8))
         cnf_matrix = confusion_matrix(y_test, y_pred)
         classes = 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)
```

```
[[ 914    0    3    9    1   36   14    1    2    0]
 [   0 1107    8    2    0    5    2    4    7    0]
 [   8   11  908   19   13   10   10   10   41    2]
 [   3    1   36  875    2   55    3    6   19   10]
 [   3    1   11    0  918    4    6    4    6   29]
 [   8    6    6   41   10  756   12    7   35   11]
 [  16    2   10    3   12   23  890    0    2    0]
 [   1   14   20   10   17    4    1  915    3   43]
 [  12   11   16   18   10   42   14    9  822   20]
 [   7    7    1   12   40   17    1   44   11  869]]
              precision    recall  f1-score   support

           0      0.940     0.933     0.936       980
           1      0.954     0.975     0.965      1135
           2      0.891     0.880     0.885      1032
           3      0.885     0.866     0.875      1010
           4      0.897     0.935     0.916       982
           5      0.794     0.848     0.820       892
           6      0.934     0.929     0.931       958
           7      0.915     0.890     0.902      1028
           8      0.867     0.844     0.855       974
           9      0.883     0.861     0.872      1009

   micro avg      0.897     0.897     0.897     10000
   macro avg      0.896     0.896     0.896     10000
weighted avg      0.898     0.897     0.897     10000
```



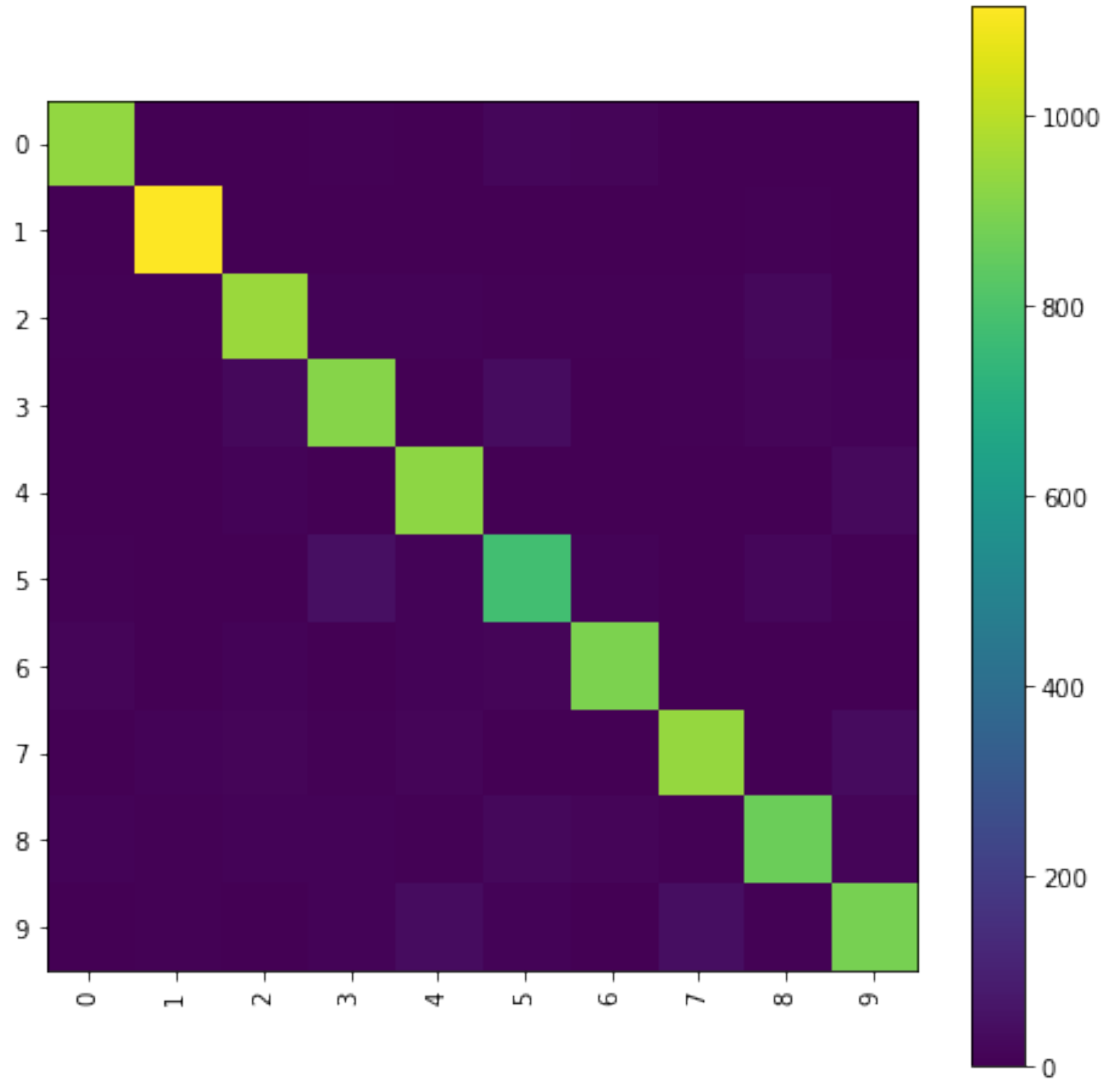
3.8.2 RBF Kernel

In [48]:
```
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)
```

```
[[ 935    0    1    8    0   20   14    1    1    0]
 [   0 1117    4    2    1    3    1    2    5    0]
 [   6    6  949   11   12    6    8    6   26    2]
 [   3    1   22  913    2   36    2    7   14   10]
 [   2    0    9    1  929    3    3    3    3   29]
 [   8    3    4   44   10  781   12    3   19    8]
 [  15    2    9    1   13   17  898    0    3    0]
 [   0   12   17    5   14    1    1  942    4   32]
 [  12    6    9   11    8   25   15    7  867   14]
 [   2    6    2   13   36   11    2   42    7  888]]
              precision    recall  f1-score   support

           0      0.951     0.954     0.953       980
           1      0.969     0.984     0.976      1135
           2      0.925     0.920     0.922      1032
           3      0.905     0.904     0.904      1010
           4      0.906     0.946     0.926       982
           5      0.865     0.876     0.870       892
           6      0.939     0.937     0.938       958
           7      0.930     0.916     0.923      1028
           8      0.914     0.890     0.902       974
           9      0.903     0.880     0.892      1009

   micro avg      0.922     0.922     0.922     10000
   macro avg      0.921     0.921     0.921     10000
weighted avg      0.922     0.922     0.922     10000
```



4 Discussion

4.1 Part 1

In this part, we seperately use Linear Discriminant Analysis(LDA) and Support Vector Machine(SVM) to classify the handwritten digits from MNIST dataset.

4.1.1 LDA

To solve this problem, we use LinearDiscriminantAnalysis function from Scikit-Learn package. We use the whole 60000 dataset to train the model, we normalize the data, scale them and then put them into the model. And the final accuracy is 87.3%. The micro average accuracy as 87.3%, macro average accuracy as 87.4% and weighted average accuracy as 87.4%. The whole model runs very fast. It has wonderful user experience.

4.1.2 SVM

2 different kernals (linear kernel and RBF kernel) are selected to compare here. LIBSVM is the chozen package for this problem. And based on duplicating trying, we find that using 15000, 20000, 25000 and 30000 rows of training set will lead to the same results, but have different processing time. Taking deadline into consideration, we decide to extract 20000 rows from training set as new training set to build model.

Linear Kernel: Initially, we set C parameter as [0.01, 0.1, 1, 5, 10, 20], and find that the smaller parameter will bring better accuracy, so we start to tune it.
After long-time tuning parameter, we find that C=0.007 among [0.0066,0.00662,0.00664,0.007] leads to best accuracy, 93.53%. And the micro average accuracy as 93.5%, macro average accuracy as 93.4% and weighted average accuracy as 93.5%.

RBF Kernel: Initially, we set C parameter as [0.01, 0.1, 1, 5, 10, 20], and find that the larger parameter will bring better accuracy. Secondly, we set C parameter as [80, 90, 100, 107, 116, 125], then find that C=80 brings best prdiction this time. So we start to tune it.
After long-time tuning parameter, we find that C=75 leads to best accuracy, 95.80%. And the micro average accuracy as 95.8%, macro average accuracy as 95.8% and weighted average accuracy as 95.8%.

4.1.3 In general

Within SVM, RBF kernel cost more time than linear kernel, but draw a bit better accuracy than linear kernel.

For such big sample dataset, compared with LDA, SVM is an extremely time-consuming method, but it's accuracy surpasses LDA's.

In a meanwhile, I also find that SVM runs so quick when the size of dataset less than 6000, even faster than LDA. Hence, if dataset is small or time is permitted, I prefer to using SVM to get good prediction.

4.2 Part 2

In this part, we firstly implement Principal Component Analysis(PCA) to deduct dimensions, then we seperately select Bayesian Decision Rule(BDR), K-Nearest Neighbor(K-NN) and Support Vector Machine(SVM) to recognize handwritten digits from MNIST dataset.

4.2.1 BDR

We do prediction here by Bayesian Decision Rule. Given that this multivariate normal density and there are 10 categories with different $\sum$, we implement formulas from case 3 on the textbook 2.6 Discriminant Functions for the Normal Density.

90% eigenvalues If we retain 90% eigenvalues in PCA part, we will keep 86 dimensions and do projection.

The accuracy is 67.47%. And the micro average accuracy as 67.5%, macro average accuracy as 76.4% and weighted average accuracy as 77.1%.

95% eigenvalues If we retain 95% eigenvalues in PCA part, we will keep 153 dimensions and do projection.

The accuracy is 84.09%. And the micro average accuracy as 84.1%, macro average accuracy as 85.7% and weighted average accuracy as 86.0%.

4.2.2 K-NN

90% eigenvalues If we retain 90% eigenvalues in PCA part, we will keep 86 dimensions and do projection.

With the new projected training and testing dataset, we continue to do K-NN. When K = 3, we get accuracy 79.55%; when K = 5, we get accuracy 80.35%. Therefore, we set K = 5 to do prediction and get the micro average accuracy as 80.3%, macro average accuracy as 80.1% and weighted average accuracy as 80.3%.

95% eigenvalues If we retain 95% eigenvalues in PCA part, we will keep 153 dimensions and do projection.

With the new projected training and testing dataset, we continue to do K-NN. When K = 3, we get accuracy 91.46%; when K = 5, we get accuracy 91.42%. Therefore, we set K = 3 to do prediction and get the micro average accuracy as 91.5%, macro average accuracy as 91.4% and weighted average accuracy as 91.5%.

4.2.3 SVM

90% eigenvalues If we retain 90% eigenvalues in PCA part, we will keep 86 dimensions and do projection.

Linear Kernel: Initially, we set C parameter as [0.01, 0.1, 1, 5, 10, 20], and find that the larger parameter will bring better accuracy, so we start to tune it.

After long-time tuning parameter, we finally set C as [53.95,54,54.1,54.3,54.5] and find that C=54 will bring us the best accuracy, 79.30%. And the micro average accuracy as 79.3%, macro average accuracy as 79.0% and weighted average accuracy as 79.3%.

RBF Kernel: Initially, we set C parameter as [0.01, 0.1, 1, 5, 10, 20], and find that the larger parameter will bring better accuracy, so we start to tune it.

After long-time tuning parameter, we finally set C as [121,124,127,129] and find that C=129 will bring us the best accuracy, 83.45%. And the micro average accuracy as 83.5%, macro average accuracy as 83.2% and weighted average accuracy as 83.4%.

95% eigenvalues If we retain 95% eigenvalues in PCA part, we will keep 153 dimensions and do projection.

Linear Kernel: Initially, we set C parameter as [0.01, 0.1, 1, 5, 10, 20], and find that the parameter between 5 and 10 will bring better accuracy, so we start to tune it.

After long-time tuning parameter, we finally set C as [9.03,9.05,9.055,9.06] and find that C=9.05 will bring us the best accuracy, 89.74%. And the micro average accuracy as 89.7%, macro average accuracy as 89.6% and weighted average accuracy as 89.8%.

RBF Kernel: Initially, we set C parameter as [0.01, 0.1, 1, 5, 10, 20], and find that larger parameter will bring better accuracy, so we start to tune it.

After long-time tuning parameter, we finally set C as [114,120,125,129] and find that C=125 will bring us the best accuracy, 92.19%. And the micro average accuracy as 92.2%, macro average accuracy as 92.1% and weighted average accuracy as 92.2%.

4.2.4 In general

BDR Here, we draw a conclusion that retaining 95% eigenvalues brings about much better prediction result with 84.09% accuracy than that of 90% eigenvalues, which is 67.47%.

K-NN Here, we draw a conclusion that retaining 95% eigenvalues and setting K = 5 will lead to best prediction result with 91.42% accuracy.

And compared with purely using K-NN, PCA+K-NN doesn't have as good prediction result as K-NN does.

SVM Usually, larger C parameter contribute to better accuracy.

Apparently, keeping 95% eigenvalues performs much better keeping 90% eigenvalues for both kernel in prediction, since that keeping 95% eigenvalues leaves more information.

For 90% eigenvalues, the accuracy of RBF kernel, 83.45%, is better than that of linear kernel, 79.30%.
For 95% eigenvalues, the accuracy of RBF kernel, 92.19%, is better than that of linear kernel, 89.74%.

Compared with purely using SVM, PCA+SVM doesn't have as good prediction result as SVM does.

Compare 3 Methods About eigenvalues, apparently, retaining 95% eigenvalues predicts much better than retaining 90% one for all three models.

About processing time, PCA + BDR < PCA + KNN < PCA + SVM(Linear Kernel) < PCA + SVM(RBF Kernel). In this project, PCA, BDR and KNN are coded by ourselves, SVM is called from LIBSVM. I know that SVM is a complex model, but I still want to say that self-coding function runs so fast, which encourages me to code more powerful machine learning models by myself in the future.

About accuracy, PCA + SVM(RBF Kernel) > PCA + KNN > PCA + SVM(Linear Kernel) > PCA +BDR. It's hard to deny that SVM has the best prediction result among the three methods. Currently, compared with matured package, my self-code one still gets a long road to go. However, this is a good start.

5 Conclusion

This project ignite me to think deeply about these models.

About dimension redection, we can check from the SVM that doing PCA + SVM predicts little worse than purely doing SVM, but PCA + SVM is time-efficiency, it costs less time. It can be an alternative choice to get results as soon as possible and loss information as less as possible.

About processing time, LDA < PCA + BDR < PCA + KNN < PCA + SVM(Linear Kernel) < PCA + SVM(RBF Kernel) < SVM(Linear Kernel) < SVM(RBF Kernel).

About accuracy, SVM-RBF(95.80%) > SVM-Linear(93.53%) >
PCA + SVM-RBF,95%(92.19) > PCA + KNN, 95%(91.42%) > PCA + SVM-Linear,95%(89.74) >
LDA(87.3%) >
PCA + BDR,95%(84.09%) >
PCA + SVM-RBF,90%(83.45%) > PCA + KNN,90%(80.35%) > PCA + SVM-Linear,90%(79.30%) >
PCA + BDR,90%(67.47%).

We can see, SVM totally surpasses all the other methods; methods retaining 95% eigenvalues totally win those retaining 90% eigenvalues; the SVM models are built with part of training dataset, we can make sure that the whole dataset will bring about better prediction.

Meanwhile, we also notice that there are some good entry points from this project. For example, enhancing the SVM's processing time while dealing with big data besides doing PCA and improving KNN's and BDR's prediction accuracy. Have to admit that those meethods are so weak in front of deep learning, but there must be some meaning to think about this kind of problems.

6 Appendix

