Machine Learning and Computational Statistics

Project

> Konstantinos Kyrkos
> Registration Number p3351907
> kkyrkos@aueb.gr

This notebook provides my approach for the Project on the course "Machine Learning and Computational Statistics" for the Part time Data Science Program (2019).
Along with this file we will provide a file named "Project-Kyrkos.pdf" which will be the same document but in pdf format, to be more readable.

Before proceeding with the project implementation we begin by importing the libraries we will later use.

In [1]:

```
import numpy as np
import pandas as pd
import sys
import matplotlib.pyplot as plt
import scipy.io as sio
import numpy.matlib
import scipy.optimize
import matplotlib.image as mpimg
from scipy.spatial import distance
from mpl_toolkits.mplot3d import Axes3D
from scipy.stats import multivariate_normal, norm
from sklearn import mixture
from scipy.optimize import minimize
from scipy.optimize import nnls
from sklearn.linear_model import Lasso
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import cross_val_score
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import KFold
from numpy import linalg as LA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics.pairwise import euclidean_distances
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

%matplotlib inline
```

Next we load our data using the hints provided in the code which was given along the project and print them to understand what we are dealing with.

In [2]:

```
Pavia = sio.loadmat('PaviaU_cube.mat')
HSI = Pavia['X']
print(HSI)
```

```
[[[ 903  763  658 ... 2988 3036 3045]
  [ 817  937  864 ... 3412 3484 3503]
  [ 811  857  905 ... 2965 2977 2983]
  ...
  [1088 1141 1163 ... 1275 1253 1257]
  [1264 1244 1023 ... 1251 1213 1240]
  [1370 1262 1222 ... 1277 1295 1336]]

 [[1234  967  733 ... 2746 2779 2843]
  [ 740  930  923 ... 2641 2711 2737]
  [1292 1106  923 ... 2314 2364 2389]
```

```
 ...
 [1143 1095 1131 ... 1300 1299 1327]
 [1498 1302 1199 ... 1298 1262 1286]
 [ 777  573  844 ... 1292 1316 1340]]

[[1377 1269  987 ... 2617 2663 2727]
 [1157  953  836 ... 2410 2429 2444]
 [1268 1209  997 ... 2205 2229 2236]
 ...
 [1151 1235 1270 ... 1433 1427 1440]
 [1478 1389 1299 ... 1330 1315 1308]
 [1170 1334 1238 ... 1295 1230 1217]]

...

[[ 649  597  635 ... 2665 2736 2786]
 [ 853  886  718 ... 3476 3573 3594]
 [ 846  584  347 ... 3586 3589 3599]
 ...
 [ 519  436  479 ... 3857 3792 3831]
 [ 744  373  136 ... 3472 3467 3495]
 [ 708  374  518 ... 3461 3498 3575]]

[[ 617  431  570 ... 2777 2804 2793]
 [ 735  576  562 ... 3648 3731 3765]
 [ 777  686  521 ... 3611 3654 3706]
 ...
 [ 593  421  370 ... 3722 3722 3859]
 [ 634  549  492 ... 3467 3469 3504]
 [ 847  485  381 ... 3184 3193 3230]]

[[ 896  521  339 ... 2554 2647 2661]
 [ 689  765  776 ... 3334 3409 3465]
 [ 551  557  353 ... 3347 3397 3394]
 ...
 [ 734  566  352 ... 3562 3512 3574]
 [ 614  640  582 ... 3338 3363 3430]
 [ 547  583  462 ... 3016 3020 3078]]]
```

Just to be sure that what stated in the exercise is true, we get the two dimensions of our data and store them in variables for possible later use.

In [3]:

```
M = HSI.shape[0]
N = HSI.shape[1]
print("M size is: ",M)
print("N size is: ",N)
```

```
M size is:  300
N size is:  200
```

Next we load our endmembers and print their values. Each row for these vaues corresponds to each of the spectral bands.

