Problem 1: PCA and Feature Selection

**SVMs and PCA**

1. please reference source code 1.1.py.

|  |  |
| --- | --- |
| 1 | 6.35634968e+01 |
| 2 | 3.67614817e+01 |
| 3 | 1.64158315e+01 |
| 4 | 1.07162045e+01 |
| 5 | 9.68540952e+00 |
| 6 | 8.94757052e+00 |

2. please reference source code 1.2.py.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | C=1 | C=10 | C=100 | C=1000 |
| K=1 | 0.4423076923076923 | 0.4423076923076923 | 0.4423076923076923 | 0.4423076923076923 |
| K=2 | 0.4423076923076923 | 0.40384615384615385 | 0.40384615384615385 | 0.40384615384615385 |
| K=3 | 0.3846153846153846 | 0.3846153846153846 | 0.3846153846153846 | 0.3846153846153846 |
| K=4 | 0.40384615384615385 | 0.46153846153846156 | 0.46153846153846156 | 0.46153846153846156 |
| K=5 | 0.32692307692307687 | 0.3076923076923077 | 0.25 | 0.25 |
| K=6 | 0.23076923076923073 | 0.28846153846153844 | 0.2692307692307693 | 0.2692307692307693 |

3. please reference source code 1.3.py. From question2 above, we know when k=6 and c=1, we get best result, So for test data:

|  |  |
| --- | --- |
|  | C=1 |
| K=6 | 0.21153846153846156 |
| Without feature selection | 0.2692307692307693 |

From above table, we can see the result with selection k do better than best classifier without out feature selection. Apart from computational benefits, reducing the data’s dimension can also reduce the complexity of the hypothesis class considered and help avoid overfitting. So it will lead to less variance and better performance.

4.

I will pick k by eigenvalue. The eigenvalue shows how much the eigenvector contributes to the training data. For example, when I set the threshold value as 1.0. I will pick all eigenvectors with eigenvalue > 1.0 and discard the others.

By the way, if we need to consider the computation capability, we need to select small enough k to fit our computational requirements (space or time).

**PCA for Feature Selection**

Please reference source code 2.1.py for reference.

Use Gaussian naïve Bayes classifier training, the accuracy on the test set is 0.5769230769230769.

1. To prove a probability distribution. We just need to prove the sum is 1.

For πj = 1/k ∑k i=1 vi j2 and each eigenvector’s square is 1, then

∑n j=1πj = 1/k ∑n j=1∑k i=1 vi j2 = 1/k ∑k i=1 vi2 = 1/k \* k = 1

So we have function π can define a probability distribution.

1. Please take source code 2.2.py for reference.

K:1

[0.45499999999999957, 0.44307692307692287, 0.44480769230769207, 0.44769230769230745, 0.44115384615384606, 0.4484615384615382, 0.43673076923076887, 0.43749999999999956, 0.4446153846153844, 0.4378846153846155, 0.43749999999999983, 0.43711538461538424, 0.4394230769230767, 0.4396153846153843, 0.4373076923076921, 0.43769230769230766, 0.435, 0.43576923076923046, 0.43576923076923085, 0.43826923076923063]

k: 2

[0.44653846153846116, 0.44038461538461526, 0.4282692307692305, 0.41942307692307673, 0.43634615384615366, 0.4336538461538463, 0.42999999999999994, 0.43365384615384595, 0.4307692307692307, 0.4390384615384614, 0.4365384615384614, 0.4313461538461537, 0.42788461538461514, 0.42711538461538423, 0.43211538461538446, 0.4276923076923074, 0.4321153846153844, 0.43, 0.432307692307692, 0.43346153846153834]

k: 3

[0.4544230769230769, 0.42865384615384605, 0.4284615384615382, 0.43307692307692297, 0.42442307692307707, 0.435576923076923, 0.42846153846153817, 0.4319230769230768, 0.4274999999999997, 0.43230769230769217, 0.43480769230769206, 0.43211538461538446, 0.4346153846153843, 0.4359615384615384, 0.42826923076923074, 0.4355769230769229, 0.4369230769230766, 0.4374999999999997, 0.4353846153846153, 0.43692307692307675]

k: 4

[0.44923076923076904, 0.44, 0.44576923076923053, 0.43769230769230744, 0.43403846153846154, 0.4332692307692306, 0.4471153846153845, 0.4371153846153844, 0.44346153846153835, 0.4340384615384613, 0.43115384615384605, 0.4305769230769227, 0.4324999999999997, 0.44076923076923064, 0.4388461538461535, 0.4359615384615381, 0.4398076923076924, 0.43826923076923063, 0.43999999999999967, 0.44038461538461504]