```
In [1]: # import dataset and seperate them as train set and test set
        # index x represents image, index y represents label
        import sys
        path = "/Users/sunjian/Downloads/libsvm-3.23/python"
        sys.path.append(path)
        import os
        import cv2
        import random
        import sklearn
        import numpy as np
        import svm, svmutil
        from svm import *
        from svmutil import *
        import sklearn.metrics
        import tensorflow as tf
        from numpy.linalg import *
        from sklearn.svm import SVC
        import matplotlib.pyplot as plt
        from sklearn.preprocessing import LabelEncoder
        from sklearn.preprocessing import StandardScaler
        from sklearn.metrics import classification_report, confusion_matrix
        from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

In [2]: # 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'}
```

7 Part 1

7.1 LDA

```
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=np.float32(x_train)
        X_test=np.float32(x_test)
        X_train = X_train / 255.0
        X_test = 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 [5]: # scaling
        sc = StandardScaler()
        x_train = sc.fit_transform(x_train)
        x_test = sc.transform(x_test)
```

```
In [6]: #creating a LDA object
        lda = LDA(n_components=2)
        lda.fit_transform(x_train, y_train) #learning the projection matrix
        y_pred = lda.predict(x_test) #gives you the predicted label for each sample
        y_prob = lda.predict_proba(x_test)
```

```
/Users/sunjian/anaconda3/envs/tfw/lib/python3.6/site-packages/sklearn/discriminant_analysis.py
  warnings.warn("Variables are collinear.")
```

```
In [7]: 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 8730 / 10000 correct
Accuracy = 0.873000
```

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

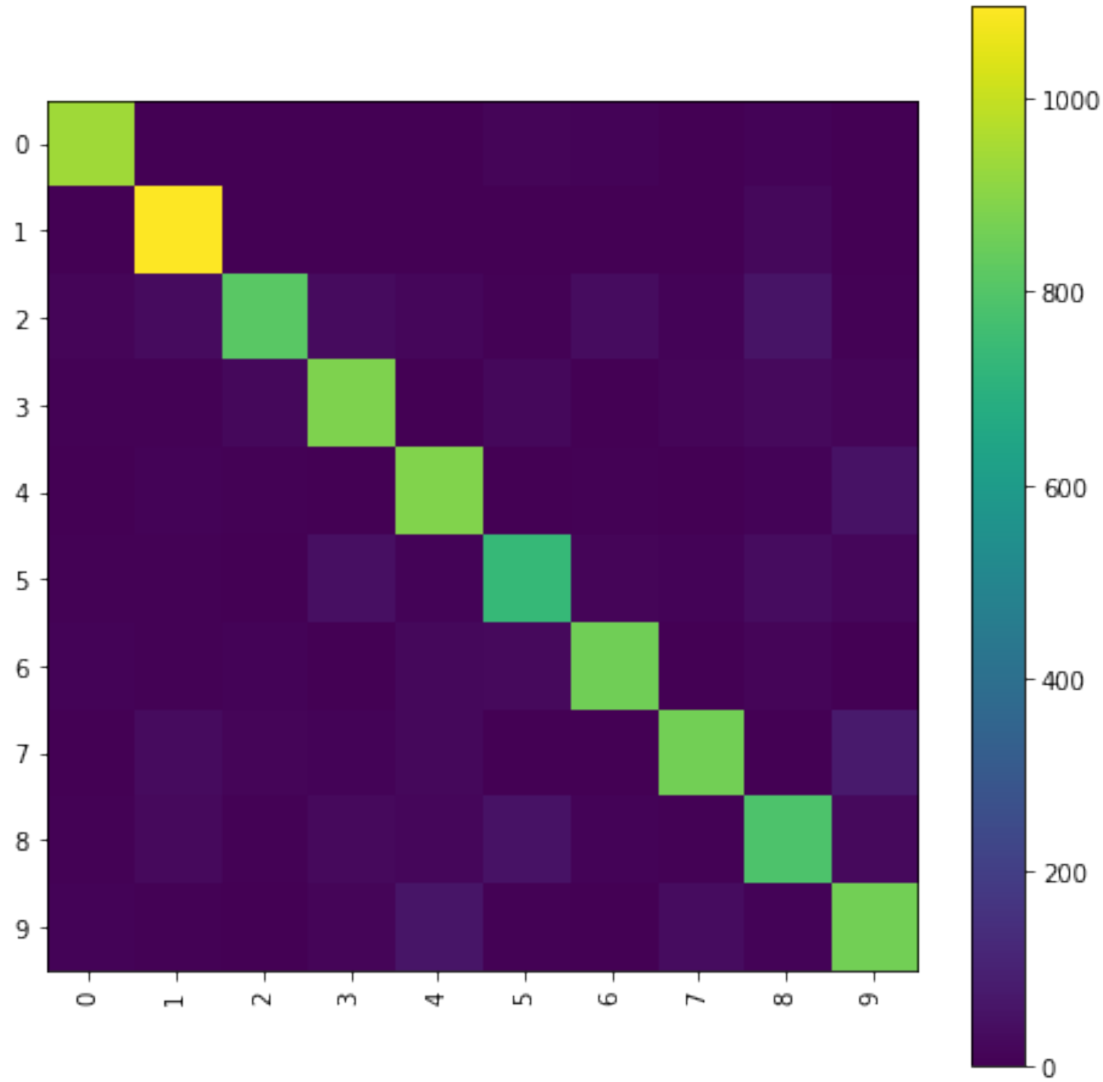
```
[[ 940    0    1    4    2   13    9    1    9    1]
 [   0 1096    4    3    2    2    3    0   25    0]
 [  15   32  816   34   21    5   37    9   57    6]
 [   5    5   25  883    4   25    3   16   29   15]
```

```
[   0   12    6    0  888    4    7    2   10   53]
[   8    8    4   44   12  735   15   10   38   18]
[  12    8   11    0   25   29  857    0   16    0]
[   2   30   15    9   22    2    0  864    4   80]
[   7   27    8   27   20   53   10    6  790   26]
[   9    7    1   13   63    6    0   37   12  861]]
              precision    recall  f1-score   support

           0      0.942     0.959     0.950       980
           1      0.895     0.966     0.929      1135
           2      0.916     0.791     0.849      1032
           3      0.868     0.874     0.871      1010
           4      0.839     0.904     0.870       982
           5      0.841     0.824     0.832       892
           6      0.911     0.895     0.903       958
           7      0.914     0.840     0.876      1028
           8      0.798     0.811     0.804       974
           9      0.812     0.853     0.832      1009

   micro avg      0.873     0.873     0.873     10000
   macro avg      0.874     0.872     0.872     10000
weighted avg      0.874     0.873     0.873     10000
```

In [9]:
```
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)
```



7.2 SVM

```
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=np.float32(x_train)
        X_test=np.float32(x_test)
        X_train = X_train / 255.0
        X_test = 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 [4]: X_train = x_train.tolist()
        Y_train = y_train.tolist()
```