In [4]:

```
ends = sio.loadmat('PaviaU_endmembers.mat')
endmembers = ends['endmembers']
print(endmembers)
```

```
[[1091.88718412  935.31299927 1260.85040495  689.59934319 2914.08309456
   895.41715976 1087.77969925 1245.21671309  642.72114137]
 [1053.39440433  857.3761801  1255.74416389  560.56157635 2994.92406877
   835.75772518 1084.84887218 1228.80668524  532.25162127]
 [1043.06588448  817.46114742 1238.04001906  484.6453202  3101.07163324
   798.19296515 1099.9112782  1233.85849582  435.59922179]
 [1073.37274368  820.48148148 1264.69938066  469.54351396 3342.28653295
   824.88527285 1147.01654135 1290.57604457  392.13488975]
 [1096.81768953  818.1815541  1291.67698904  455.98850575 3585.61318052
   855.34319527 1186.36541353 1335.91030641  373.07652399]
 [1101.86552347  807.85257807 1290.99285374  429.9408867  3755.61747851
   863.88856016 1201.69849624 1348.54150418  352.45006485]
```

```
 [1104.07761733  796.62745098 1295.09718914  408.19047619 3898.53868195
   870.18474688 1217.10902256 1356.49359331  334.82230869]
 [1094.05054152  776.33405955 1284.23058599  383.47619048 4009.76074499
   873.90828402 1230.83834586 1353.56211699  311.35278859]
 [1090.47833935  772.07262164 1276.52215341  363.24794745 4127.06017192
   880.26890204 1244.66766917 1358.89972145  289.54863813]
 [1102.88718412  782.13435004 1287.75083373  351.94581281 4267.33524355
   891.3678501  1264.82631579 1381.04512535  275.17509728]
 [1117.82942238  789.5221496  1306.19485469  343.58949097 4400.83524355
   907.35470085 1282.96240602 1406.21727019  260.85214008]
 [1111.95577617  778.13507625 1300.3268223   324.09359606 4469.29512894
   913.57133465 1287.66766917 1406.87465181  239.78339818]
 [1112.93050542  778.18228032 1298.67413054  310.18226601 4555.28510029
   921.1078238  1305.8        1419.28245125  228.85992218]
 [1120.46750903  787.77705156 1305.0028585   301.22988506 4632.67335244
   931.54996713 1325.27669173 1435.85682451  223.15175097]
 [1131.19043321  798.3761801  1322.01143402  296.01642036 4675.06733524
   943.36456279 1339.11428571 1452.2367688   214.72243839]
 [1137.71209386  804.68409586 1333.80323964  290.77832512 4696.55873926
   955.19230769 1345.90225564 1464.88969359  202.94811933]
 [1148.82851986  815.92374728 1347.78037161  291.38752053 4728.08595989
   976.16831032 1358.28045113 1484.42172702  194.50064851]
 [1158.52075812  829.01525054 1362.49880896  293.97865353 4745.6504298
   996.63708087 1367.47894737 1502.03064067  189.11024643]
 [1166.8465704   844.51924473 1380.70795617  300.5090312  4737.19054441
  1017.9556213  1369.14661654 1518.11253482  185.05966278]
 [1176.07761733  862.43645606 1400.2086708   312.56321839 4726.30515759
  1041.0190664  1373.0481203  1536.60947075  183.49286641]
 [1183.38808664  879.51633987 1412.87946641  328.84236453 4709.41690544
  1066.20907298 1376.84661654 1552.66462396  183.14656291]
 [1192.86642599  901.10094408 1430.76417342  353.93267652 4691.54727794
  1095.44115713 1381.72030075 1571.0183844   183.90661479]
 [1201.90252708  924.94408134 1457.61314912  386.40558292 4655.0286533
  1122.73438527 1384.76240602 1589.72311978  183.52918288]
 [1209.92238267  947.25562818 1477.82896617  421.48604269 4623.81661891
  1151.70512821 1387.49924812 1605.61949861  181.14656291]
 [1223.98375451  972.14088598 1500.48022868  457.93760263 4619.02722063
  1188.35963182 1397.59172932 1628.03509749  180.01037613]
 [1243.15974729 1003.12273057 1533.13911386  493.46141215 4620.8782235
  1230.73734385 1410.97218045 1659.01002786  180.86251621]
 [1247.86552347 1021.17066086 1549.39733206  516.61740558 4571.46704871
  1256.27284681 1410.91954887 1672.8724234   178.47470817]
 [1246.88357401 1033.17211329 1557.38351596  532.02791461 4497.16332378
  1273.73602893 1405.06541353 1678.00445682  174.46692607]
 [1247.59205776 1048.19753086 1572.43639828  544.08210181 4419.4469914
  1290.62393162 1400.84360902 1685.96100279  171.17898833]
 [1255.50180505 1070.70079884 1596.29252025  556.8226601  4360.0487106
  1314.34155161 1406.45789474 1705.17883008  169.74189364]
 [1266.09927798 1093.26143791 1620.00238209  566.46962233 4305.52292264
  1344.0581854  1414.36466165 1726.64289694  168.31128405]
 [1270.41606498 1110.61873638 1634.85802763  567.71921182 4222.95558739
  1363.8530572  1415.35338346 1739.72423398  166.16731518]
 [1274.45216606 1124.33986928 1648.54978561  560.50082102 4130.87106017
  1376.72649573 1416.25263158 1750.91086351  163.94811933]
 [1286.43592058 1143.32171387 1671.73272987  549.09688013 4051.04441261
  1393.72222222 1424.84135338 1770.95041783  162.44747082]
 [1297.17689531 1161.02251271 1693.87565507  531.72085386 3964.45272206
  1410.52728468 1431.28421053 1788.37103064  160.10116732]
 [1306.56768953 1177.02178649 1714.55597904  510.08866995 3869.34240688
  1423.88593031 1436.28195489 1804.58440111  157.85473411]
 [1313.68140794 1190.71169208 1729.9204383   487.36124795 3765.14326648
  1432.97205786 1441.57744361 1817.95487465  154.98962387]
 [1319.51263538 1204.2875817  1742.88804192  468.24630542 3646.89398281
  1441.5634451  1444.39548872 1828.59832869  151.68871595]
 [1322.49097473 1214.8097313  1746.75226298  452.23316913 3523.62893983
  1452.20118343 1444.96766917 1834.12479109  149.50064851]
 [1323.29151625 1225.28031954 1746.43449262  440.51724138 3388.80372493
  1462.90762656 1445.42030075 1836.89526462  148.65758755]
 [1324.21841155 1238.60929557 1756.42496427  434.32019704 3231.21919771
  1467.84516765 1443.71729323 1841.8718663   148.81582361]
 [1323.81949458 1249.86928105 1765.54216293  429.50738916 3070.02005731
  1469.67587114 1440.32255639 1844.68356546  147.81063554]
 [1325.97563177 1260.32098765 1771.36779419  424.52873563 2929.40974212
  1476.7452334  1440.09172932 1849.31866295  146.48378729]
 [1330.21570397 1272.77632534 1774.88232492  419.2955665  2810.46561605
  1487.55884287 1443.58796992 1854.82061281  146.04020752]
 [1335.97202166 1287.73420479 1778.21486422  413.23481117 2712.37965616
```

```
 1500.81328074 1449.10451128 1862.41114206  146.1232166 ]
 [1341.17599278 1301.01815541 1783.34016198  405.90147783 2632.39255014
  1511.59631821 1455.52330827 1868.89470752  146.95719844]
 [1342.44314079 1310.21423384 1786.4259171   397.57142857 2556.18767908
  1515.33366206 1456.82556391 1870.37548747  147.25162127]
 [1340.90523466 1316.3231663  1784.47927585  389.35960591 2488.45272206
  1515.48816568 1454.58045113 1867.50417827  146.17120623]
 [1342.41064982 1326.45098039 1783.79180562  385.13793103 2440.60458453
  1521.32741617 1455.52556391 1868.94373259  146.42931258]
 [1337.6200361  1332.49382716 1777.08575512  381.18062397 2390.25931232
  1519.46416831 1450.09022556 1862.15431755  146.20103761]
 [1342.04241877 1348.3449528  1787.35016675  381.52545156 2354.30945559
  1527.07626561 1451.90526316 1868.29359331  146.87808042]
 [1350.74638989 1364.52069717 1799.95283468  379.08538588 2332.3782235
  1540.28205128 1459.7962406  1879.11253482  146.85992218]
 [1354.18140794 1372.74437182 1799.97474988  370.54022989 2308.4713467
  1546.37475345 1463.76766917 1882.96044568  145.65758755]
 [1357.4467509  1382.08641975 1797.95998094  360.09688013 2283.55587393
  1552.30276134 1467.86315789 1887.38495822  144.22827497]
 [1355.20758123 1385.47857662 1792.20533587  348.69293924 2249.36962751
  1550.92241946 1465.74511278 1882.66183844  143.01686122]
 [1353.84386282 1390.27233115 1790.46641258  338.1001642  2221.19197708
  1546.95857988 1462.6518797  1879.51922006  141.65110246]
 [1357.56768953 1402.27596224 1797.86040972  329.09195402 2208.83954155
  1547.91321499 1465.37894737 1884.81615599  139.70817121]
 [1365.0532491  1417.68409586 1814.21867556  320.10837438 2210.98137536
  1551.52038133 1470.84135338 1895.1454039   136.93255512]
 [1369.01263538 1427.19970951 1817.66555503  308.33333333 2223.07879656
  1552.86489152 1473.73609023 1900.1637883   133.05836576]
 [1368.06768953 1430.65141612 1808.89899952  296.68472906 2238.22492837
  1551.83136095 1473.02706767 1895.98495822  129.93514916]
 [1371.35108303 1440.21859114 1808.47212959  290.85550082 2266.23352436
  1556.19165023 1476.21654135 1897.57604457  128.45654994]
 [1376.38267148 1455.05010893 1812.54930919  290.9047619  2301.83810888
  1565.68902038 1481.33233083 1902.98941504  128.24254215]
 [1378.66155235 1468.64052288 1812.84182944  295.45812808 2334.32951289
  1575.82971729 1482.51203008 1903.65738162  129.47341115]
 [1376.50180505 1479.4647785  1801.93377799  307.96880131 2359.9756447
  1586.75904011 1480.34511278 1897.54930362  131.76783398]
 [1372.64620939 1495.84313725 1794.11386374  341.57307061 2376.09312321
  1600.80374753 1475.19473684 1889.24233983  140.79766537]
 [1374.65703971 1529.66739288 1811.93901858  402.1001642  2391.20057307
  1624.1469428  1471.4924812  1892.78885794  150.05317769]
 [1379.48555957 1575.27523602 1831.60028585  487.84236453 2420.31805158
  1669.57495069 1471.79699248 1905.4189415   149.01037613]
 [1381.17870036 1620.66884532 1836.67889471  611.8045977  2446.44126074
  1728.64595661 1471.3481203  1911.30473538  151.60311284]
 [1390.55234657 1677.27087872 1852.38970939  776.29392447 2470.37535817
  1802.42537804 1475.23759398 1924.31086351  159.76653696]
 [1397.15613718 1730.81699346 1857.60362077  960.22003284 2478.3495702
  1877.53747535 1476.5406015  1932.17270195  168.26459144]
 [1398.11552347 1774.83660131 1842.25631253 1148.08538588 2472.89111748
  1950.52892834 1475.06015038 1929.72869081  176.62775616]
 [1395.11371841 1809.67320261 1815.19533111 1340.53037767 2452.3982808
  2021.04569362 1472.45112782 1917.40557103  188.57457847]
 [1396.42509025 1852.91866376 1811.15054788 1570.77504105 2410.81088825
  2083.3139382  1467.36466165 1910.83844011  208.96108949]
 [1403.1200361  1911.24691358 1831.36017151 1840.26929392 2367.87249284
  2141.49145299 1463.23007519 1917.7810585   228.47341115]
 [1411.46028881 1968.43572985 1849.2582182  2111.54515599 2343.51002865
  2201.1120973  1462.7593985  1927.59554318  242.55512322]
 [1426.2265343  2031.64778504 1873.64792758 2390.31198686 2343.81518625
  2266.48323471 1468.47894737 1945.79777159  253.81971466]
 [1434.83935018 2077.05083515 1881.24106717 2637.03940887 2356.57593123
  2323.19921105 1470.73233083 1956.71420613  259.50713359]
 [1443.41606498 2113.02251271 1881.01000476 2851.36453202 2385.80659026
  2377.84911243 1474.15864662 1965.43509749  266.29442283]
 [1447.33303249 2139.13362382 1877.14721296 3028.20032841 2416.83667622
  2415.17554241 1474.68345865 1967.15543175  274.46303502]
 [1451.31137184 2159.76833696 1872.41305384 3168.35303777 2453.73065903
  2443.60683761 1476.4887218  1967.29637883  282.84565499]
 [1459.22743682 2182.14088598 1866.76798475 3278.76026273 2499.01002865
  2477.02892834 1484.12255639 1971.25348189  291.72373541]
 [1457.83303249 2179.25853304 1832.42877561 3321.55336617 2530.36962751
  2500.42373439 1485.23834586 1957.93816156  300.8690013 ]
 [1449.99729242 2151.40740741 1779.91519771 3288.41707718 2532.32091691
  2491.99408284 1477.66842105 1925.46852368  324.769131  ]
```

```
 [1460.19765343 2166.79375454 1818.00905193 3293.61740558 2499.26217765
  2438.93425378 1474.1112782  1923.31086351  368.17380026]
 [1477.59837545 2238.79520697 1909.98141972 3426.15763547 2509.29226361
  2437.84582512 1478.00225564 1963.06908078  360.79507134]
 [1470.34837545 2255.17283951 1903.38113387 3483.72413793 2533.77936963
  2473.60026298 1474.93533835 1966.22506964  331.36705577]
 [1462.37454874 2252.69789397 1873.27632206 3488.47454844 2543.81375358
  2496.05489809 1471.87293233 1956.92534819  318.41634241]
 [1455.64079422 2248.54901961 1853.83134826 3477.18390805 2532.76361032
  2498.91584484 1465.79924812 1945.39777159  315.72373541]
 [1453.80505415 2250.54611474 1843.25678895 3469.75369458 2524.13323782
  2501.421762   1462.86541353 1937.21615599  317.00129702]
 [1458.80776173 2259.96151053 1841.34397332 3469.69293924 2529.68624642
  2507.5105194  1464.74210526 1936.58384401  321.58106355]
 [1469.18140794 2280.84676834 1849.74273464 3483.13957307 2549.99713467
  2520.34878369 1468.6962406  1944.21392758  329.9079118 ]
 [1468.32039711 2285.11111111 1847.63268223 3476.67980296 2561.87249284
  2517.04372124 1463.4887218  1939.01615599  334.94033722]
 [1461.63357401 2281.90196078 1837.04716532 3457.79310345 2573.64326648
  2505.85404339 1456.16616541 1928.27632312  336.13359274]
 [1463.62635379 2290.61873638 1835.4835636  3454.32512315 2608.77507163
  2509.93918475 1457.47819549 1926.84958217  339.8612192 ]
 [1465.17418773 2294.43209877 1826.0328728  3444.21018062 2647.75787966
  2517.234714   1456.80977444 1922.01281337  343.58106355]
 [1457.92689531 2278.12999274 1797.60457361 3399.52380952 2672.03868195
  2509.21729126 1449.87744361 1902.7810585   350.11802853]
 [1456.98375451 2270.17792302 1782.939495   3355.87684729 2693.51432665
  2498.78205128 1446.53157895 1892.90807799  366.74708171]
 [1464.52888087 2284.99491649 1800.76941401 3346.97701149 2719.87106017
  2486.66732413 1446.07293233 1899.4735376   383.85473411]
 [1462.15794224 2293.85766158 1807.01143402 3342.8045977  2745.85530086
  2472.08908613 1439.03233083 1897.99554318  384.79377432]
 [1454.32310469 2297.76470588 1793.63030014 3335.49589491 2774.87106017
  2466.28270874 1431.33308271 1890.79331476  377.00518807]
 [1443.3601083  2303.35439361 1786.46021915 3331.65845649 2789.6504298
  2455.43293886 1420.57669173 1882.08635097  368.84824903]
 [1444.95036101 2325.04066812 1806.80943306 3366.75369458 2822.9713467
  2463.26397107 1418.08796992 1887.0551532   364.16990921]
 [1450.17779783 2337.30428468 1816.2939495  3397.45320197 2874.86819484
  2483.343524   1423.13759398 1893.23231198  361.10894942]]
```

In [5]:

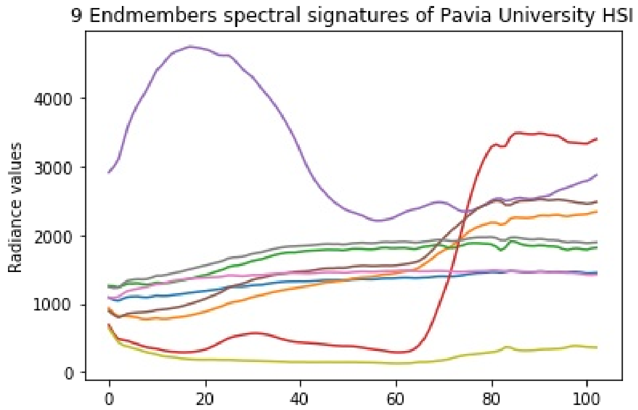
```python
print("The size of our endmembers is: ",len(endmembers))
```

```
The size of our endmembers is:  103
```

We next print our Endmembers to have a look at our labels.

In [5]:

```python
fig = plt.figure()
plt.plot(endmembers)
plt.ylabel('Radiance values')
plt.xlabel('Spectral bands')
plt.title('9 Endmembers spectral signatures of Pavia University HSI')
plt.show()
```



Spectral bands

At this point we can declare some usefull variables which we will later use multiple times.

In [6]:

```
L = HSI.shape[2]
pixels= M*N
```

Afte that we load the class label for each of our pixels and print these data to have a look at them. We also reshape we data here. The reshaping was done because this new shape of our data structure was much easier to handle and iterate than before. The same technique will be later used again.

In [8]:

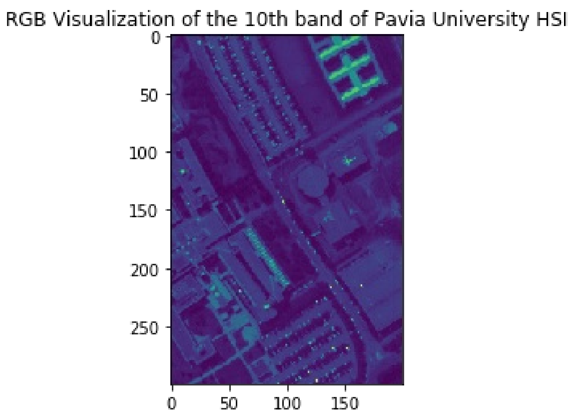
```
ground_truth = sio.loadmat('PaviaU_ground_truth.mat')
truth = ground_truth['y']
labels = truth.reshape((pixels,1))
print(labels)
```

```
[[0]
 [0]
 [0]
 ...
 [0]
 [0]
 [0]]
```

In order to visualize our image, we print the 10th band of it.

In [9]:

```
fig = plt.figure()
plt.imshow(HSI[:,:,10])
plt.title('RGB Visualization of the 10th band of Pavia University HSI')
plt.show()
```



Part 1: Spectral Unmixing

Part A

We start by reshaping our HSI dataset to a more usefull form and then turn any possible negative values to zero.

In [27]:

```
HSI_flat = HSI.reshape((pixels,L))

for i in range(pixels):
    for j in range(L):
```

```
        if HSI_flat[i,j]<=0:
            HSI_flat[i,j]=0

print(HSI_flat)
```

```
[[ 903  763  658 ... 2988 3036 3045]
 [ 817  937  864 ... 3412 3484 3503]
 [ 811  857  905 ... 2965 2977 2983]
 ...
 [ 734  566  352 ... 3562 3512 3574]
 [ 614  640  582 ... 3338 3363 3430]
 [ 547  583  462 ... 3016 3020 3078]]
```

We define next the error function which will calculate for us the recunstruction error using the type suggested by the exercise. We will later use this function multiple times for our cases.

In [14]:

```
def error(y, y_f):
    errors = np.empty((0))

    for i in range(pixels):
        if labels[i]!=0:
            error_val = y[i,:] - y_f[i,:]
            errors = np.append(errors, [np.linalg.norm(error_val)**2])

    response = np.mean(errors)
    return response
```

Next we define the abundance method which will give 9 different abundance maps, one for each material and helps us present our results for each of these endmembers.

In [15]:

```
def abundance(x):
    fig = plt.figure(figsize=(4 * 5, 4  * 5))
    for i in range(9):
        plt.subplot(1, 9,i+1)
        theta_est_i = x[:,i].reshape((M,N))
        plt.imshow(theta_est_i)
        title = ('Endmember :'+ str(i+1))
        plt.title(title, size=12)
        plt.xticks(())
        plt.yticks(())
    plt.close()
    return fig
```

We will try to perform the Spectral Unmixing method using various methods bellow starting with:
a) Least squares
Here for this method we will use the classic type we have used multiple time for the homeworks.

In [28]:

```
X = endmembers
thetaA = np.zeros((pixels,9))

for i in range(pixels):
    if labels[i]!=0:
        y = HSI_flat[i,:]
        XTX_inv = np.linalg.inv(np.dot(X.T,X))
        thetaA[i] = (XTX_inv).dot(X.T).dot(y.T)
```

We print the theta values we found to see if they seem logical.

In [29]:

```
print(thetaA)
```

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

These zeros make you wonder wether we accomplished what we initially wanted.
We try again by printing a random line.

In [30]:

```
print(thetaA[99])
```

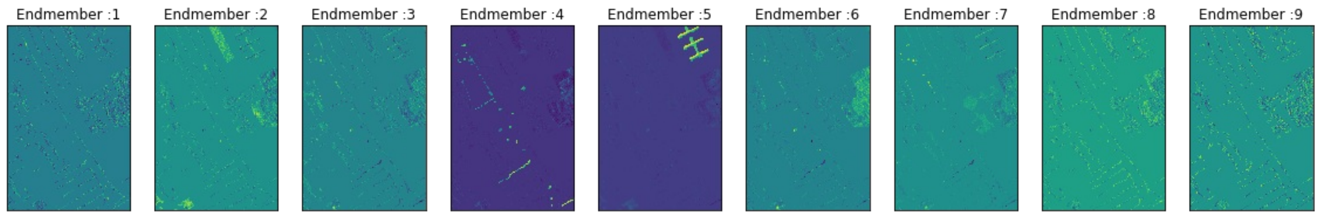
```
[-2.46612292  1.18696598 -0.63611708 -0.08153619 -0.01761107 -0.15800184
  0.27782567  2.13243896  1.55054253]
```

This time the theta values are not all zeros again and this means that maybe we did a good job.
We move on by printing the 9 abundance maps utilizing the method we have created.

In [31]:

```
abundance(thetaA)
```

Out[31]:



Then we compute and print the reconstruction error based on the method we created above.

In [32]:

```
y = HSI_flat
y_estA = X.dot(thetaA.T).transpose()

print('The mean reconstruction error is :', round(error(y, y_estA),3))
```

```
The mean reconstruction error is : 118783.181
```

b) Least squares imposing the sum-to-one constraint
For that exercise we can use scipy.optimize.minimize() with method='SLSQP' which uses fmin_slsqp. At first we need to create a function which we will minimize. We also need an 'eq' constraint function for the minimize method to utilize.
We start by creating the minimization function.

In [21]:

```
def fn(x, A, b):
    return np.linalg.norm(A.dot(x) - b)
```

Then we set up our constraint for the minimization function.

In [22]:

```
constraints = ({'type': 'eq', 'fun': lambda x:  np.sum(x)-1})
```

Finally we have to do again what we did in the last question in order to find our theta estimates. But before doing that, since as non optional input for our minimize method is an initial guess we use the nnls method suggested by the hints of the exercise to feeds its optional input for our minimize method is an initial guess we use the nnls method suggested by the hints of the exercise to feeds its solution vector to the minimize method.

In [33]:

```
thetaB = np.zeros((pixels,9))
X = endmembers
y = HSI_flat[i,:]
x_nnls, rnorm = nnls(X, y)

for i in range(pixels):
    if labels[i]!=0:
        y = HSI_flat[i,:]
        thetaB[i] = minimize(fn, x0=x_nnls, args = (X,y), method = 'SLSQP', constraints = constraint
s).x
```

Next we print our theta values to see what we have found. We also print a random theta value to see that it differs from the others at some point.

In [34]:

```
print(thetaB)
print(thetaB[99])
```

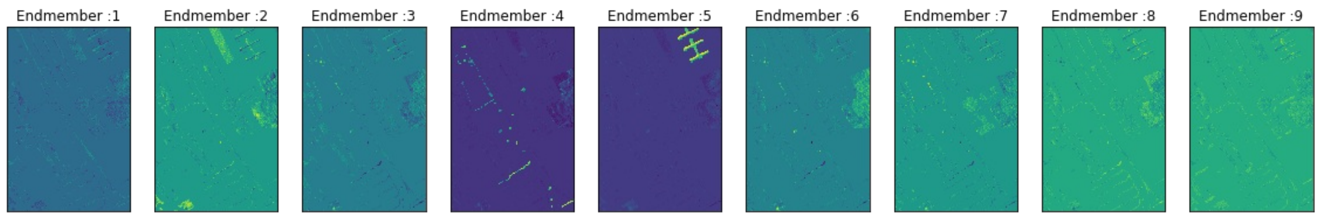
```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
[ 2.66012594e+00  7.57046840e-01  9.86837629e-01 -1.66018153e-02
  7.18552925e-04  4.44028758e-02 -9.72693783e-01 -1.86054067e+00
 -5.99295568e-01]
```

After having found our theta values ,we move on by printing the 9 abundance maps utilizing the method we have created above.

In [35]:

```
abundance(thetaB)
```

Out[35]:



Then again we compute and print the reconstruction error based on the method we created above.

In [36]:

```
y = HSI_flat
y_estB = X.dot(thetaB.T).transpose()

print('The mean reconstruction error is :', round(error(y, y_estB),3))
```

```
The mean reconstruction error is : 160049.931
```

c) Least squares imposing the non-negativity constraint on the entries of θ.
For this exercise we will follow the same logic used in the past two exercises but this time we will utilize the nnls method suggested by the hints of the exercise.

In [37]:

```
thetaC = np.zeros((pixels,9))

for i in range(pixels):
    if labels[i]!=0:
        y = HSI_flat[i,:]
        x_nnls, rnorm = nnls(X, y)
        thetaC[i] = x_nnls
```

After that like before we print our theta estimates to see what we have found

In [38]:

```
print(thetaC)
print(thetaC[99])
```

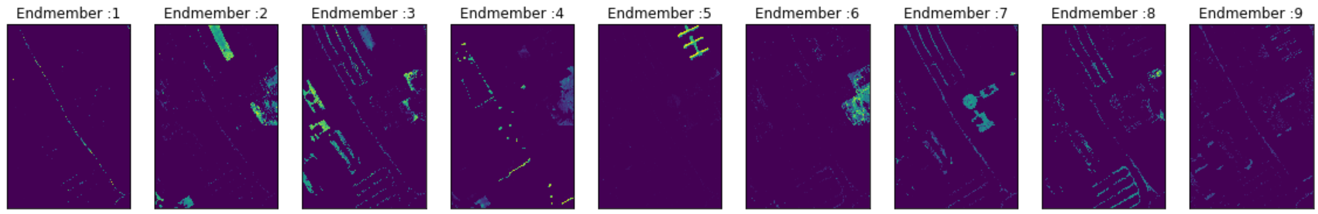
```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
[0.         0.86639606 0.11691542 0.         0.         0.
 0.         0.         0.26364118]
```

The values seem logical and so we move on to the next objective which is to present the 9 abundance maps.

In [39]:

```
abundance(thetaC)
```

Out[39]:



Finally utilizing the error method we have create we try again to calculate the mean reconstruction error.

In [40]:

```
y = HSI_flat
y_estC = X.dot(thetaC.T).transpose()

print('The mean reconstruction error is :', round(error(y, y_estC),3))
```

```
The mean reconstruction error is : 569339.291
```

d) Least squares imposing both the non-negativity and the sum-to-one constraint on the entries of θ.
For the purpose of this exercise we have to work like the exercise 1b but this time simply change it a bit by adding one more constraint.
We start again by setting up our minimization function.

In [41]:

```
def fn(x, A, b):
    return np.linalg.norm(A.dot(x) - b)
```

Then we create our constraint

In [42]:

```
cons = ({'type': 'eq', 'fun': lambda x:  np.sum(x)-1})
```

After some research in https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html . We found that we can use the bounds optional parameter of the minimize method to set bounds on variables for SLSQP method we will use.

In [43]:

```
bounds = [(0., sys.float_info.max) for x in x_nnls]
```

In [44]:

```
thetaD = np.zeros((pixels,9))

y = HSI_flat[i,:]
x_nnls, rnorm = nnls(X, y)

for i in range(pixels):
    if labels[i]!=0:
        y = HSI_flat[i,:]
        thetaD[i] = minimize(fn, x0=x_nnls, args = (X,y), method = 'SLSQP', constraints = cons, boun
ds = bounds).x
```

Next we print our theta estimates to see if our results make sense.

In [45]:

```
print(thetaD)
print(thetaD[99])
```

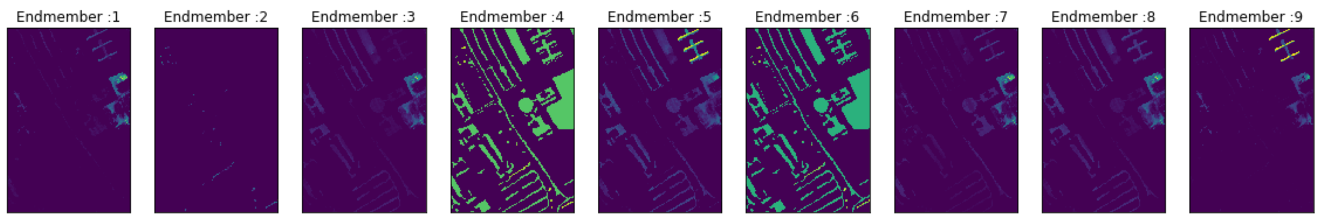
```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
[6.63638927e-06 0.00000000e+00 7.36058713e-03 8.49649420e-01
 6.80851565e-02 1.14528177e-01 3.11714409e-03 9.85744270e-03
 1.81521985e-05]
```

Then we print our abundance maps based on the theta estimates we found.

In [46]:

```
abundance(thetaD)
```

Out[46]:



Finally utilizing the error method we have create we try again to calculate the mean reconstruction error.

In [48]:

```
y = HSI_flat
y_estD = X.dot(thetaD.T).transpose()

print('The mean reconstruction error is :', round(error(y, y_estD),3))
```

```
The mean reconstruction error is : 99392525.493
```

e) LASSO

For this exercise we will use the same method as before but this time we will utilize the Lasso method provided by the sklearn library.

In [51]:

```
X = endmembers
thetaE = np.zeros((pixels,9))

for i in range(pixels):
    if labels[i]>=0:
        y = HSI_flat[i,:]
        clf = Lasso(alpha = 0.01, positive =True, fit_intercept=False, max_iter = 1e7)
        clf.fit(X,y)
        thetaE[i] = clf.coef_
```

At this point we again print our theta estimates to have a look at our work.

In [53]:

```
print(thetaE)
print(thetaE[99])
```

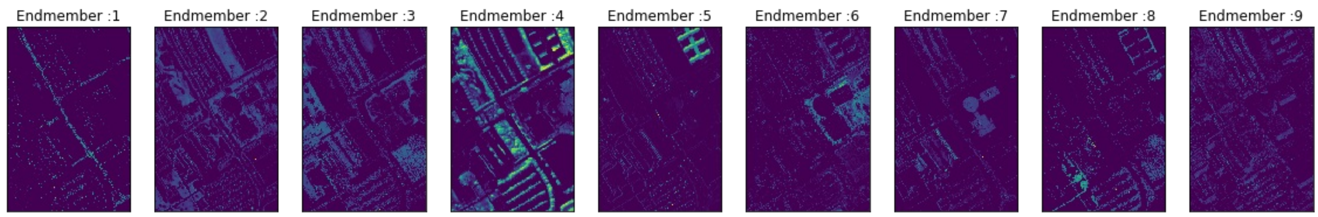
```
[[0.         0.37148733 0.02617406 ... 0.         0.         0.03812831]
 [0.         0.09703562 0.22764319 ... 0.         0.         0.05958018]
 [0.         0.         0.17973425 ... 0.         0.         0.05077734]
 ...
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]]
[0.         0.86690574 0.11660544 0.         0.         0.
 0.         0.         0.26263571]
```

Next, we print our abundance maps based on the theta estimates we found.

In [54]:

```
abundance(thetaE)
```

Out[54]:



Finally utilizing the error method we have create we try again to calculate the mean reconstruction error.

In [52]:

```
y = HSI_flat
y_estE = X.dot(thetaE.T).transpose()

print('The mean reconstruction error is :', round(error(y, y_estE),3))
```

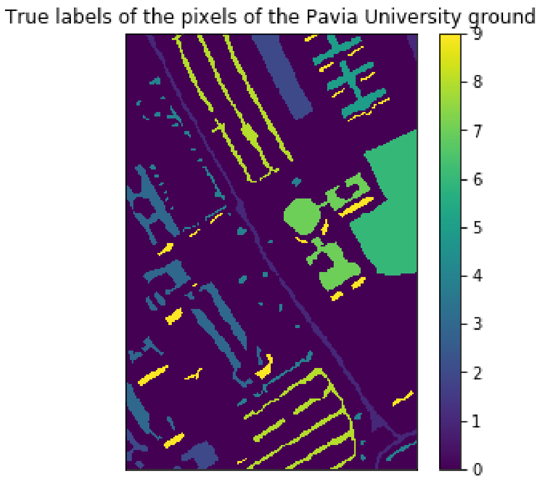
```
The mean reconstruction error is : 570530.294
```

Part B Comparing the results obtained from the above five methods and short comments

Above we have seen the abundance maps for each method tried. Now to get a clear view of the true class labels we print the

In [55]:

```
fig = plt.figure(figsize = (5,5))
plt.imshow(ground_truth['y'])
plt.title('True labels of the pixels of the Pavia University ground')
plt.colorbar()
plt.xticks(())
plt.yticks(())
plt.show()
```



Summarizing and commenting on our results:

Starting with the abundance maps we have ploted and also using the true labels of the pixels for the University we can comment on the following:
1) The best spectral unmixing would give us abundance maps with each only presenting only one of the respective class and contributing 100% to that class. 2) Comparing the class labels for each pixel of the Pavia ground truth with the abundance maps from each of the methods used, we can see that the overall performance of the LS method with non-negativity constraint seems to be the optimal choice.

Moving on with the reconstruction error:
We can see that comparing the reconstruction error for each of the methods we used, the smallest one came from LS without adding any constraints.

This discrepancy between the two methods we have used to find the optimal method for our problem shows us that the reconstruction error surely is not that reliable to be used to measure the performance of a method.

Part 2: Classification

We start the classification part of the exercise by loading the set which we will be using for classification. We do that by utilizing the code given by the exercise.

In [56]:

```
Pavia_labels = sio.loadmat('classification_labels_Pavia.mat')
Training_Set = (np.reshape(Pavia_labels['training_set'],(200,300))).T
Test_Set = (np.reshape(Pavia_labels['test_set'],(200,300))).T
Operational_Set = (np.reshape(Pavia_labels['operational_set'],(200,300))).T
```

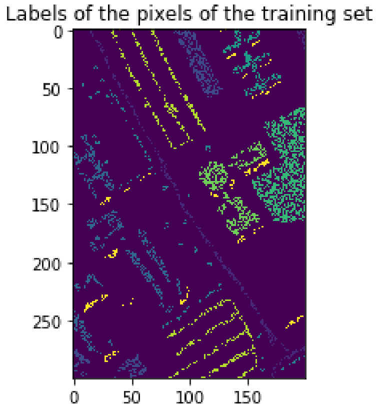
Now we can print the three sets we have created in order to have a look at the data the contain.
We start with the training set.

In [57]:

```
print(Training_Set)
fig = plt.figure()
plt.imshow(Training_Set)
plt.title('Labels of the pixels of the training set')
plt.show()
```

```
plt.show()
```

```
[[0 0 0 ... 1 1 1]
 [0 0 0 ... 1 0 0]
 [0 0 0 ... 0 1 1]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

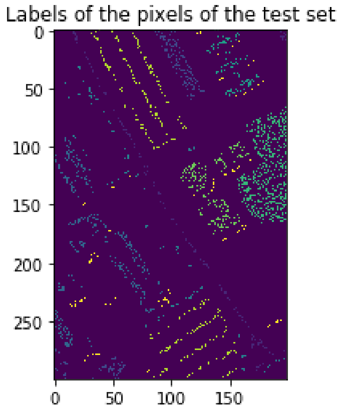


Moving to the test set.

In [58]:

```
print(Test_Set)
fig = plt.figure()
plt.imshow(Test_Set)
plt.title('Labels of the pixels of the test set')
plt.show()
```

```
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 1]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

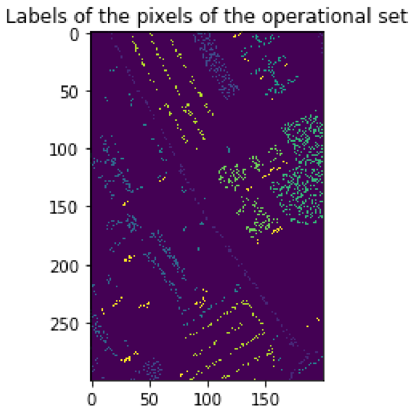


Finally we print our operational set.

In [59]:

```
print(Operational_Set)
fig = plt.figure()
plt.imshow(Operational_Set)
plt.title('Labels of the pixels of the operational set')
plt.show()
```

```
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 1 0]
 [0 0 0 ... 1 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```



To start training our models we have to first split our training, test and operational sets to X and Y in order to feed them to our model. So we start with that. As it is only logical, we check only for pixels with are not labeled as zero.

First we create reshaped editions of our data which we will use, as they gives us easier access to the data.

In [60]:

```
Training_Set_reshaped = Training_Set.reshape((pixels,1))
print('Training set reshaped:')
print(Training_Set_reshaped[10:20])
print('--------------------------')
Test_Set_reshaped = Test_Set.reshape((pixels,1))
print('Test set reshaped:')
print(Test_Set_reshaped[10:20])
print('--------------------------')
Operational_Set_reshaped = Operational_Set.reshape((pixels,1))
print('Operational set reshaped:')
print(Operational_Set_reshaped[10:20])
```

```
Training set reshaped:
[[0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [1]
 [1]
 [0]]
--------------------------
Test set reshaped:
[[0]
 [0]
 [0]
 [0]
 [1]
 [0]
 [1]
 [0]
 [0]
 [0]]
--------------------------
Operational set reshaped:
[[0]
 [0]
 [0]
 [0]
 [0]
```

```
 [0]
 [1]
 [0]
 [0]
 [0]
 [1]]
```

Next we move on with what we started and split our data to X and y variables. We start with the training set.

In [61]:

```
X_train = np.empty((0,103))
y_train =  np.empty((0,1))

for i in range(pixels):
    if Training_Set_reshaped[i]!=0:
        X_train = np.append(X_train, [HSI_flat[i,:]], axis = 0)
        y_train = np.append(y_train, [Training_Set_reshaped[i]], axis = 0)

print('Train set X:')
print(X_train[0:5])
print('-------------------------------------------------------------------')
print('Train set y:')
print(y_train[0:5])
```

```
Train set X:
[[1512. 1384. 1113. 1151. 1266. 1252. 1263. 1284. 1275. 1273. 1284. 1265.
  1222. 1219. 1208. 1185. 1215. 1236. 1234. 1272. 1295. 1273. 1296. 1305.
  1304. 1339. 1362. 1361. 1356. 1381. 1398. 1398. 1388. 1396. 1422. 1418.
  1428. 1463. 1466. 1467. 1497. 1506. 1502. 1502. 1508. 1509. 1495. 1482.
  1483. 1488. 1481. 1498. 1526. 1542. 1538. 1544. 1553. 1550. 1538. 1524.
  1522. 1535. 1544. 1530. 1538. 1568. 1572. 1554. 1560. 1562. 1565. 1547.
  1539. 1561. 1599. 1626. 1615. 1588. 1565. 1556. 1552. 1532. 1511. 1592.
  1677. 1664. 1636. 1620. 1610. 1600. 1595. 1587. 1580. 1573. 1574. 1557.
  1539. 1548. 1550. 1572. 1560. 1563. 1580.]
 [1502. 1425. 1305. 1278. 1313. 1279. 1285. 1280. 1292. 1319. 1300. 1284.
  1298. 1313. 1309. 1283. 1264. 1264. 1286. 1303. 1330. 1336. 1338. 1378.
  1389. 1368. 1369. 1388. 1406. 1408. 1422. 1450. 1457. 1439. 1423. 1433.
  1463. 1479. 1471. 1461. 1460. 1465. 1476. 1466. 1450. 1453. 1462. 1456.
  1440. 1447. 1507. 1546. 1525. 1482. 1454. 1457. 1455. 1484. 1502. 1485.
  1475. 1475. 1489. 1482. 1475. 1483. 1482. 1479. 1495. 1499. 1479. 1460.
  1470. 1488. 1499. 1538. 1546. 1521. 1511. 1528. 1525. 1461. 1372. 1444.
  1572. 1552. 1510. 1495. 1491. 1482. 1499. 1506. 1512. 1543. 1524. 1473.
  1468. 1496. 1487. 1463. 1459. 1491. 1496.]
 [ 958.  989. 1020. 1117. 1243. 1425. 1500. 1393. 1379. 1414. 1408. 1375.
  1402. 1463. 1515. 1496. 1470. 1441. 1434. 1465. 1494. 1495. 1507. 1502.
  1501. 1503. 1486. 1503. 1530. 1571. 1577. 1565. 1583. 1615. 1622. 1636.
  1660. 1681. 1662. 1640. 1653. 1655. 1651. 1670. 1700. 1695. 1693. 1699.
  1677. 1657. 1679. 1699. 1696. 1725. 1736. 1706. 1691. 1713. 1745. 1742.
  1731. 1723. 1711. 1697. 1705. 1715. 1716. 1731. 1760. 1762. 1732. 1706.
  1724. 1762. 1789. 1792. 1774. 1801. 1821. 1818. 1811. 1774. 1727. 1773.
  1852. 1844. 1809. 1772. 1756. 1777. 1796. 1804. 1794. 1783. 1808. 1846.
  1816. 1800. 1762. 1739. 1797. 1862. 1869.]
 [1495. 1348. 1317. 1432. 1501. 1526. 1509. 1445. 1456. 1615. 1615. 1492.
  1504. 1547. 1552. 1539. 1537. 1548. 1570. 1579. 1604. 1640. 1659. 1650.
  1682. 1731. 1750. 1733. 1716. 1743. 1752. 1754. 1767. 1806. 1821. 1814.
  1825. 1833. 1843. 1872. 1878. 1852. 1850. 1860. 1879. 1886. 1884. 1874.
  1872. 1873. 1894. 1917. 1924. 1901. 1882. 1902. 1924. 1955. 1959. 1930.
  1907. 1912. 1925. 1931. 1923. 1944. 1960. 1948. 1953. 1978. 1988. 1975.
  1958. 1947. 1942. 1952. 1969. 1986. 1986. 1968. 1963. 1956. 1902. 1911.
  1982. 1992. 2004. 1999. 1969. 1933. 1905. 1900. 1917. 1948. 1939. 1898.
  1876. 1874. 1866. 1870. 1903. 1928. 1891.]
 [1378. 1253. 1241. 1530. 1598. 1527. 1597. 1623. 1518. 1502. 1627. 1657.
  1591. 1565. 1601. 1656. 1689. 1693. 1674. 1682. 1706. 1740. 1782. 1812.
  1828. 1847. 1844. 1839. 1824. 1818. 1817. 1822. 1830. 1868. 1882. 1855.
  1863. 1878. 1869. 1850. 1854. 1857. 1847. 1840. 1834. 1818. 1806. 1787.
  1768. 1758. 1748. 1741. 1725. 1726. 1692. 1689. 1698. 1684. 1660. 1632.
  1614. 1624. 1612. 1563. 1546. 1548. 1547. 1529. 1532. 1522. 1487. 1454.
  1449. 1466. 1454. 1440. 1430. 1419. 1411. 1390. 1379. 1342. 1324. 1373.
  1395. 1356. 1330. 1309. 1311. 1308. 1338. 1355. 1332. 1311. 1287. 1262.
  1264. 1292. 1300. 1307. 1324. 1299. 1237.]]
-------------------------------------------------------------------
Train set y:
[[1.]
 [1.]
```

 [8.]
 [8.]
 [8.]]

We move on to the test set.

In [62]:

```
X_test = np.empty((0,103))
y_test =  np.empty((0,1))

for i in range(pixels):
    if Test_Set_reshaped[i]!=0:
        X_test= np.append(X_test, [HSI_flat[i,:]], axis = 0)
        y_test = np.append(y_test, [Test_Set_reshaped[i]], axis = 0)

print('Test set X:')
print(X_test[0:5])
print('---------------------------------------------------------------')
print('Test set y:')
print(y_test[0:5])
```

```
Test set X:
[[1351. 1214. 1293. 1482. 1389. 1330. 1322. 1287. 1334. 1365. 1324. 1230.
  1179. 1175. 1204. 1222. 1258. 1291. 1296. 1304. 1313. 1333. 1334. 1307.
  1315. 1325. 1311. 1304. 1316. 1322. 1331. 1351. 1361. 1361. 1363. 1377.
  1392. 1389. 1375. 1378. 1415. 1417. 1404. 1394. 1377. 1385. 1398. 1392.
  1387. 1384. 1383. 1365. 1358. 1391. 1410. 1401. 1393. 1406. 1411. 1409.
  1398. 1397. 1410. 1412. 1393. 1376. 1381. 1393. 1414. 1411. 1411. 1402.
  1412. 1448. 1464. 1467. 1457. 1466. 1487. 1469. 1466. 1418. 1371. 1452.
  1533. 1509. 1482. 1480. 1494. 1480. 1455. 1461. 1434. 1429. 1431. 1410.
  1407. 1418. 1424. 1442. 1477. 1512. 1499.]
 [1462. 1231. 1272. 1356. 1274. 1250. 1312. 1212. 1112. 1104. 1180. 1220.
  1223. 1220. 1243. 1245. 1241. 1268. 1300. 1278. 1211. 1177. 1205. 1213.
  1200. 1245. 1272. 1250. 1229. 1259. 1316. 1313. 1293. 1311. 1343. 1367.
  1359. 1346. 1335. 1339. 1367. 1389. 1393. 1382. 1375. 1398. 1414. 1426.
  1437. 1414. 1416. 1441. 1442. 1427. 1412. 1413. 1435. 1454. 1455. 1458.
  1465. 1481. 1480. 1470. 1451. 1469. 1481. 1496. 1529. 1527. 1486. 1446.
  1435. 1441. 1502. 1562. 1550. 1535. 1546. 1558. 1548. 1502. 1465. 1536.
  1625. 1599. 1540. 1511. 1530. 1567. 1602. 1595. 1556. 1531. 1484. 1469.
  1514. 1573. 1586. 1563. 1556. 1572. 1587.]
 [1125. 1389. 1507. 1554. 1460. 1453. 1534. 1608. 1563. 1460. 1432. 1433.
  1500. 1553. 1595. 1613. 1649. 1672. 1671. 1691. 1696. 1671. 1688. 1717.
  1730. 1772. 1799. 1797. 1789. 1797. 1830. 1837. 1852. 1850. 1856. 1887.
  1917. 1923. 1909. 1905. 1901. 1911. 1927. 1920. 1939. 1970. 1964. 1946.
  1929. 1913. 1914. 1919. 1945. 1957. 1958. 1953. 1966. 1979. 1969. 1965.
  1963. 1957. 1964. 1950. 1940. 1959. 1983. 1995. 2019. 2039. 2015. 1965.
  1951. 1974. 2000. 2023. 2013. 2018. 2026. 2027. 2012. 1951. 1898. 1936.
  2009. 2000. 1973. 1958. 1957. 1954. 1948. 1937. 1897. 1866. 1856. 1867.
  1848. 1878. 1872. 1786. 1725. 1758. 1802.]
 [1263. 1362. 1380. 1413. 1530. 1525. 1449. 1414. 1460. 1509. 1518. 1516.
  1555. 1553. 1569. 1603. 1601. 1606. 1644. 1688. 1688. 1719. 1777. 1807.
  1843. 1897. 1904. 1887. 1876. 1909. 1951. 1957. 1978. 2042. 2072. 2087.
  2116. 2135. 2109. 2104. 2141. 2134. 2138. 2156. 2167. 2189. 2189. 2197.
  2201. 2205. 2221. 2203. 2177. 2183. 2199. 2198. 2201. 2219. 2229. 2224.
  2202. 2184. 2199. 2220. 2226. 2246. 2266. 2243. 2232. 2233. 2196. 2181.
  2175. 2163. 2178. 2233. 2255. 2239. 2222. 2202. 2194. 2181. 2103. 2086.
  2190. 2211. 2176. 2154. 2158. 2145. 2156. 2160. 2122. 2073. 2047. 2039.
  2009. 1993. 2004. 1997. 2018. 2063. 2033.]
 [ 882.  639.  609.  760.  745.  733.  740.  688.  687.  681.  758.  816.
   830.  812.  810.  826.  822.  806.  808.  841.  884.  900.  920.  946.
   984.  998. 1005. 1015. 1025. 1036. 1050. 1067. 1045. 1031. 1063. 1080.
  1090. 1128. 1148. 1126. 1128. 1167. 1174. 1203. 1254. 1269. 1250. 1250.
  1255. 1245. 1263. 1284. 1293. 1285. 1274. 1286. 1306. 1337. 1348. 1345.
  1339. 1334. 1333. 1360. 1411. 1433. 1450. 1483. 1566. 1638. 1676. 1695.
  1752. 1813. 1839. 1913. 1994. 2039. 2051. 2081. 2127. 2158. 2122. 2108.
  2167. 2205. 2198. 2193. 2184. 2159. 2184. 2224. 2239. 2240. 2239. 2199.
  2163. 2166. 2213. 2263. 2272. 2268. 2245.]]
---------------------------------------------------------------
Test set y:
[[1.]
 [1.]
 [8.]
 [8.]
 [2.]]
```

Finally we do the same work for the operation set.

In [63]:

```
Operational_Set_reshaped = Operational_Set.reshape((pixels,1))
X_operation = np.empty((0,103))
y_operation =  np.empty((0,1))

for i in range(pixels):
    if Operational_Set_reshaped[i]!=0:
        X_operation= np.append(X_operation, [HSI_flat[i,:]], axis = 0)
        y_operation = np.append(y_operation, [Operational_Set_reshaped[i]], axis = 0)

print('Operation set X:')
print(X_operation[0:5])
print('------------------------------------------------------------')
print('Operation set y:')
print(y_operation[0:5])
```