k: 5

[0.43403846153846115, 0.42057692307692285, 0.42019230769230753, 0.4084615384615384, 0.408653846153846, 0.40557692307692295, 0.4084615384615384, 0.40384615384615385, 0.4001923076923078, 0.4174999999999997, 0.41134615384615375, 0.4034615384615381, 0.40461538461538443, 0.4074999999999999, 0.40999999999999964, 0.4028846153846152, 0.4105769230769228, 0.41711538461538455, 0.40730769230769204, 0.4119230769230768]

k: 6

[0.4232692307692305, 0.42711538461538445, 0.41384615384615403, 0.413076923076923, 0.4073076923076922, 0.4048076923076922, 0.41057692307692295, 0.41211538461538444, 0.41711538461538455, 0.4159615384615384, 0.40826923076923044, 0.4211538461538462, 0.41365384615384604, 0.41249999999999987, 0.41211538461538455, 0.4192307692307691, 0.41365384615384604, 0.41346153846153816, 0.40749999999999964, 0.4190384615384613]

k: 7

[0.42923076923076914, 0.4275, 0.4107692307692307, 0.40557692307692306, 0.40557692307692306, 0.3982692307692309, 0.4017307692307691, 0.4046153846153846, 0.40903846153846124, 0.40807692307692284, 0.41134615384615375, 0.4067307692307693, 0.4019230769230769, 0.4051923076923076, 0.4105769230769228, 0.41615384615384593, 0.4134615384615384, 0.41596153846153816, 0.4086538461538459, 0.4082692307692307]

k: 8

[0.4342307692307689, 0.4159615384615384, 0.4159615384615384, 0.39307692307692277, 0.4067307692307693, 0.40423076923076906, 0.4096153846153847, 0.403076923076923, 0.3976923076923074, 0.4198076923076919, 0.40480769230769204, 0.40923076923076906, 0.4171153846153841, 0.40615384615384614, 0.4013461538461536, 0.4173076923076921, 0.40942307692307695, 0.4128846153846155, 0.4051923076923076, 0.41999999999999976]

k: 9

[0.4323076923076921, 0.41807692307692307, 0.4067307692307691, 0.40211538461538443, 0.4069230769230768, 0.4123076923076925, 0.4084615384615385, 0.40711538461538466, 0.40519230769230774, 0.4069230769230768, 0.4146153846153844, 0.4132692307692307, 0.41384615384615364, 0.40884615384615364, 0.4119230769230768, 0.41596153846153827, 0.40999999999999986, 0.4098076923076922, 0.4148076923076921, 0.4157692307692306]

k: 10

[0.44788461538461505, 0.42192307692307696, 0.40461538461538477, 0.40730769230769226, 0.4138461538461538, 0.40653846153846146, 0.4153846153846154, 0.4130769230769228, 0.40807692307692295, 0.4144230769230768, 0.41192307692307667, 0.4007692307692306, 0.4192307692307693, 0.41615384615384593, 0.411730769230769, 0.4144230769230768, 0.4165384615384613, 0.40980769230769226, 0.4157692307692306, 0.417884615384615]

3.

Yes. This provide an alternative solution to naïve Bayes.

Advantage: Most times this approach does as better as naïve Bayes method. This approach requires much less probability distribution than naïve Bayes method. It needs less time and resources for calculation. When handle multiple attributes, it does not require all attributes independent as naïve Bayes.

Disadvantage: Since it does not use all attributes, it may has higher bias than naïve Bayes. We can see as k and s grows, the error rate decrease. So, we may need to select some parameters for it.

Problem 2: Spectral Clustering

**The Basic Algorithm**

1. Please take source code 3.1.py for reference.

L = D – A = [ [∑n i=1A[j,1] – A[1,1], -A[1,2], -A[1,3]……-A[1,n]]

[-A[2,1], ∑n i=1A[j,2] – A[2,2], -A[2,3] …… -A[2,n]]

……

[-A[n,1], -A[n,2], -A[n,3] …… ∑n i=1A[j,n] – A[n,n]] ]

To prove L as positive semidefinite, we just need to prove:

xT L x = λ >=0,

xT L x = ∑n i=1∑n j=1A[i,j](xi2 - xixj) then

2 xT L x = ∑n i=1∑n j=1A[i,j](xi2 - 2xixj + xj2) = ∑n i=1∑n j=1A[i,j](xi - xj)2 >= 0

So, it proves that L is positive semidefinite.