```
X_test  = x_test.tolist()
Y_test  = y_test.tolist()
```

7.2.1 Linear Kernel

```
In [6]: C=[0.0066,0.00662,0.00664,0.007]
        prob = svm_problem(Y_train[0:20000], X_train[0:20000])
        param1 = svm_parameter('-t 0 -c 0.0066 -b 1')
        param2 = svm_parameter('-t 0 -c 0.00662 -b 1')
        param3 = svm_parameter('-t 0 -c 0.00664 -b 1')
        param4 = svm_parameter('-t 0 -c 0.007 -b 1')
```

```
In [11]: P=[param1,param2,param3,param4]
         for i in range(len(P)):
             model = svm_train(prob, P[i])
             p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
             print(p_acc)
```

```
Model supports probability estimates, but disabled in predicton.
Accuracy = 93.48% (9348/10000) (classification)
(93.47999999999999, 1.1326, 0.8693896414231087)
Model supports probability estimates, but disabled in predicton.
Accuracy = 93.5% (9350/10000) (classification)
(93.5, 1.1324, 0.8694092963493714)
Model supports probability estimates, but disabled in predicton.
Accuracy = 93.5% (9350/10000) (classification)
(93.5, 1.1344, 0.8691891556099334)
Model supports probability estimates, but disabled in predicton.
Accuracy = 93.5% (9350/10000) (classification)
(93.5, 1.1376, 0.8688183095772)
```

```
In [12]: param = svm_parameter('-t 0 -c 0.007 -b 1')
         model = svm_train(prob, param)
         p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
```

```
Model supports probability estimates, but disabled in predicton.
Accuracy = 93.53% (9353/10000) (classification)
```

```
In [13]: y_pred=p_label
         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)))
```

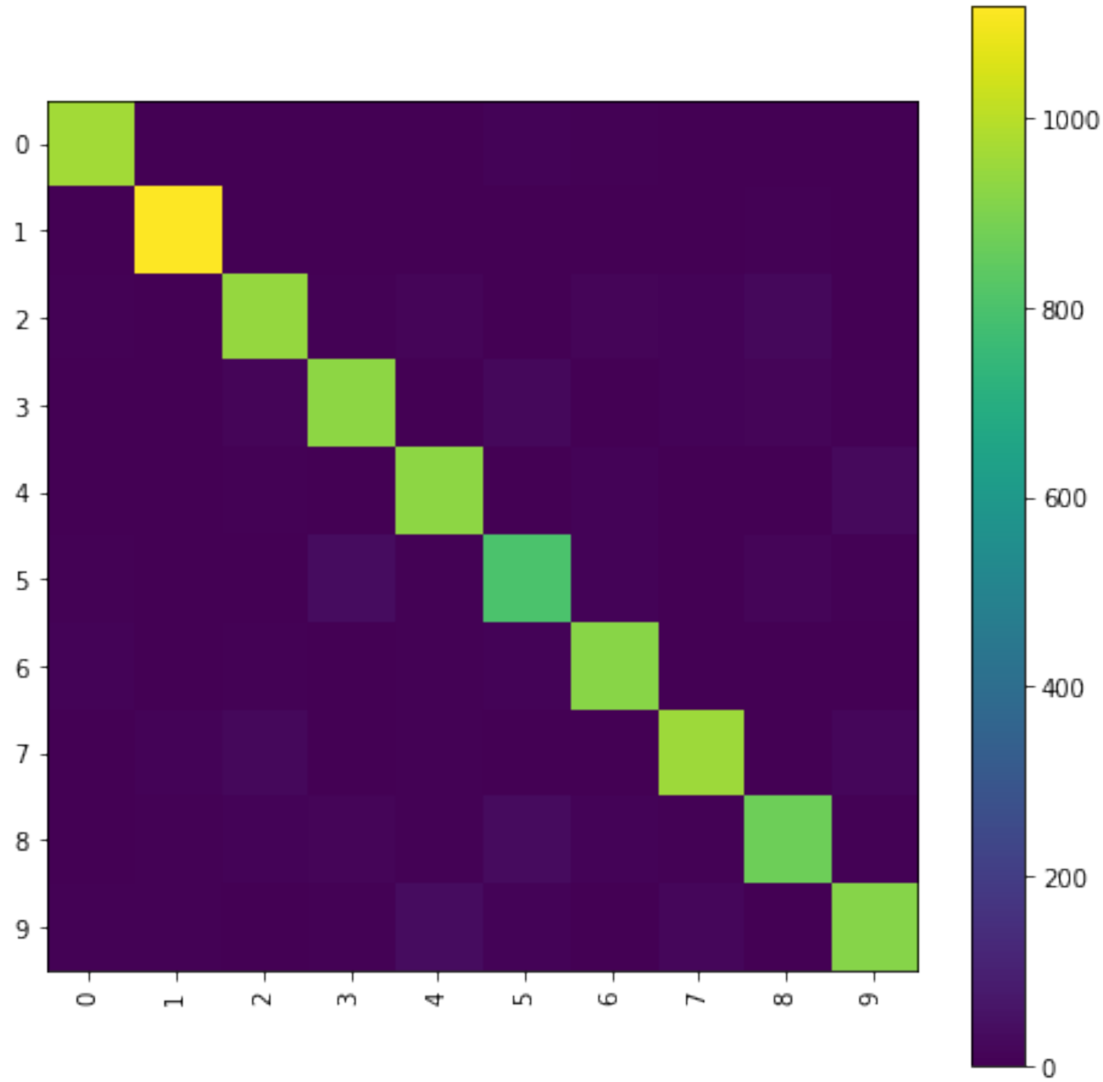
```
Got 9353 / 10000 correct
Accuracy = 0.935300
```

```
In [14]: 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)
```

```
[[ 962    0    0    1    0    9    5    1    1    1]
 [   0 1119    2    3    0    3    2    0    5    1]
 [   8    2  944    7   14    4   14   12   25    2]
 [   2    2   17  930    1   24    1   10   16    7]
 [   1    0    5    0  931    1   10    2    2   30]
 [   7    4    4   37    5  800   13    2   15    5]
 [  10    3    5    1    7    9  922    0    1    0]
 [   2   11   23    4    8    1    0  955    4   20]
 [   4    6   10   17    8   31   11    7  873    7]
 [   7    6    1    8   39    9    1   19    2  917]]
              precision    recall  f1-score   support

           0      0.959     0.982     0.970       980
           1      0.971     0.986     0.978      1135
           2      0.934     0.915     0.924      1032
           3      0.923     0.921     0.922      1010
           4      0.919     0.948     0.933       982
           5      0.898     0.897     0.897       892
           6      0.942     0.962     0.952       958
           7      0.947     0.929     0.938      1028
           8      0.925     0.896     0.910       974
           9      0.926     0.909     0.917      1009

   micro avg      0.935     0.935     0.935     10000
   macro avg      0.934     0.934     0.934     10000
weighted avg      0.935     0.935     0.935     10000
```



7.2.2 RBF Kernel

```
In [5]: C=[50,60,70,75]
        prob = svm_problem(Y_train[0:20000], X_train[0:20000])
        param1 = svm_parameter('-t 2 -c 50 -b 1')
        param2 = svm_parameter('-t 2 -c 60 -b 1')
        param3 = svm_parameter('-t 2 -c 70 -b 1')
        param4 = svm_parameter('-t 2 -c 75 -b 1')
```

```
In [74]: P=[param1,param2,param3,param4]
         for i in range(len(P)):
             model = svm_train(prob, P[i])
             p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
             print(p_acc)
```

```
Model supports probability estimates, but disabled in predicton.
Accuracy = 95.62% (9562/10000) (classification)
(95.62, 0.7919, 0.9078695802035437)
Model supports probability estimates, but disabled in predicton.
Accuracy = 95.69% (9569/10000) (classification)
(95.69, 0.7559, 0.9120079755730383)
Model supports probability estimates, but disabled in predicton.
Accuracy = 95.76% (9576/10000) (classification)
(95.76, 0.7498, 0.9127387339953145)
Model supports probability estimates, but disabled in predicton.
Accuracy = 95.8% (9580/10000) (classification)
(95.8, 0.7406, 0.9137451356503374)
```

```
In [8]: param = svm_parameter('-t 2 -c 75 -b 1')
        model = svm_train(prob, param)
        p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
```

```
Model supports probability estimates, but disabled in predicton.
Accuracy = 95.8% (9580/10000) (classification)
```

```
In [9]: y_pred=p_label
        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 9580 / 10000 correct
Accuracy = 0.958000
```

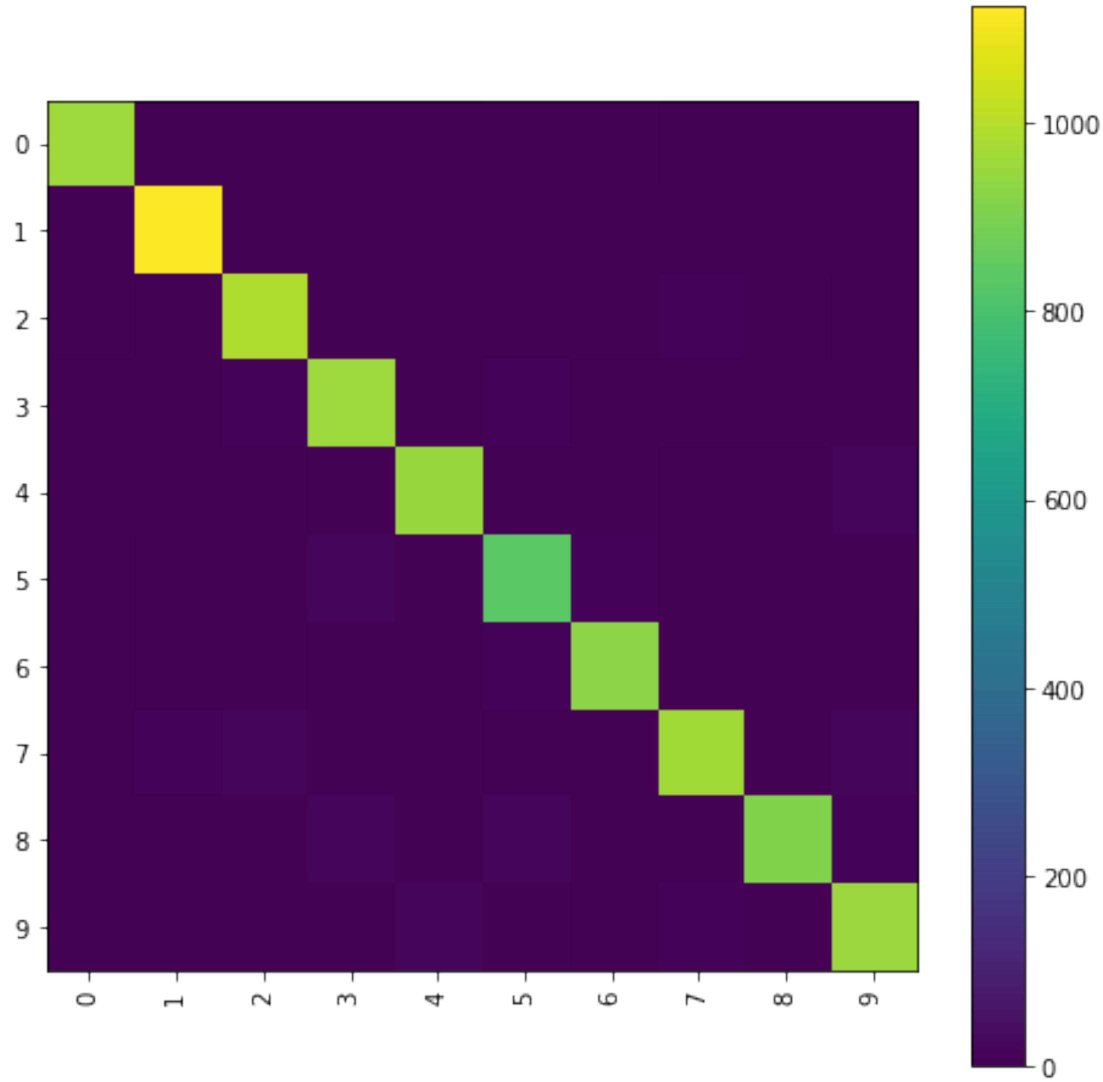
```
In [10]: print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, y_pred,
                                     target_names=list(label_dict.values()),digits=3))
         plt.figure(figsize=(8,8))
         cnf_matrix = confusion_matrix(y_test, y_pred)
         classes = 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)
```

```
[[ 961    0    3    1    0    7    6    1    1    0]
 [   0 1124    3    0    1    3    2    1    1    0]
 [   5    3  992    1    2    4    5   10    6    4]
 [   2    1   12  958    3   12    0    8    7    7]
 [   2    0    5    0  950    0    5    2    2   16]
```

```
 [   6    1    2   21    2  838    9    2    6    5]
 [   8    3    5    0    4   10  927    0    1    0]
 [   2   12   17    6    8    0    0  963    0   20]
 [   3    3    3   15    5   17    3    4  912    9]
 [   4    5    0    7   20    6    0   11    1  955]]
              precision    recall  f1-score   support

           0      0.968     0.981     0.974       980
           1      0.976     0.990     0.983      1135
           2      0.952     0.961     0.957      1032
           3      0.949     0.949     0.949      1010
           4      0.955     0.967     0.961       982
           5      0.934     0.939     0.937       892
           6      0.969     0.968     0.968       958
           7      0.961     0.937     0.949      1028
           8      0.973     0.936     0.954       974
           9      0.940     0.946     0.943      1009

   micro avg      0.958     0.958     0.958     10000
   macro avg      0.958     0.957     0.958     10000
weighted avg      0.958     0.958     0.958     10000
```



8 Part 2

8.1 PCA + BDR

In [175]:
```
# 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)
x_test = x_test.reshape(np.shape(x_test)[0], 28*28)
```

In [176]:
```
# add noise to images
def add_noisy(image):
```

```
        row,col = np.shape(image)
        mean = 0
        var = 0.01
        sigma = var**0.5
        gauss = np.random.normal(mean,sigma,(row,col))
        gauss = np.reshape(gauss,(row,col))
        noisy = image + gauss
        return noisy
```

In [177]:
```
# add noise and reconstruct dataset and stack them as a big one
# for dimension deduction
X_train = add_noisy(x_train)
X_test = add_noisy(x_test)
big_X=np.vstack((X_train,X_test))
```

8.1.1 PCA Class to Deduct Dimensions for Both Training and Testing Set

In [167]:
```
# 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, xtr_m, tr_vec, keep_dim):
            x_proj= np.dot(xtr_m, tr_vec.T[:,0:keep_dim])
            return x_proj
```

8.1.2 Bayesian Decision Rule Class

```
In [168]: # 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)/(len(data)-1)
                  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
```

8.1.3 Retain 90% Eigenvalues

```
In [178]: # 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(xtr_m, tr_vec, keep_dim)
          print(keep_dim)
```

```
86
```

```
In [179]: # 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 [180]: # 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 [181]:
```
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 [182]:
```
# 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 [183]:
```
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)))
```

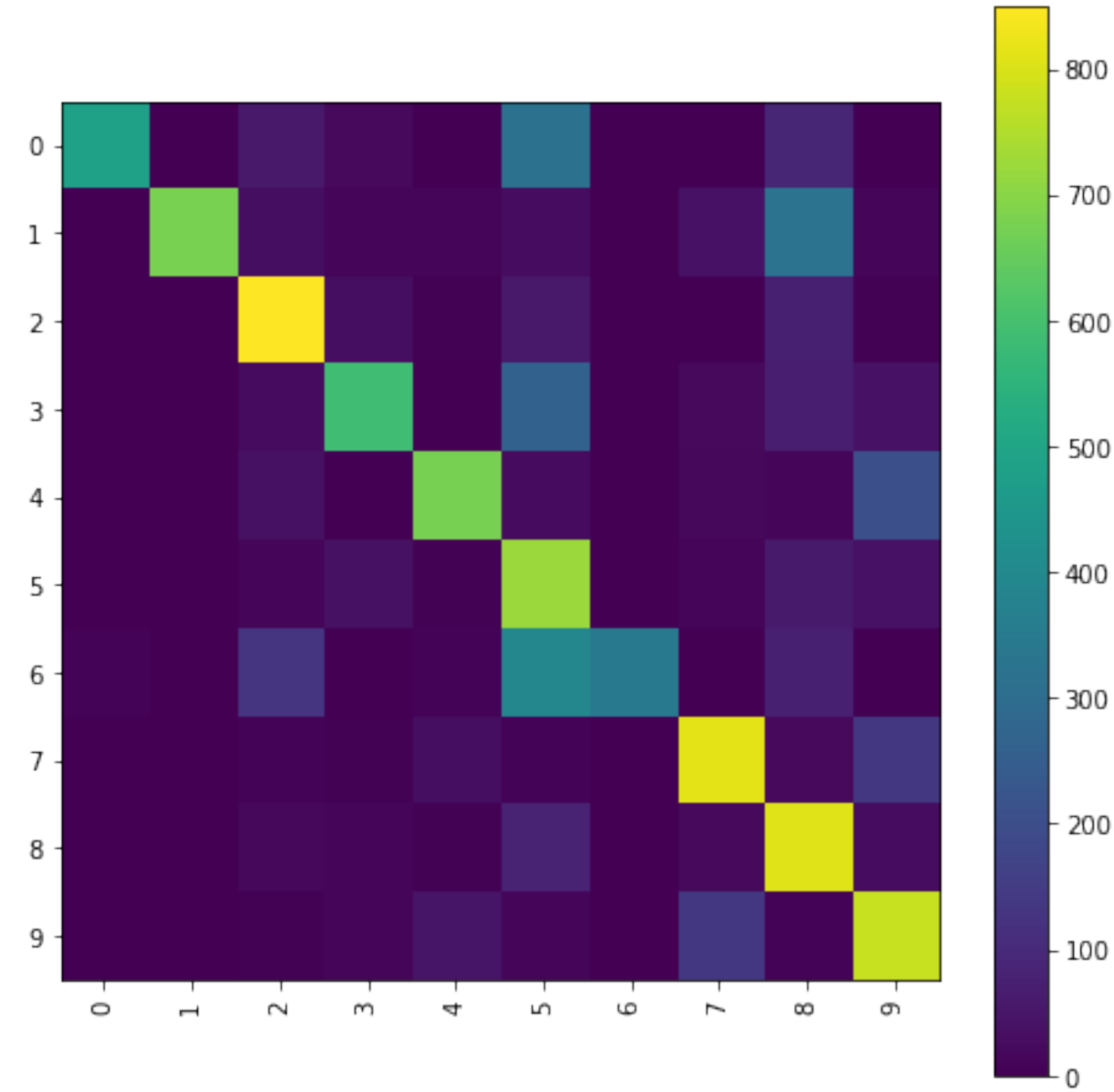
```
Got 6747 / 10000 correct
Accuracy = 0.674700
```

```
In [184]: 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)
```

```
[[485   0  57  22   2 320   1   1  92   0]
 [  0 678  32  10  10  28   0  43 323  11]
 [  3   0 850  31   6  59   0   2  75   6]
 [  0   0  24 588   1 265   0  20  70  42]
 [  0   0  37   1 675  26   0  19  15 209]
 [  0   0  11  39   4 722   0  13  60  43]
 [  7   0 131   2   8 392 344   0  74   0]
 [  0   0   7   6  33   8   0 816  20 138]
 [  0   0  18  11   4  81   0  23 810  27]
 [  0   0   4  16  49  15   0 139   7 779]]
              precision    recall  f1-score   support

           0      0.980     0.495     0.658       980
           1      1.000     0.597     0.748      1135
           2      0.726     0.824     0.772      1032
           3      0.810     0.582     0.677      1010
           4      0.852     0.687     0.761       982
           5      0.377     0.809     0.514       892
           6      0.997     0.359     0.528       958
           7      0.758     0.794     0.776      1028
           8      0.524     0.832     0.643       974
           9      0.621     0.772     0.688      1009

   micro avg      0.675     0.675     0.675     10000
   macro avg      0.764     0.675     0.676     10000
weighted avg      0.771     0.675     0.680     10000
```



8.1.4 Retain 95% Eigenvalues

```
In [156]: # 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(xtr_m, tr_vec, keep_dim)
          print(keep_dim)
```

```
153
```

In [157]:
```
# 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 [158]:
```
# 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 [159]:
```
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 [160]:
```
# 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 [161]:
```
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 8409 / 10000 correct
Accuracy = 0.840900
```

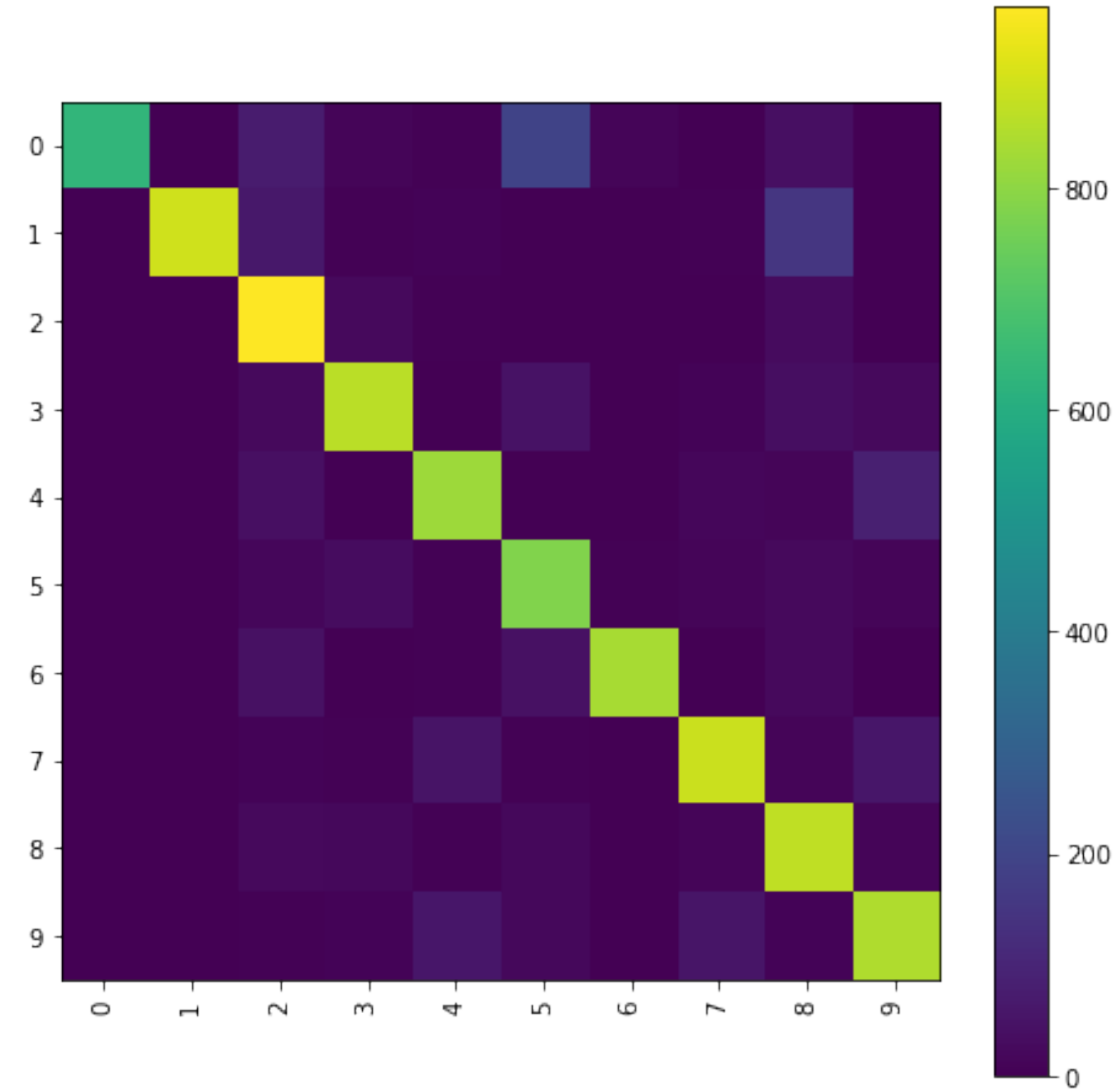
In [162]:
```
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred,
                            target_names=list(label_dict.values()),digits=3))
plt.figure(figsize=(8,8))
cnf_matrix = confusion_matrix(y_test, y_pred)
classes = list(label_dict.values())
plt.imshow(cnf_matrix, interpolation='nearest')
plt.colorbar()
tick_marks = np.arange(len(classes))
_ = plt.xticks(tick_marks, classes, rotation=90)
_ = plt.yticks(tick_marks, classes)
```

```
[[636   0  74  15   6 195  12   1  41   0]
 [  0 895  62   7  11   0   0   6 154   0]
 [  0   0 964  24   7   3   0   3  28   3]
 [  0   0  23 865   1  48   1  11  36  25]
 [  0   0  41   2 821   1   0  18  13  86]
 [  1   0  16  31   4 783   4  14  25  14]
 [  0   0  43   1   7  45 836   0  26   0]
```

```
 [  0   0   8   7  52   5   0 886  13  57]
 [  1   0  26  20   5  19   3  14 873  13]
 [  0   0   7  10  57  19   0  55  11 850]]
              precision    recall  f1-score   support

           0      0.997     0.649     0.786       980
           1      1.000     0.789     0.882      1135
           2      0.763     0.934     0.840      1032
           3      0.881     0.856     0.868      1010
           4      0.846     0.836     0.841       982
           5      0.700     0.878     0.779       892
           6      0.977     0.873     0.922       958
           7      0.879     0.862     0.870      1028
           8      0.716     0.896     0.796       974
           9      0.811     0.842     0.826      1009

   micro avg      0.841     0.841     0.841     10000
   macro avg      0.857     0.842     0.841     10000
weighted avg      0.860     0.841     0.842     10000
```



8.2 PCA + KNN

In [53]:
```
# 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 / 255.0
x_test = x_test / 255.0
# reconstruct dataset and stack them as a big one for dimension deduction
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 [54]:
```
# 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, xtr_m, tr_vec, keep_dim):
        x_proj= np.dot(xtr_m, tr_vec.T[:,0:keep_dim])
        return x_proj
```

In [55]:
```
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)
```

8.2.1 Retain 90% Eigenvalues

```
In [56]: # 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(xtr_m, tr_vec, keep_dim)
```

```
print(keep_dim)
```

```
86
```

```
In [57]: # 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 [58]: # select best k
         K=[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)))
```

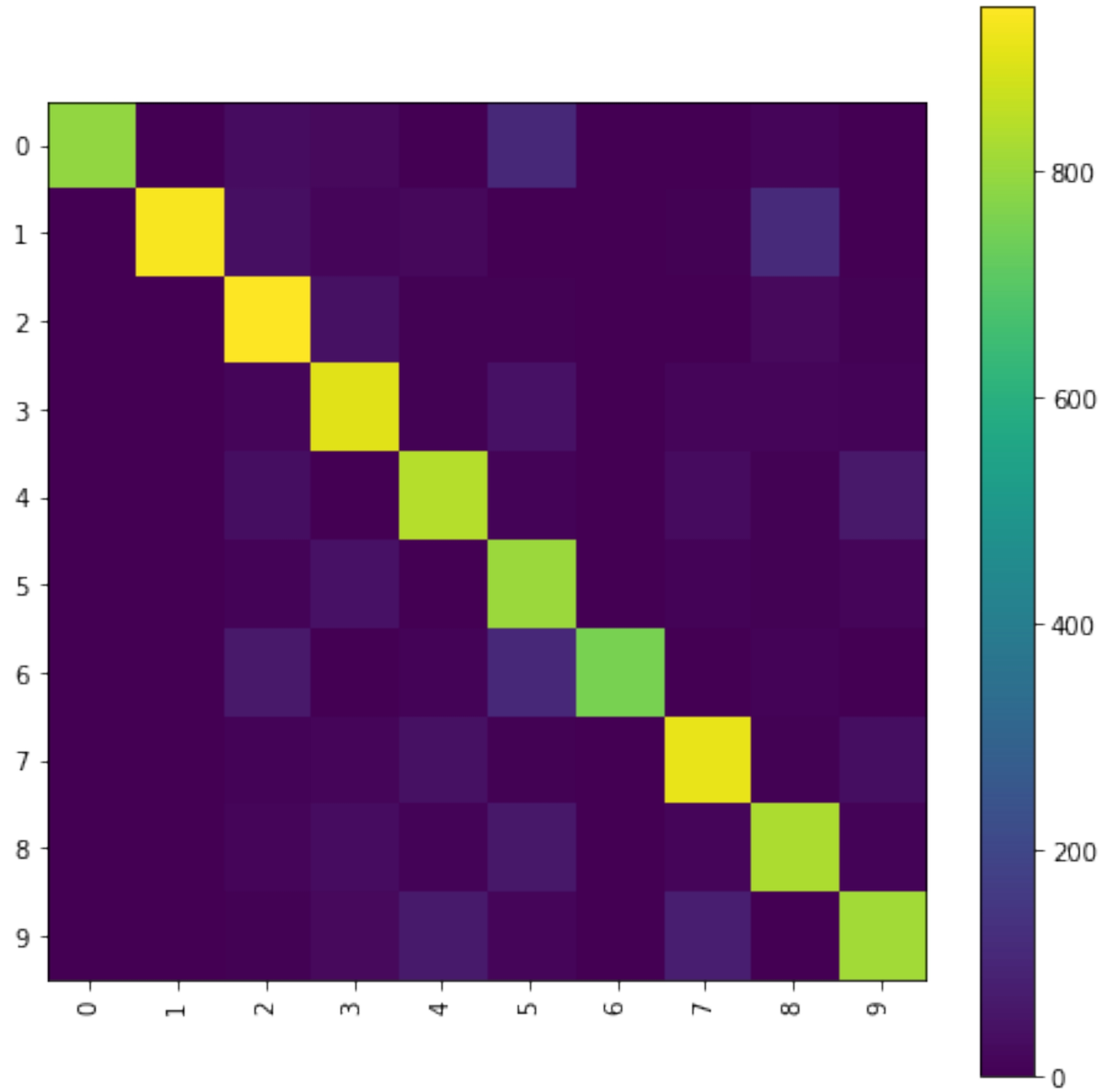
```
Got 7955 / 10000 correct
k = 3, Accuracy = 0.795500
Got 8035 / 10000 correct
k = 5, Accuracy = 0.803500
```

```
In [59]: Y_test_pred=classifier.predict(x_test, k=5)
         num_correct = np.sum(Y_test_pred == y_test)
         print('Got %d / %d correct' % (num_correct, num_test))
         print('k = %s, Accuracy = %f' % (5, np.mean(y_test == Y_test_pred)))
         print(confusion_matrix(y_test, Y_test_pred))
         print(classification_report(y_test, Y_test_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)
```

```
Got 8035 / 10000 correct
k = 5, Accuracy = 0.803500
[[ 892    0    3   20    1   13   38    0   13    0]
 [   0 1110    7    2    1    1    6    4    2    2]
 [  25   32  824   29   11   16   49    3   41    2]
 [  36    1   19  778    5  106   16    6   32   11]
 [   2   16   22    2  805    5   10   19    5   96]
 [  34   10    8  205   12  541   18   13   33   18]
 [  43    5   12    6    5    3  876    0    8    0]
 [   1   30    6    5   44    5    2  780    6  149]
 [  50   27   18   31    7   33   33   12  751   12]
 [   3   13    6   25   87    6    4  183    4  678]]
              precision    recall  f1-score   support

           0      0.821     0.910     0.864       980
           1      0.892     0.978     0.933      1135
           2      0.891     0.798     0.842      1032
           3      0.705     0.770     0.736      1010
           4      0.823     0.820     0.821       982
           5      0.742     0.607     0.667       892
           6      0.833     0.914     0.872       958
           7      0.765     0.759     0.762      1028
           8      0.839     0.771     0.804       974
           9      0.700     0.672     0.686      1009

   micro avg      0.803     0.803     0.803     10000
   macro avg      0.801     0.800     0.799     10000
weighted avg      0.803     0.803     0.801     10000
```



8.2.2 Retain 95% Eigenvalues

```
In [61]: # 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(xtr_m, tr_vec, keep_dim)
         print(keep_dim)
```

```
153
```

```
In [62]: # 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 [63]: # select best k
         K=[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 9146 / 10000 correct
k = 3, Accuracy = 0.914600
Got 9142 / 10000 correct
k = 5, Accuracy = 0.914200
```

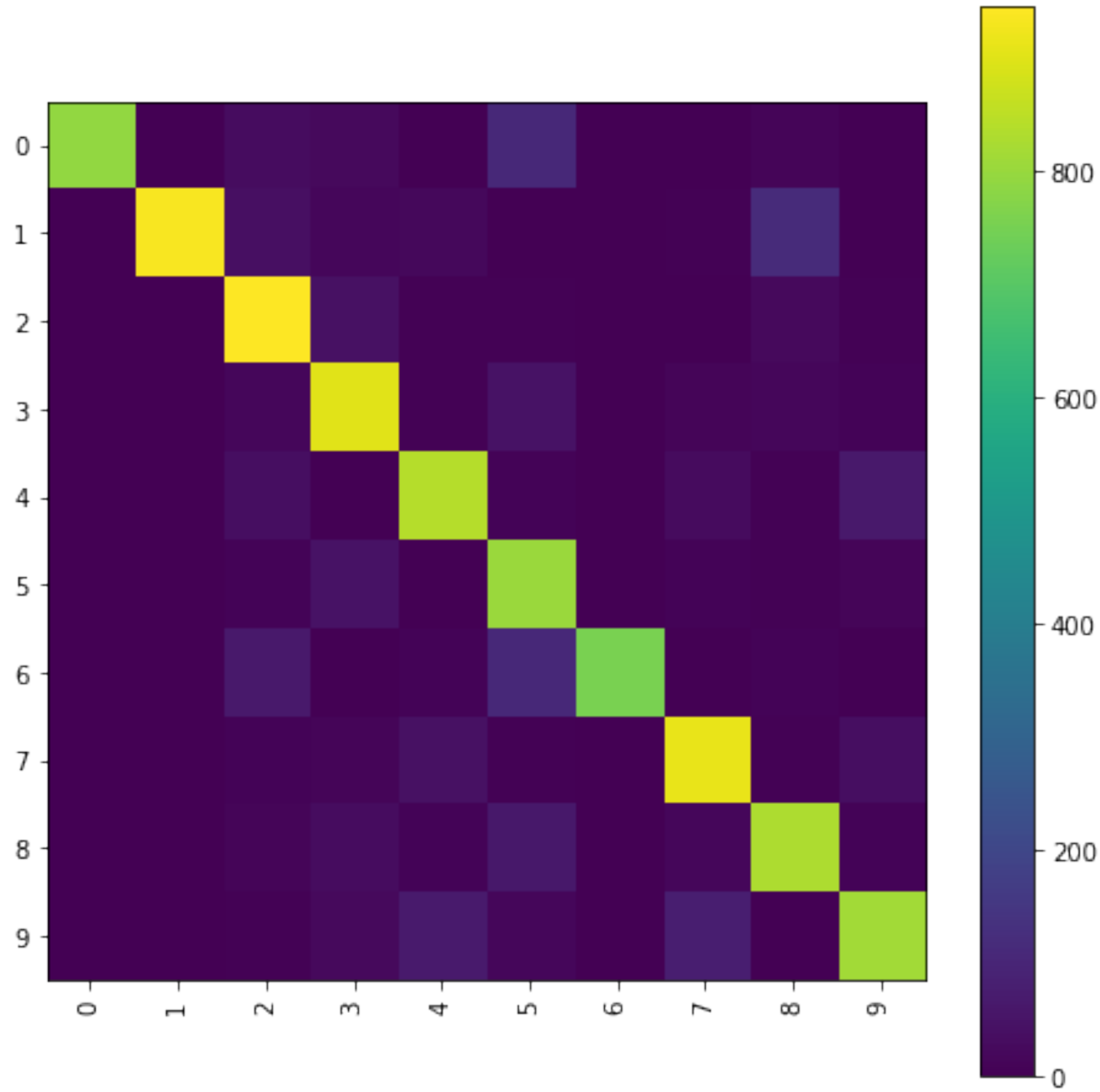
```
In [66]: Y_test_pred=classifier.predict(x_test, k=3)
         num_correct = np.sum(Y_test_pred == y_test)
         print('Got %d / %d correct' % (num_correct, num_test))
         print('k = %s, Accuracy = %f' % (3, np.mean(y_test == Y_test_pred)))
         print(confusion_matrix(y_test, Y_test_pred))
         print(classification_report(y_test, Y_test_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)
```

```
Got 9146 / 10000 correct
k = 3, Accuracy = 0.914600
[[ 960    0    0    4    0    0   12    0    4    0]
 [   0 1125    4    2    0    1    1    1    1    0]
```

```
 [  17   11  952   11    5    4    9    7   13    3]
 [  14    2   10  899    2   55    1    4   13   10]
 [   1    9    5    0  887    0    7   10    0   63]
 [  14    5    1   92    7  731   12    4   16   10]
 [  20    4    2    1    6    3  920    0    2    0]
 [   0   27    9    0   18    0    1  925    1   47]
 [  26    8    6   13    4   20   10    6  872    9]
 [   4   10    4   18   34    6    2   54    2  875]]
              precision    recall  f1-score   support

           0      0.909     0.980     0.943       980
           1      0.937     0.991     0.963      1135
           2      0.959     0.922     0.940      1032
           3      0.864     0.890     0.877      1010
           4      0.921     0.903     0.912       982
           5      0.891     0.820     0.854       892
           6      0.944     0.960     0.952       958
           7      0.915     0.900     0.907      1028
           8      0.944     0.895     0.919       974
           9      0.860     0.867     0.864      1009

   micro avg      0.915     0.915     0.915     10000
   macro avg      0.914     0.913     0.913     10000
weighted avg      0.915     0.915     0.914     10000
```



8.3 PCA + SVM

In [58]:
```
(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 / 255.0
X_test = 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)
big_X=np.vstack((x_train,x_test))
```

In [59]:
```
# build PCA class
```

```python
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, xtr_m, tr_vec, keep_dim):
        x_proj= np.dot(xtr_m, tr_vec.T[:,0:keep_dim])
        return x_proj
```

8.3.1 Retain 90% Eigenvalues

In [60]:
```python
# 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(xtr_m, tr_vec, keep_dim)
print(keep_dim)
```

86

```
In [61]: # 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 [62]: X_train = x_train.tolist()
         Y_train = y_train.tolist()
         X_test  = x_test.tolist()
         Y_test  = y_test.tolist()
```

Linear Kernel

```
In [63]: C=[53.95,54,54.1,54.3,54.5]
         prob = svm_problem(Y_train[0:20000], X_train[0:20000])
         param1 = svm_parameter('-t 0 -c 53.95 -b 1')
         param2 = svm_parameter('-t 0 -c 54 -b 1')
         param3 = svm_parameter('-t 0 -c 54.1 -b 1')
         param4 = svm_parameter('-t 0 -c 54.3 -b 1')
         param5 = svm_parameter('-t 0 -c 54.5 -b 1')
```

```
In [9]: P=[param1,param2,param3,param4,param5]
        for i in range(len(P)):
            model = svm_train(prob, P[i])
            p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
            print(p_acc)
```

```
Model supports probability estimates, but disabled in predicton.
Accuracy = 79.28% (7928/10000) (classification)
(79.28, 3.2961, 0.6453380439059659)
Model supports probability estimates, but disabled in predicton.
Accuracy = 79.3% (7930/10000) (classification)
(79.3, 3.2981, 0.6451649241897514)
Model supports probability estimates, but disabled in predicton.
Accuracy = 79.28% (7928/10000) (classification)
(79.28, 3.2994, 0.6450263724846607)
Model supports probability estimates, but disabled in predicton.
Accuracy = 79.29% (7929/10000) (classification)
(79.29, 3.2943, 0.6454761151052784)
Model supports probability estimates, but disabled in predicton.
```

```
Accuracy = 79.28% (7928/10000) (classification)
(79.28, 3.297, 0.6452699412369616)
```

```
In [64]: param2 = svm_parameter('-t 0 -c 54 -b 1')
         model = svm_train(prob, param2)
         p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
```

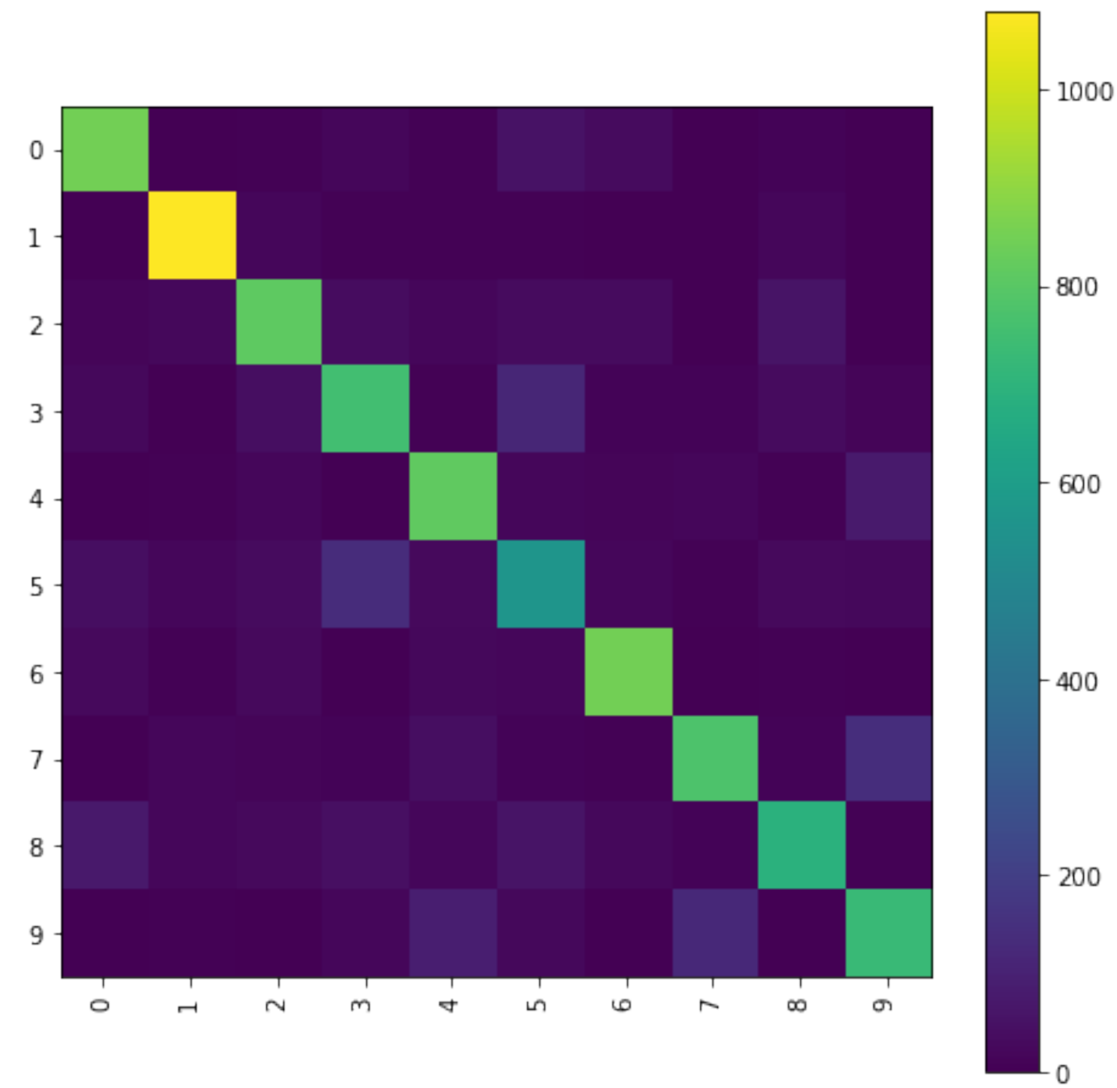
```
Model supports probability estimates, but disabled in predicton.
Accuracy = 79.3% (7930/10000) (classification)
```

```
In [65]: y_pred=p_label
         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)))
         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)
```

```
Got 7930 / 10000 correct
Accuracy = 0.793000
[[ 850    1    7   18    6   54   32    0   12    0]
 [   0 1080   17    5    5    6    3    2   17    0]
 [  15   23  814   35   19   32   33    3   55    3]
 [  25    2   38  755    6  116   10    9   33   16]
 [   4    7   18    4  816   18   13   20    6   76]
 [  38   20   33  138   26  562   18    6   29   22]
 [  27    7   26    1   22   19  851    0    5    0]
 [   0   18   14   10   39   11    5  780    9  142]
 [  75   21   26   45   20   55   25   11  691    5]
 [   4    7    3   20   90   23    1  126    4  731]]
              precision    recall  f1-score   support

           0      0.819     0.867     0.842       980
           1      0.911     0.952     0.931      1135
           2      0.817     0.789     0.803      1032
           3      0.732     0.748     0.740      1010
           4      0.778     0.831     0.804       982
           5      0.627     0.630     0.629       892
```

6	0.859	0.888	0.873	958
7	0.815	0.759	0.786	1028
8	0.803	0.709	0.753	974
9	0.735	0.724	0.730	1009
micro avg	0.793	0.793	0.793	10000
macro avg	0.790	0.790	0.789	10000
weighted avg	0.793	0.793	0.792	10000



RBF Kernel

```
In [67]: C=[121,124,127,129]
         prob = svm_problem(Y_train[0:20000], X_train[0:20000])
         param1 = svm_parameter('-t 2 -c 121 -b 1')
         param2 = svm_parameter('-t 2 -c 124 -b 1')
         param3 = svm_parameter('-t 2 -c 127 -b 1')
         param4 = svm_parameter('-t 2 -c 129 -b 1')
```

```
In [13]: P=[param1,param2,param3,param4]
         for i in range(len(P)):
             model = svm_train(prob, P[i])
             p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
             print(p_acc)
```

```
Model supports probability estimates, but disabled in predicton.
Accuracy = 83.39% (8339/10000) (classification)
(83.39, 2.7104, 0.7036783550745542)
Model supports probability estimates, but disabled in predicton.
Accuracy = 83.43% (8343/10000) (classification)
(83.43, 2.7022, 0.7045479398580405)
Model supports probability estimates, but disabled in predicton.
Accuracy = 83.42% (8342/10000) (classification)
(83.42, 2.7038, 0.7043683398516122)
Model supports probability estimates, but disabled in predicton.
Accuracy = 83.45% (8345/10000) (classification)
(83.45, 2.706, 0.704216641982182)
```

```
In [68]: param4 = svm_parameter('-t 2 -c 129 -b 1')
         model = svm_train(prob, param4)
         p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
```

```
Model supports probability estimates, but disabled in predicton.
Accuracy = 83.45% (8345/10000) (classification)
```

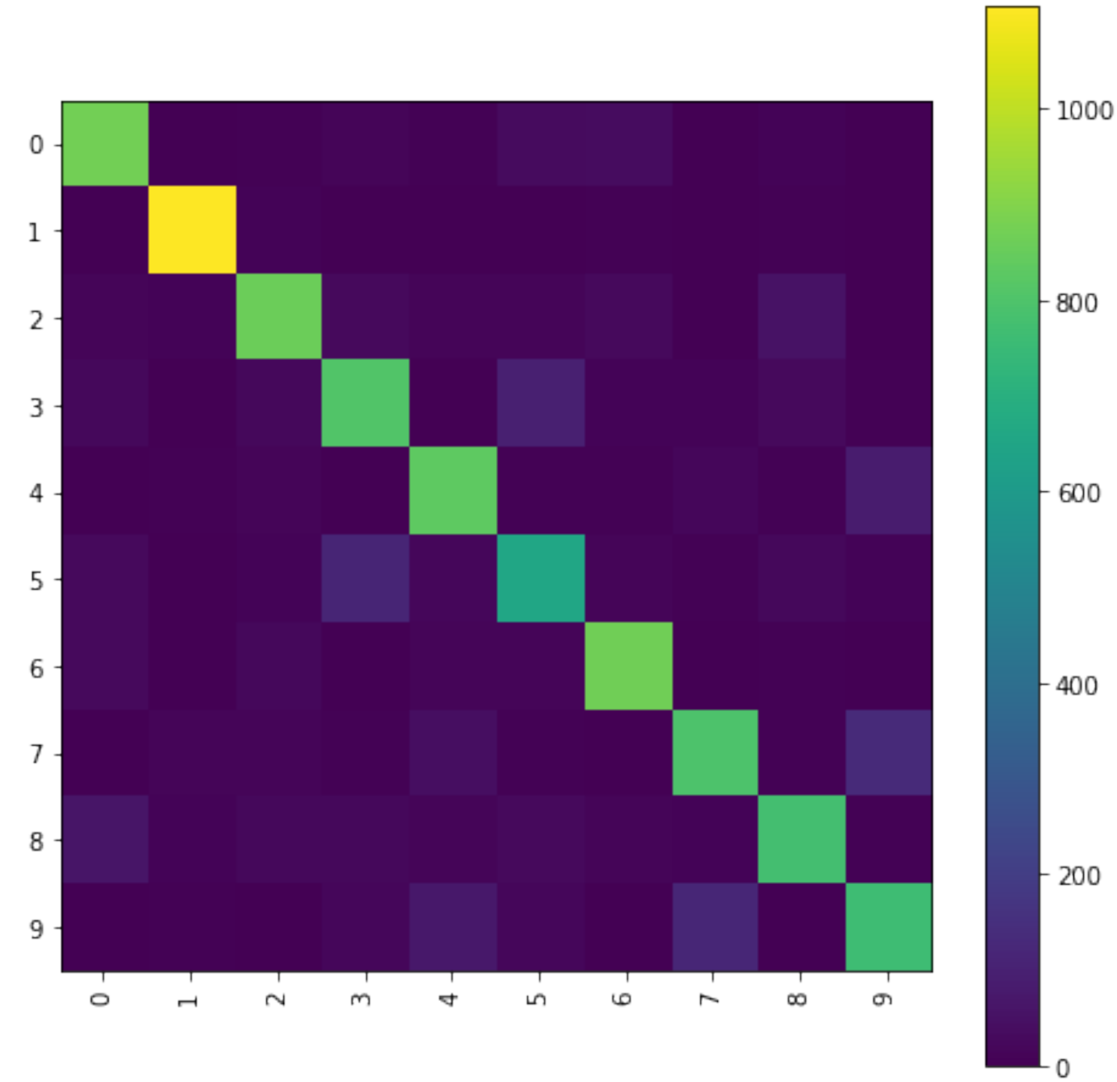
```
In [69]: y_pred=p_label
         num_test = len(Y_test)
         num_correct = np.sum(y_pred == y_test)
         print('Got %d / %d correct' % (num_correct, num_test))
         print('Accuracy = %f' % (np.mean(y_test == y_pred)))
         print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, 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)
```

```
Got 8345 / 10000 correct
Accuracy = 0.834500
[[ 872    0    7   15    7   31   36    0   12    0]
 [   0 1108   10    2    2    1    5    2    5    0]
 [  16   11  859   29   14   17   28    4   52    2]
 [  22    0   22  809    3   98   10    9   29    8]
 [   4    8   16    1  831    6    8   18    6   84]
 [  30    4   11  113   21  656   16    7   24   10]
 [  27    3   23    1   16   15  868    0    5    0]
 [   1   17   15    6   41    6    1  800    7  134]
 [  64   11   23   22   16   27   16   11  777    7]
 [   4    7    3   19   71   19    2  117    2  765]]
              precision    recall  f1-score   support

           0      0.838     0.890     0.863       980
           1      0.948     0.976     0.962      1135
           2      0.869     0.832     0.850      1032
           3      0.795     0.801     0.798      1010
           4      0.813     0.846     0.829       982
           5      0.749     0.735     0.742       892
           6      0.877     0.906     0.891       958
           7      0.826     0.778     0.802      1028
           8      0.845     0.798     0.821       974
           9      0.757     0.758     0.758      1009

   micro avg      0.835     0.835     0.835     10000
   macro avg      0.832     0.832     0.832     10000
weighted avg      0.834     0.835     0.834     10000
```



8.3.2 Retain 95% Eigenvalues

```
In [16]: # 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(xtr_m, tr_vec, keep_dim)
         print(keep_dim)
```

```
153
```

```
In [17]: # 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 [18]: X_train = x_train.tolist()
         Y_train = y_train.tolist()
         X_test  = x_test.tolist()
         Y_test  = y_test.tolist()
```

Linear Kernel

```
In [19]: C=[9.03,9.05,9.055,9.06]
         prob = svm_problem(Y_train[0:20000], X_train[0:20000])
         param1 = svm_parameter('-t 0 -c 9.03 -b 1')
         param2 = svm_parameter('-t 0 -c 9.05 -b 1')
         param3 = svm_parameter('-t 0 -c 9.055 -b 1')
         param4 = svm_parameter('-t 0 -c 9.06 -b 1')
```

```
In [20]: P=[param1,param2,param3,param4]
         for i in range(len(P)):
             model = svm_train(prob, P[i])
             p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
             print(p_acc)
```

```
Model supports probability estimates, but disabled in predicton.
Accuracy = 89.74% (8974/10000) (classification)
(89.74, 1.6183, 0.8153624855084941)
Model supports probability estimates, but disabled in predicton.
Accuracy = 89.74% (8974/10000) (classification)
(89.74, 1.6169, 0.8155189255113864)
Model supports probability estimates, but disabled in predicton.
Accuracy = 89.74% (8974/10000) (classification)
(89.74, 1.6169, 0.8155189255113864)
Model supports probability estimates, but disabled in predicton.
Accuracy = 89.73% (8973/10000) (classification)
(89.73, 1.6205, 0.8151134658623745)
```

```
In [49]: param2 = svm_parameter('-t 0 -c 9.05 -b 1')
         model = svm_train(prob, param2)
         p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
```

```
Model supports probability estimates, but disabled in predicton.
Accuracy = 89.74% (8974/10000) (classification)
```

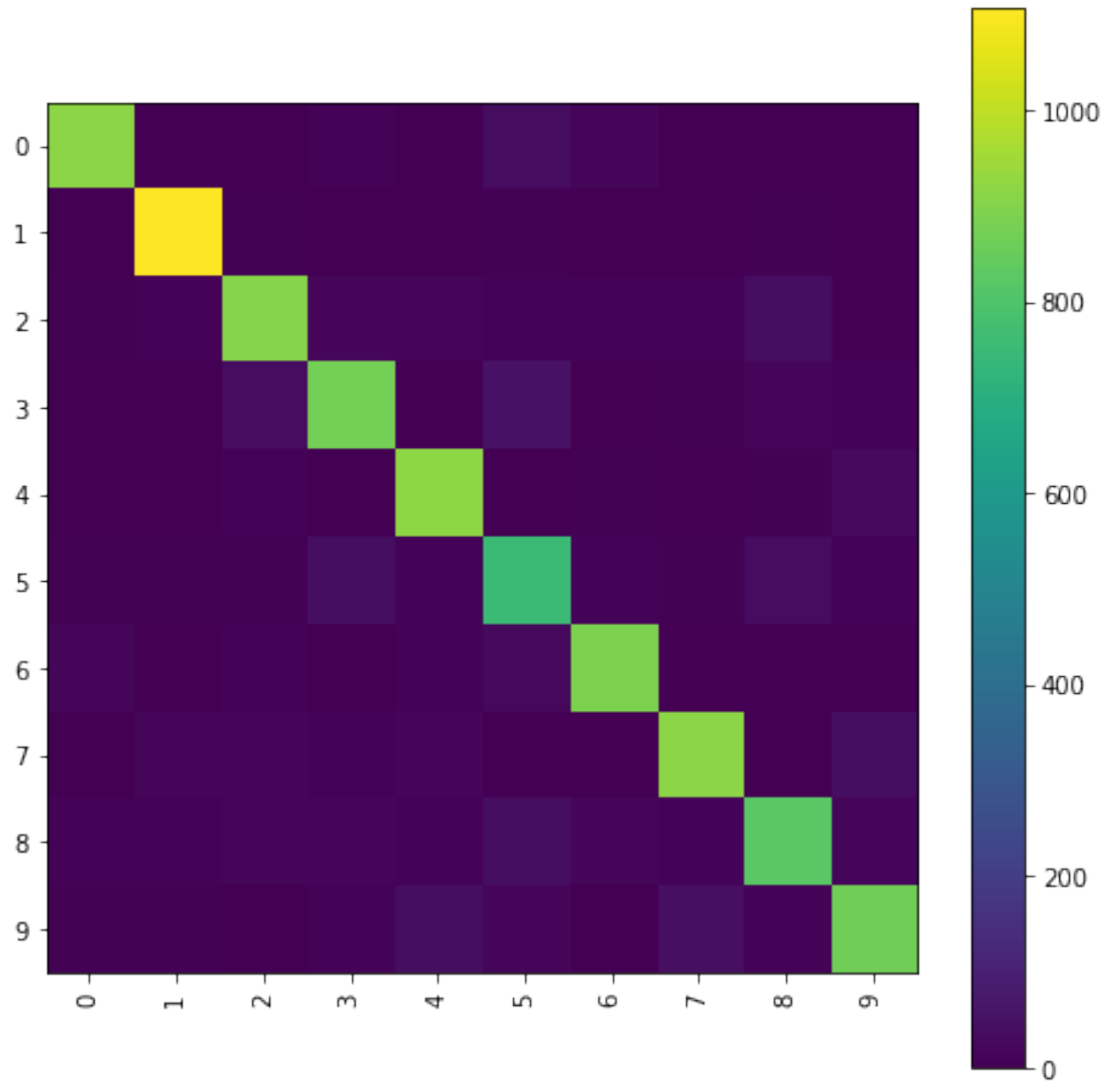
```
In [50]: y_pred=p_label
         num_test = len(Y_test)
         num_correct = np.sum(y_pred == y_test)
         print('Got %d / %d correct' % (num_correct, num_test))
         print('Accuracy = %f' % (np.mean(y_test == y_pred)))
         print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, 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)
```

```
Got 8974 / 10000 correct
Accuracy = 0.897400
[[ 914    0    3    9    1   36   14    1    2    0]
 [    0 1107    8    2    0    5    2    4    7    0]
 [    8   11  908   19   13   10   10   10   41    2]
 [    3    1   36  875    2   55    3    6   19   10]
 [    3    1   11    0  918    4    6    4    6   29]
 [    8    6    6   41   10  756   12    7   35   11]
 [   16    2   10    3   12   23  890    0    2    0]
 [    1   14   20   10   17    4    1  915    3   43]
 [   12   11   16   18   10   42   14    9  822   20]
 [    7    7    1   12   40   17    1   44   11  869]]
              precision    recall  f1-score   support

           0      0.940     0.933     0.936       980
           1      0.954     0.975     0.965      1135
           2      0.891     0.880     0.885      1032
           3      0.885     0.866     0.875      1010
           4      0.897     0.935     0.916       982
           5      0.794     0.848     0.820       892
           6      0.934     0.929     0.931       958
           7      0.915     0.890     0.902      1028
           8      0.867     0.844     0.855       974
           9      0.883     0.861     0.872      1009

   micro avg      0.897     0.897     0.897     10000
   macro avg      0.896     0.896     0.896     10000
```

```
weighted avg       0.898      0.897      0.897      10000
```



RBF Kernel

```
In [43]: C=[114,120,125,129]
         prob = svm_problem(Y_train[0:20000], X_train[0:20000])
         param1 = svm_parameter('-t 2 -c 114 -b 1')
         param2 = svm_parameter('-t 2 -c 120 -b 1')
         param3 = svm_parameter('-t 2 -c 125 -b 1')
         param4 = svm_parameter('-t 2 -c 129 -b 1')
```

```
In [44]: P=[param1,param2,param3,param4]
         for i in range(len(P)):
             model = svm_train(prob, P[i])
             p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
             print(p_acc)
```

```
Model supports probability estimates, but disabled in predicton.
Accuracy = 92.14% (9214/10000) (classification)
(92.14, 1.2379, 0.8573398319503869)
Model supports probability estimates, but disabled in predicton.
Accuracy = 92.14% (9214/10000) (classification)
(92.14, 1.241, 0.8570281706490617)
Model supports probability estimates, but disabled in predicton.
Accuracy = 92.19% (9219/10000) (classification)
(92.19000000000001, 1.2275, 0.8585118655121964)
Model supports probability estimates, but disabled in predicton.
Accuracy = 92.18% (9218/10000) (classification)
(92.17999999999999, 1.2298, 0.8582607124236713)
```

```
In [46]: param2 = svm_parameter('-t 2 -c 125 -b 1')
         model = svm_train(prob, param2)
         p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
```

```
Model supports probability estimates, but disabled in predicton.
Accuracy = 92.19% (9219/10000) (classification)
```

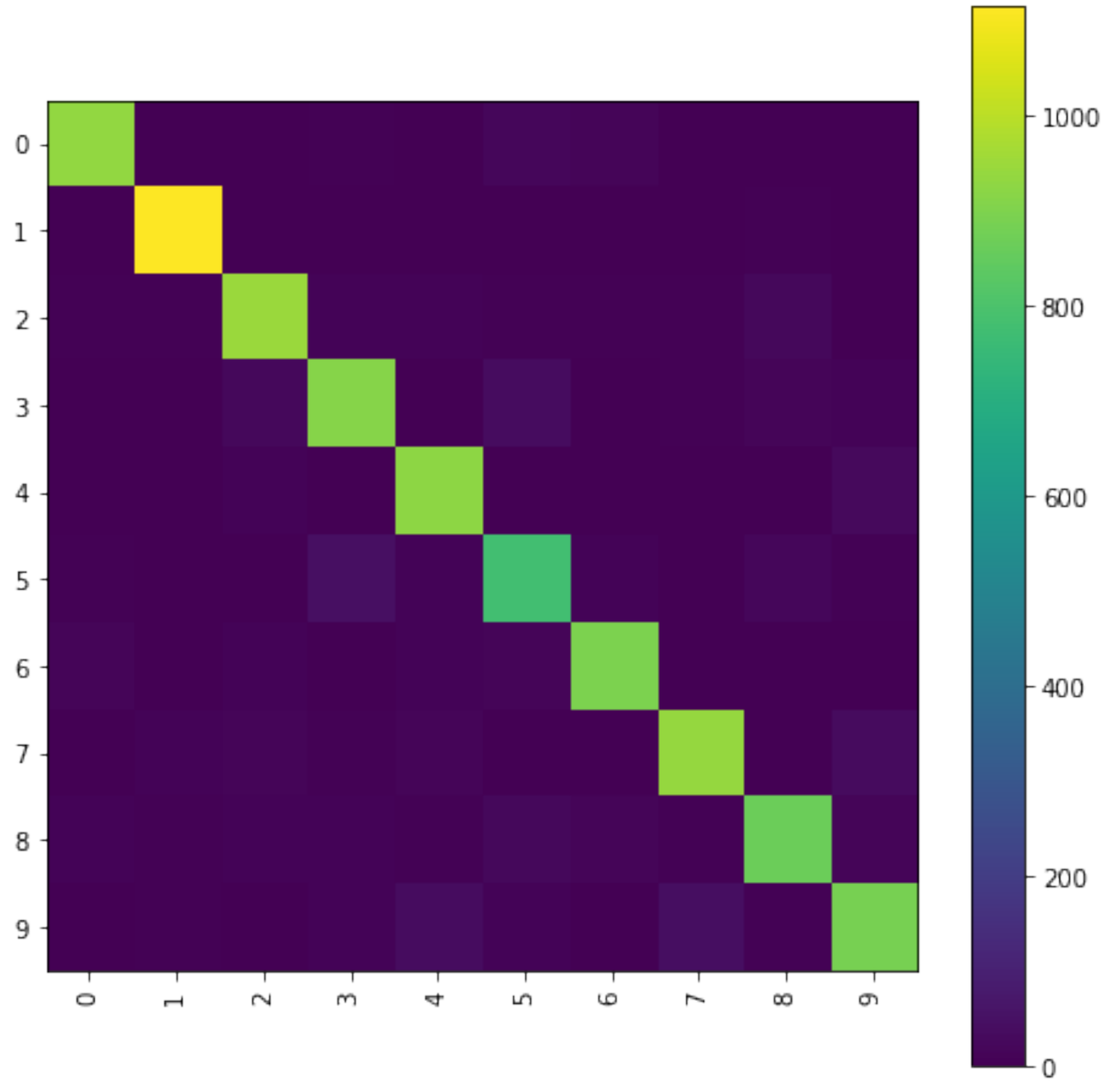
```
In [47]: y_pred=p_label
         num_test = len(Y_test)
         num_correct = np.sum(y_pred == y_test)
         print('Got %d / %d correct' % (num_correct, num_test))
         print('Accuracy = %f' % (np.mean(y_test == y_pred)))
         print(confusion_matrix(y_test, y_pred))
         print(classification_report(y_test, 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)
```

```
Got 9219 / 10000 correct
Accuracy = 0.921900
[[ 935    0    1    8    0   20   14    1    1    0]
 [   0 1117    4    2    1    3    1    2    5    0]
```

```
[   6    6  949   11   12    6    8    6   26    2]
[   3    1   22  913    2   36    2    7   14   10]
[   2    0    9    1  929    3    3    3    3   29]
[   8    3    4   44   10  781   12    3   19    8]
[  15    2    9    1   13   17  898    0    3    0]
[   0   12   17    5   14    1    1  942    4   32]
[  12    6    9   11    8   25   15    7  867   14]
[   2    6    2   13   36   11    2   42    7  888]]
              precision    recall  f1-score   support

           0      0.951     0.954     0.953       980
           1      0.969     0.984     0.976      1135
           2      0.925     0.920     0.922      1032
           3      0.905     0.904     0.904      1010
           4      0.906     0.946     0.926       982
           5      0.865     0.876     0.870       892
           6      0.939     0.937     0.938       958
           7      0.930     0.916     0.923      1028
           8      0.914     0.890     0.902       974
           9      0.903     0.880     0.892      1009

   micro avg      0.922     0.922     0.922     10000
   macro avg      0.921     0.921     0.921     10000
weighted avg      0.922     0.922     0.922     10000
```


0
1
2
3
4
5
6
7
8
9
0
1
2
3
4
5
6
7
8
9
1000
800
600
400
200
0