```
Operation set X:
[[1377. 1379. 1425. 1467. 1427. 1302. 1258. 1245. 1268. 1276. 1214. 1199.
  1270. 1265. 1224. 1231. 1239. 1193. 1167. 1193. 1206. 1247. 1266. 1248.
  1258. 1244. 1234. 1264. 1296. 1282. 1264. 1260. 1283. 1319. 1321. 1331.
  1334. 1323. 1327. 1339. 1338. 1328. 1325. 1328. 1337. 1362. 1363. 1336.
  1334. 1340. 1361. 1381. 1370. 1325. 1324. 1353. 1375. 1398. 1407. 1385.
  1389. 1407. 1428. 1415. 1397. 1408. 1401. 1396. 1423. 1440. 1434. 1375.
  1362. 1398. 1434. 1468. 1476. 1481. 1488. 1497. 1493. 1465. 1452. 1493.
  1560. 1536. 1515. 1506. 1485. 1460. 1456. 1462. 1489. 1509. 1524. 1470.
  1448. 1464. 1469. 1472. 1461. 1466. 1439.]
 [1541. 1390. 1333. 1371. 1326. 1263. 1225. 1233. 1221. 1197. 1192. 1235.
  1252. 1245. 1256. 1298. 1319. 1293. 1259. 1253. 1276. 1287. 1292. 1304.
  1304. 1323. 1329. 1336. 1345. 1349. 1342. 1324. 1324. 1346. 1362. 1369.
  1398. 1403. 1382. 1372. 1390. 1415. 1404. 1388. 1393. 1391. 1387. 1371.
  1374. 1384. 1396. 1415. 1416. 1407. 1408. 1428. 1433. 1445. 1458. 1438.
  1426. 1426. 1407. 1397. 1408. 1437. 1442. 1441. 1449. 1455. 1452. 1416.
  1396. 1420. 1443. 1455. 1442. 1457. 1465. 1449. 1443. 1444. 1400. 1420.
  1483. 1462. 1443. 1425. 1421. 1443. 1444. 1427. 1428. 1415. 1434. 1444.
  1431. 1455. 1465. 1442. 1453. 1450. 1431.]
 [1728. 1531. 1431. 1442. 1542. 1592. 1586. 1532. 1395. 1370. 1419. 1440.
  1420. 1437. 1483. 1503. 1532. 1541. 1543. 1544. 1548. 1590. 1606. 1596.
  1570. 1612. 1659. 1679. 1699. 1702. 1691. 1681. 1692. 1723. 1766. 1786.
  1783. 1766. 1744. 1727. 1718. 1727. 1730. 1734. 1723. 1726. 1741. 1769.
  1778. 1744. 1734. 1762. 1784. 1771. 1766. 1774. 1756. 1759. 1769. 1756.
  1760. 1779. 1768. 1725. 1713. 1726. 1738. 1723. 1734. 1748. 1738. 1688.
  1651. 1692. 1723. 1733. 1721. 1715. 1702. 1711. 1705. 1661. 1594. 1661.
  1748. 1735. 1707. 1696. 1689. 1657. 1639. 1630. 1623. 1632. 1611. 1572.
  1577. 1622. 1645. 1614. 1599. 1630. 1638.]
 [ 782.  580.  786.  927.  888.  737.  724.  766.  794.  814.  810.  790.
   770.  744.  767.  771.  780.  776.  759.  771.  801.  832.  867.  901.
   906.  914.  908.  921.  963.  994. 1005. 1012. 1046. 1074. 1077. 1084.
  1113. 1156. 1167. 1146. 1146. 1153. 1160. 1189. 1217. 1220. 1210. 1218.
  1230. 1233. 1244. 1252. 1273. 1292. 1286. 1286. 1292. 1295. 1308. 1323.
  1325. 1337. 1358. 1373. 1374. 1404. 1454. 1499. 1560. 1621. 1684. 1741.
  1778. 1825. 1905. 1980. 2035. 2101. 2140. 2149. 2181. 2196. 2161. 2143.
  2205. 2237. 2240. 2232. 2219. 2220. 2259. 2287. 2285. 2294. 2305. 2265.
  2226. 2254. 2305. 2336. 2301. 2287. 2286.]
 [ 814.  834.  884.  905.  898.  865.  826.  809.  766.  730.  744.  721.
   707.  706.  738.  817.  852.  841.  854.  877.  873.  887.  911.  936.
   992. 1020. 1016. 1016. 1017. 1053. 1098. 1091. 1103. 1128. 1147. 1178.
  1189. 1194. 1195. 1205. 1225. 1241. 1239. 1264. 1299. 1302. 1308. 1336.
  1369. 1363. 1353. 1393. 1439. 1457. 1451. 1439. 1458. 1493. 1488. 1499.
  1531. 1561. 1578. 1593. 1616. 1623. 1652. 1685. 1719. 1753. 1802. 1848.
  1849. 1860. 1904. 1990. 2054. 2087. 2093. 2108. 2120. 2113. 2089. 2104.
  2191. 2197. 2194. 2182. 2179. 2200. 2223. 2213. 2207. 2224. 2228. 2215.
  2216. 2236. 2252. 2245. 2189. 2183. 2207.]]
------------------------------------------------------------
Operation set y:
[[1.]
 [1.]
 [8.]
 [2.]
 [2.]]
```

Part A

Having completed all preparations for the next steps, we start with the first model needed by the exercise.
For each model used below we will do what the exercise suggests.

1. We train our model on the training set and perform a 10-fold cross validation. Next each time we will report the estimated validation error as the mean of the ten resulting error values we will compute also the standard deviation as the exercise suggests.
2. After train we will use the whole training set to train the classifier and evaluate its performance on the test set as follows: First, we will compute the confusion matrix (a 9x9matrix, whose (i,j) element is the number of pixels that belong to the i-class and are assigned from the classifier to the j-th class. Clearly, the “more diagonal” the matrix, the better the performance of the classifier is) and we will identify the classes that are not well separated by the classifier. Then, we will compute the success rate of the classifier as the sum of the diagonal elements of the confusion matrix divided by the sum of all elements of the matrix.

i) Naive Bayes Classifier

1)

In [64]:

```
NB_score = cross_val_score(GaussianNB(), X_train, y_train.reshape(-1), cv = 10)
NB_error = 1 - NB_score
NB_mean = round(NB_error.mean(),3)
NB_std = round(NB_error.std(),3)
print('Naive Bayes validation error is:', NB_mean)
print('Naive Bayes error standard deviation error is:', NB_std)
```

```
Naive Bayes validation error is: 0.355
Naive Bayes error standard deviation error is: 0.057
```

2)

In [65]:

```
NB_model = GaussianNB()
NB_model.fit(X_train, y_train.reshape(-1))
NB_predictions = NB_model.predict(X_test)
print('Some Naive Bayes predictions ',NB_predictions[0:5])
```

```
Some Naive Bayes predictions  [7. 7. 8. 8. 2.]
```

Then we create and print the confusion matrix.

In [66]:

```
NB_cm = confusion_matrix(y_test, NB_predictions)
print('Naive Bayes Confusion Matrix:')
print(NB_cm)
```

```
Naive Bayes Confusion Matrix:
[[131    0   37    0    0    0   80   13    0]
 [   0 326    4    6    0   17    0    0    0]
 [  25    2 127    0    0   13   70  299    0]
 [   0    0    0 154    1    1    0    0    0]
 [   0    0    1    0  166    1    0    0    0]
 [   0 312    2   55   32  363    0    0    0]
 [  18    0   26    0    0    0  277    0    0]
 [   2    1   67    0    0    1    2  388    0]
 [   0    0    0    2    0    0    0    0  185]]
```

Next we calculate the succes rate as adviced by the exercise

In [67]:

```
#numpy trace function returns the sum of the diagonal elements
#numpy sum function returns the sum of all the elements
NB_sc = np.trace(NB_cm)/ np.sum(NB_cm)
print('Naive Bayes classifier success rate is:',NB_sc)
```

```
Naive Bayes classifier success rate is: 0.660118490801372
```

ii) Minimum Euclidean distance classifier

1) As we cannot use an already made function for this classifier below we present a simple way to create a Minimum Euclidean distance classifier.

In [69]:

```
def get_class_means(X,y):
    X_per_class = np.empty((0,103))
    x_means = np.empty((0,103))

    for label in range(1,10):
        for i in range(X.shape[0]):
            if y[i]== label:
                X_per_class = np.append(X_per_class, [X[i]], axis = 0)
        mean_per_class = np.mean(X_per_class, axis = 0)
        x_means = np.append(x_means, [mean_per_class], axis = 0)
        X_per_class = np.empty((0,103))
    return x_means
```

In [70]:

```
def classify_dist(min_dist, dist):
    if min_dist == dist[0]:
        return 1
    elif min_dist == dist[1]:
        return 2
    elif min_dist == dist[2]:
        return 3
    elif min_dist == dist[3]:
        return 4
    elif min_dist == dist[4]:
        return 5
    elif min_dist == dist[5]:
        return 6
    elif min_dist == dist[6]:
        return 7
    elif min_dist == dist[7]:
        return 8
    else:
        return 9
```

In [71]:

```
kf = KFold(n_splits=10)
errors_cv = np.empty((0,1))

#now for the cross validation part

for i, j in kf.split(X_train, y_train):
    X_train_cv = X_train[i]
    y_train_cv = y_train[i]
    X_test_cv = X_train[j]
    y_test_cv = y_train[j]
    C = X_test_cv.shape[0]

    class_means = get_class_means(X_train_cv, y_train_cv)

    MED_pred = np.empty((0,1))

    dist = np.empty((C,9))

    for i in range(C):
        for j in range(9):
            dist[i,j] = np.linalg.norm(X_test_cv[i] - class_means[j])**2
```

```
            min_dist = np.min(dist[i,:])

        MED_pred = np.append(MED_pred,classify_dist(min_dist, dist[i]))

    count_errors = 0

    for k in range(C):
        diff = y_test_cv[k] - MED_pred[k]
        if diff == 0:
            continue
        else:
            count_errors +=1

    prob_error = count_errors / C
    errors_cv = np.append(errors_cv, prob_error)


MED_mean = round(np.mean(errors_cv),3)
MED_std = round(np.std(errors_cv),3)

print('The Minimum Euclidean distance classifier validation error is', MED_mean)
print('The Minimum Euclidean distance classifier standard deviation validation error is', MED_std)
```

```
The Minimum Euclidean distance classifier validation error is 0.465
The Minimum Euclidean distance classifier standard deviation validation error is 0.105
```

2)

Next like before we make some predictions and print the confusion matrix.

In [72]:

```
MED_class_means = get_class_means(X_train, y_train)
D =  X_test.shape[0]
med_pred2 = np.empty((0,1))
dist = np.empty((D,9))

for i in range(D):
    for j in range(9):
        dist[i,j] = np.linalg.norm(X_test[i] - MED_class_means[j])**2
        min_dist = np.min(dist[i,:])

    med_pred2 = np.append(med_pred2,classify_dist(min_dist, dist[i]))

MED_cm = confusion_matrix(y_test, med_pred2)
print('Minimum Euclidean Distance classifier confusion matrix:')
print(MED_cm)
```

```
Minimum Euclidean Distance classifier confusion matrix:
[[152   0  46   0   0   0  61   2   0]
 [  1 188   0   5   0 156   0   3   0]
 [ 66   2 198   0   0   1  39 230   0]
 [  0   0   0 154   0   0   0   0   2]
 [  0   0   0   0 128   0   0  40   0]
 [ 11 317   0  12  16 240   0 168   0]
 [ 61   0  23   0   0   0 237   0   0]
 [  2   1 145   0   0   1   7 305   0]
 [  0   0   0   0   0   0   0   0 187]]
```

Next we calculate the succes rate as adviced by the exercise

In [73]:

```
MED_sc = np.trace(MED_cm)/ np.sum(MED_cm)
print('Minimum euclidean distance classifier success rate is:', MED_sc)
```

```
Minimum euclidean distance classifier success rate is: 0.5578422201434362
```

iii) KNN classifier

1)

Next like before we train our model with cross validation and print the mean error and standard deviation.

In [74]:

```
KNN_score = cross_val_score(KNeighborsClassifier(n_neighbors=9), X_train, y_train.reshape(-1), cv =
10)
KNN_error = 1 - KNN_score
KNN_mean = round(KNN_error.mean(),3)
KNN_std = round(KNN_error.std(),3)
print('KNN validation error mean is:', KNN_mean)
print('KNN error standard deviation error is:', KNN_std)
```

```
KNN validation error mean is: 0.148
KNN error standard deviation error is: 0.05
```

2)

Next we make predictions and calculate and print the confusion matrix

In [75]:

```
KNN_model = KNeighborsClassifier(n_neighbors=9)
KNN_model.fit(X_train, y_train.reshape(-1))
KNN_predictions = KNN_model.predict(X_test)
print('Some KNN predictions ',KNN_predictions[0:5])

KNN_cm = confusion_matrix(y_test, KNN_predictions)
print('KNN Confusion Matrix:')
print(KNN_cm)
```

```
Some KNN predictions  [1. 1. 8. 8. 2.]
KNN Confusion Matrix:
[[188   0  13   0   0   0  28  32   0]
 [  0 323   0   1   0  28   0   1   0]
 [ 11   2 446   0   0   5   1  71   0]
 [  0   0   0 155   0   1   0   0   0]
 [  0   0   1   0 166   0   0   1   0]
 [  0  63   1   0   1 697   0   2   0]
 [ 13   0   5   0   0   0 301   2   0]
 [  9   2  88   0   0   0   2 360   0]
 [  0   0   0   0   0   0   0   0 187]]
```

Next we calculate the success rate like before

In [76]:

```
KNN_sc = np.trace(KNN_cm)/ np.sum(KNN_cm)
print('KNN classifier success rate is:',KNN_sc)
```

```
KNN classifier success rate is: 0.8802619270346118
```

iv) Bayesian classifier

1)

For the Bayes classifier we will use the QuadraticDiscriminantAnalysis which is basically a Bayes classifier developed for us in sklearn.

In [77]:

```
B_score = cross_val_score(QuadraticDiscriminantAnalysis(), X_train, y_train.reshape(-1), cv = 10)
B_error = 1 - B_score
B_mean = round(B_error.mean(),3)
B_std = round(B_error.std(),3)
print('Bayes validation error is:', B_mean)
```

```
print('Bayes error standard deviation error is:', B_std)
```

```
Bayes validation error is: 0.144
Bayes error standard deviation error is: 0.029
```

2)

Next we make predictions and calculate and print the confusion matrix

In [78]:

```
B_model = QuadraticDiscriminantAnalysis()
B_model.fit(X_train, y_train.reshape(-1))
B_predictions = B_model.predict(X_test)
print('Some Bayes classifier predictions ',B_predictions[0:5])

B_cm = confusion_matrix(y_test, B_predictions)
print('Bayes classifier Confusion Matrix:')
print(B_cm)
```

```
Some Bayes classifier predictions  [1. 1. 8. 3. 6.]
Bayes classifier Confusion Matrix:
[[155   0  46   0   0   2  10  48   0]
 [  0 328   0   3   0  22   0   0   0]
 [ 10   1 430   0   0   0   0  95   0]
 [  0   0   0 154   0   2   0   0   0]
 [  0   0   0   0 168   0   0   0   0]
 [  0   1   0   1   0 762   0   0   0]
 [ 14   0  10   0   0   2 291   4   0]
 [ 19   0  73   0   0   2   0 367   0]
 [  3   0   0   1   2   0   0   0 181]]
```

Next we calculate the success rate like before

In [79]:

```
B_sc = np.trace(B_cm)/ np.sum(B_cm)
print('Bayes classifier success rate is:',B_sc)
```

```
Bayes classifier success rate is: 0.8843155597131276
```

Part B

At this point we will compare the results of the classifiers and comment on them. As adviced by the exercise we will relate the confusion matrices obtained from each classifier to each other. We will present them and pay attention to nondiagonal entries that are “not nearly zero”. Since for each method we tried, we used the same dataset for training and then the same for testing and then we also used same folds for cross validation, it is safe to compare the results of the classifiers. We start by comparing the success rates of each method and then we move on to the confusion matrix for each one of them.

In [80]:

```
print('Naive Bayes classifier success rate is:',NB_sc)
print('Minimum euclidean distance classifier success rate is:', MED_sc)
print('KNN classifier success rate is:',KNN_sc)
print('Bayes classifier success rate is:',B_sc)
```

```
Naive Bayes classifier success rate is: 0.660118490801372
Minimum euclidean distance classifier success rate is: 0.5578422201434362
KNN classifier success rate is: 0.8802619270346118
Bayes classifier success rate is: 0.8843155597131276
```

Comparing the success rates for each classifier, we can clearly see that the simple Bayes classifier did the best job, scoring the best success rate of 0.884! The worst job amongst the classifiers used, we performed by the minimum euclidean distance classifier with score of only above average (0.557).

Next we compare the confusion matrices.

In [81]:

```
print('Naive Bayes classifier Confusion Matrix:')
print(NB_cm)
print('-----------------------------------')
print('Minimum Euclidean Distance classifier confusion matrix:')
print(MED_cm)
print('-----------------------------------')
print('KNN classifier Confusion Matrix:')
print(KNN_cm)
print('-----------------------------------')
print('Bayes classifier Confusion Matrix:')
print(B_cm)
```

```
Naive Bayes classifier Confusion Matrix:
[[131   0  37   0   0   0  80  13   0]
 [  0 326   4   6   0  17   0   0   0]
 [ 25   2 127   0   0  13  70 299   0]
 [  0   0   0 154   1   1   0   0   0]
 [  0   0   1   0 166   1   0   0   0]
 [  0 312   2  55  32 363   0   0   0]
 [ 18   0  26   0   0   0 277   0   0]
 [  2   1  67   0   0   1   2 388   0]
 [  0   0   0   2   0   0   0   0 185]]
------------------------------------
Minimum Euclidean Distance classifier confusion matrix:
[[152   0  46   0   0   0  61   2   0]
 [  1 188   0   5   0 156   0   3   0]
 [ 66   2 198   0   0   1  39 230   0]
 [  0   0   0 154   0   0   0   0   2]
 [  0   0   0   0 128   0   0  40   0]
 [ 11 317   0  12  16 240   0 168   0]
 [ 61   0  23   0   0   0 237   0   0]
 [  2   1 145   0   0   1   7 305   0]
 [  0   0   0   0   0   0   0   0 187]]
------------------------------------
KNN classifier Confusion Matrix:
[[188   0  13   0   0   0  28  32   0]
 [  0 323   0   1   0  28   0   1   0]
 [ 11   2 446   0   0   5   1  71   0]
 [  0   0   0 155   0   1   0   0   0]
 [  0   0   1   0 166   0   0   1   0]
 [  0  63   1   0   1 697   0   2   0]
 [ 13   0   5   0   0   0 301   2   0]
 [  9   2  88   0   0   0   2 360   0]
 [  0   0   0   0   0   0   0   0 187]]
------------------------------------
Bayes classifier Confusion Matrix:
[[155   0  46   0   0   2  10  48   0]
 [  0 328   0   3   0  22   0   0   0]
 [ 10   1 430   0   0   0   0  95   0]
 [  0   0   0 154   0   2   0   0   0]
 [  0   0   0   0 168   0   0   0   0]
 [  0   1   0   1   0 762   0   0   0]
 [ 14   0  10   0   0   2 291   4   0]
 [ 19   0  73   0   0   2   0 367   0]
 [  3   0   0   1   2   0   0   0 181]]
```

As suggested by the exercise, for each of the confusion matrices we check the non-diagonal terms and check for non zero numbers. As we can see from the matrices presented above the last one which is the Bayes Classifier has the least off diagonal non zero values. Also the terms which are non zero have the smallest values compared with the other classifieres.

Part 3: Combination of results

For this final part we will comment briefly on the possible correlation of the results obtained from the spectral unmixing procedure with those obtained from classification.
At first we will comment on each part of the project and its results and then on the correlation between the results from each method.
The first part of the project consisted of spectral unmixing. This way we got the abundance maps for each endmember of the image.

The second part ot the exercise was classification. This method assigned each of the pixels of the image to a class.
The two methods are not really that far in practice and in the end the results obtained from the first method should agree with the results obtained by the other. So when we can find an endmbember with the highest contribution percentage for the image from the spectral unmixing, then this should be assigned by the classifier to the same class and so the results should aggree. As we saw from the spectral unmixing part of the exercise, the methods we used gave us not really good results and the best method we the Ls without any contraints.
From the second part, we saw that the simple Bayes classifier did the best job classifying our pixels.
Finally what we can comment is that both the spectral unmixing and the classification for the image given seem to be really difficult problems for each of the methoss used and that the results from both parts of the exercise seem to agree.