1.~5. Use python:

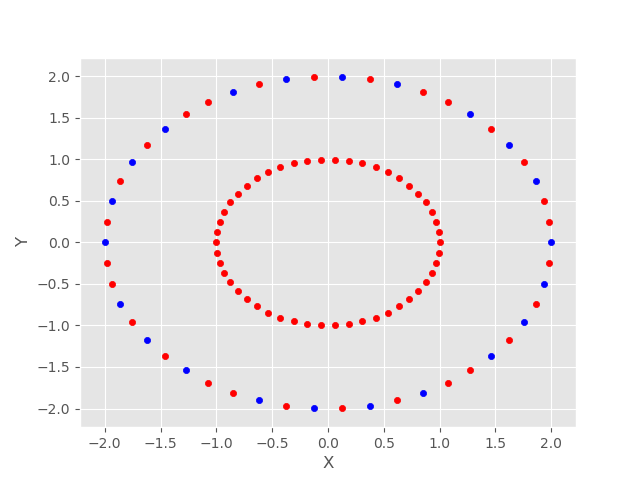
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import math  import numpy as np  import sklearn.cluster as cluster  #varianace is σ  def basic\_algorithm(datas, variance, k):  len = datas.shape[0]  A = np.zeros((len,len))  for i in range(len):  for j in range(i,len):  A[i,j] = A[j,i] = math.exp(-0.5 / math.pow(variance,2) \* np.dot(datas[i] - datas[j],datas[i] - datas[j]))  D = np.zeros((len,len))  D\_tmp = np.sum(A,axis=0)  for i in range(len):  D[i,i] = D\_tmp[i]  L = D - A  print("L:",L)  eigenvalue,eigenvector = np.linalg.eig(L)  print("eigenvector:",eigenvector)  V = eigenvector[:,-k:]  estimator = cluster.KMeans(n\_clusters = k)  estimator.fit(V)  C\_tmp = estimator.predict(V)  C = [[] for i in range(k)]  for i in range(C\_tmp.size):  C[C\_tmp[i]].append(i)  print("C:",C) |

**A Simple Comparison**

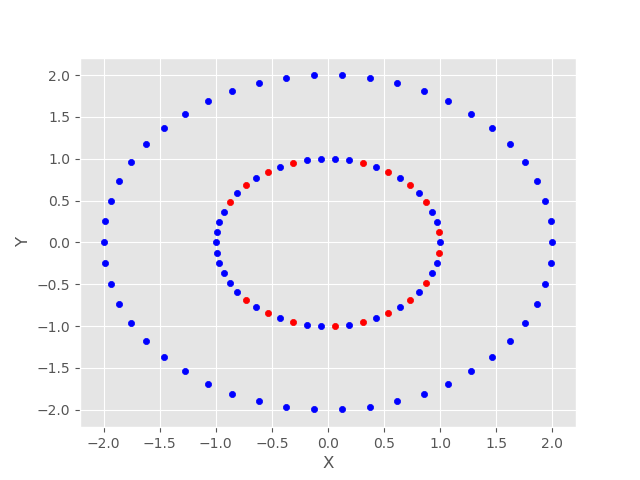
1. Please take source code 4.1.py for reference.

I use k=2 and σ = 0.01, 0.5

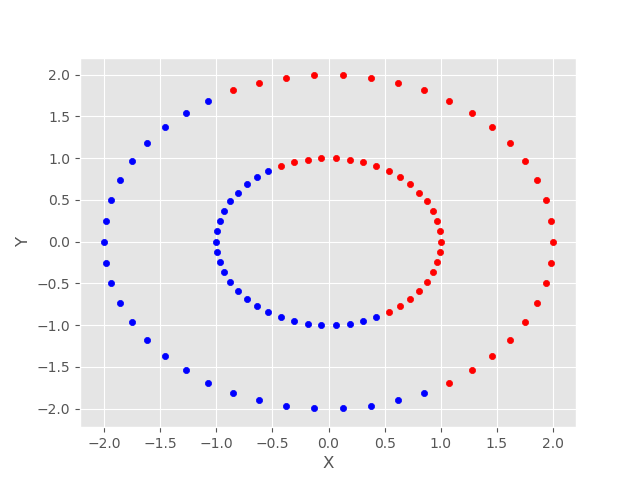
1. When use spectral clustering algorithm and σ = 0.01:



When use spectral clustering algorithm and σ = 0.5:



1. When use k-mean algorithm:



As we can see above, when σ = 0.01 the points selected by spectral clustering algorithm clearly are more concentrated than the points selected by k-means algorithm. So there is no k-means solution that performs this well.

**Partitioning Images**

1~2.

Please reference source code 5.1.py

Use spectral clustering with σ = 0.05:



Use kmean clustering:

