

Neural Networks

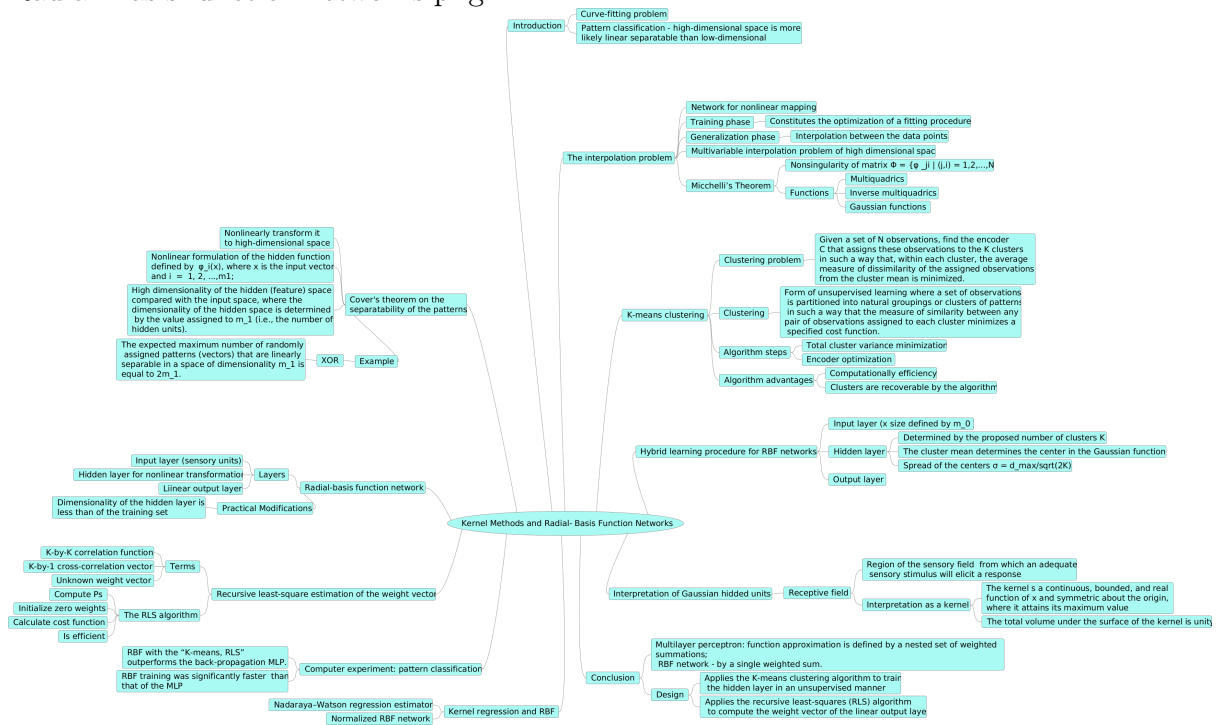
- Homework 7 -

Petr Lukin, Evgeniya Ovchinnikova

Lecture date: 14 November 2016

1 Mind map

Figure 1: Mind map. Chapter 5 from Haykin's book. A zoomed version is attached as Radial-BasisFunctionNetworks.png



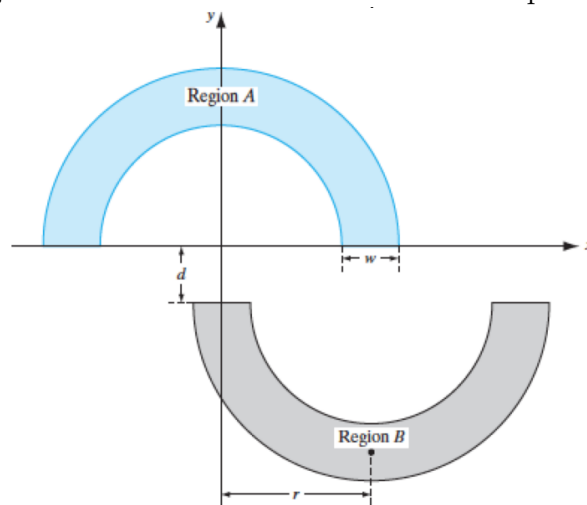
2 Exercises

2.1 Exercise 5.10

The purpose of this computer experiment is to investigate the clustering process performed by the K-means algorithm. To provide insight into the experiment, we fix the number of clusters at $K=6$, but vary the vertical separation between the two moons in Fig. 2. Specifically, the requirement is to do the following, using an unlabeled training sample of 1,000 data points picked randomly from the two regions of the double-moon pictured in Fig. 2:

- Experimentally, determine the mean $\hat{\mu}_j$ and variance $\hat{\sigma}_j^2$, $j = 1, 2, \dots, 6$, for the sequence of eight uniformly spaced vertical separations starting at $d = 1$ and reducing them by one till separation $d = -6$ is reached.
- In light of the results obtained in previous part, comment on how the mean $\hat{\mu}_j$ of cluster j is affected by reducing the separation d for $j = 1, 2$ and 3 .
- Plot the variance $\hat{\sigma}_j^2$ versus the separation d for $j = 1, 2, \dots, 6$.
- Compare the common σ^2 computed in accordance with the empirical formula of the equation $\sigma = \frac{d_{max}}{\sqrt{2K}}$ with the trends exhibited in the plots obtained in (c).

Figure 2: The double moon classification problem



Solution:

To solve this problem first we've put some values to the moons' radii and w : $r_{Large} = 30$, $r_{Small} = 23$, $w = 5$. So, the equations to describe these moons are the following: large top circle $y = \sqrt{(30^2 - x^2)}$, small top circle $y = \sqrt{(25^2 - x^2)}$, large bottom circle $y = -d - \sqrt{(30^2 - (x - 27.5)^2)}$, large bottom circle $y = -d - \sqrt{(25^2 - (x - 27.5)^2)}$.

```

1 import numpy as np
2 from sklearn.cluster import KMeans
3
4 #double moon parameters. w and radii are not given, so we
   assume that w = 5, r = 25, R = 30.
5 # So, the moons equations are the following:
6 # large top circle y = sqrt(30^2 - x^2)
7 # small top circle y = sqrt(25^2 - x^2)
8 # large bottom circle y = -d - sqrt(30^2 - (x - 27.5)^2)
9 # large bottom circle y = -d - sqrt(25^2 - (x - 27.5)^2)
10 w = 5
11 d = np.array([1,0,-1,-2,-3,-4,-5,-6])
12 r = 25
13 R = 30

```

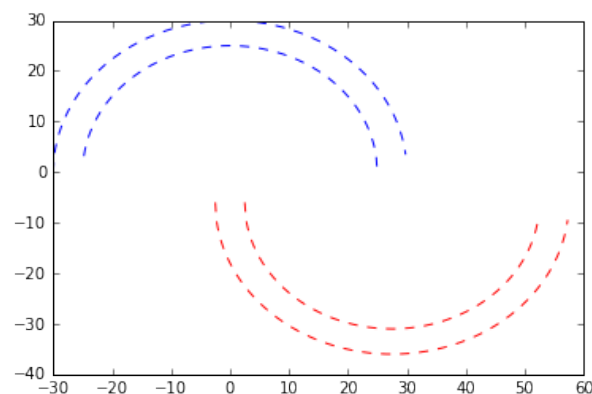
```

14
15 #the moons
16
17 import matplotlib.pyplot as plt
18
19 %matplotlib inline
20
21 # evenly sampled time at 200ms intervals
22 t_up = np.arange(-30., 30, 0.2)
23 t_bot = np.arange(-2.5, 57.5, 0.2)
24
25 # red dashes, blue squares and green triangles
26 plt.plot(t_up, np.sqrt(R**2 - t_up**2), 'b--', t_up, np.sqrt(r
    **2 - t_up**2), 'b--',
27         t_bot, - np.sqrt(R**2 - (t_bot - 27.5)**2) + d[7], 'r--',
    t_bot, - np.sqrt(r**2 - (t_bot - 27.5)**2) + d[7], 'r--')
28 plt.show()

```

With these equations we get the moons depicted in Fig. 3.

Figure 3: The double moon plotted with chosen parameters



Then we generate the 1000 points inside these circles - by creating random arrays for the radiuses between the radius of a large and a small circles:

```

1 # train points generation
2 randomRad = np.random.randint(250, high=300, size=1000)/10.0
3 randomAngle = np.random.randint(0, high=360, size=1000)
4
5 def generate_points(d):
6     X_train = np.array([[0, 27]])
7     for i in range(len(randomAngle)):
8         angle = math.pi*randomAngle[i]/180
9         if (randomAngle[i] < 180):
10             x = randomRad[i]*math.sin(angle)
11             y = randomRad[i]*math.cos(angle)
12         else:
13             x = randomRad[i]*math.sin(angle) + 27.5

```

```

14         y = randomRad[i]*math.cos(angle) - d
15         point = np.array([x,y])
16         X_train = np.row_stack((X_train, point))
17     return X_train
18
19
20 X = np.array([generate_points(d[0]), generate_points(d[1]),
21              generate_points(d[2]), generate_points(d[3]),
22              generate_points(d[4]), generate_points(d[5]),
23              generate_points(d[6]), generate_points(d[7])])

```

(a) After it we need to initialize centroids. We choose 6 centroids for each d randomly from the generated points:

```

1 #Second step is centroids initialization. We choose 6 centroids
  for each d randomly from the points:
2 def init_centroids(X):
3     centroids = np.array([])
4     indices = np.random.randint(0, high=1000, size=6)
5     for i in range(K):
6         centroids = np.append(centroids, X[indices[i]])
7     return centroids.reshape(K,2)
8 centroids = np.array([])
9 for i in range(len(d)):
10     centroids = np.append(centroids, init_centroids(X[i]))
11 print centroids.reshape(len(d),K,2)
12
13
14 [[ [ 24.0315424    6.8909339 ]
15    [ 17.37620376   28.81490158]
16    [ 14.00600244  -21.45778328]
17    [  4.13496686   10.44008615]
18    [ 23.11084857  -10.77676567]
19    [ 27.13230009  -12.0800783 ]]
20
21 ...
22
23 [[ 28.2827604    0.98765576]
24    [  0.42616763   4.39267054]
25    [  0.33280959 -16.42850733]
26    [  2.83046993 -16.47159424]
27    [  1.00403608  -5.53751123]
28    [ 16.23646609  20.05036581]]]
29
30
31 plt.grid(True)
32 plt.plot(X[1][:,0], X[1][:,1], 'r^', centroids[0][:,0], centroids
33          [0][:,1], 'bo')
34 plt.show()

```

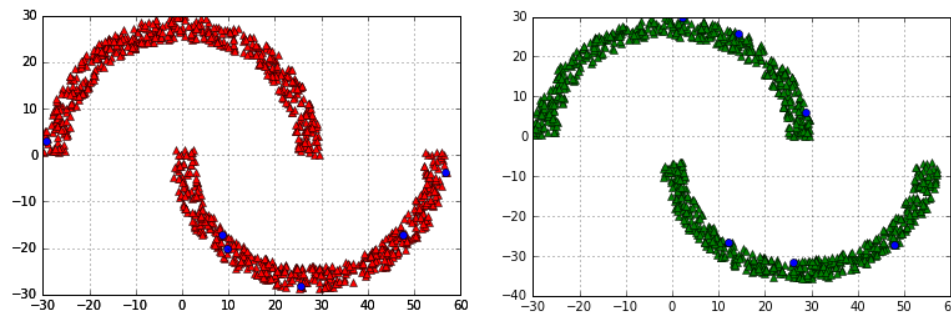
```

34
35 plt.grid(True)
36 plt.plot(X[7][:,0], X[7][:,1], 'g^', centroids[7][:,0], centroids
    [7][:,1], 'bo')
37 plt.show()

```

The centroids for first and last ds are depicted in Fig. 4.

Figure 4: The double moon (red - for $d = 1$ and green - for $d = -6$ triangles) with centroids (blue points)



The next step is to assign points to clusters by finding the shortest distance between the points and clusters:

```

1 import scipy.spatial.distance
2 #find the closest to centroids points
3 def assign_points_to_centroids(X, centroid_set):
4     shortest = np.array([])
5     distances = scipy.spatial.distance.cdist(X, centroid_set)
6     for i in range(len(distances)):
7         point_to_cluster = np.array([np.argmin(distances[i]),
8                                       np.amin(distances[i])])
9         shortest = np.append(shortest, point_to_cluster)
10    return shortest.reshape(len(distances), 2)
11
12 points_in_clusters = np.array([])
13 for i in range(len(d)):
14     points_in_clusters = np.append(points_in_clusters,
15                                   assign_points_to_centroids(X[i], centroids[i]))
16
17 points_in_clusters = points_in_clusters.reshape(len(d), len(X
18 [0]), 2)
19
20 print points_in_clusters
21
22
23
24

```

```

[[[ 3.          8.01371165]
  [ 0.          23.24707832]
  [ 1.          32.58619713]
  ...,
  [ 4.          8.02426658]
  [ 5.          10.33305738]
  [ 0.          3.52343674]]

```

```

25
26 ...
27
28 [[ 1.          3.42927548]
29  [ 0.          13.41239277]
30  [ 3.          18.72193546]
31  ...,
32  [ 0.          3.43011793]
33  [ 1.          12.20407394]
34  [ 5.          3.42414427]]]

```

And then follows the actual learning. We need to move centroids in such a way that they would be in the middle of the point cloud assigned to them:

```

1 def centroids_moving(X, points_in_clusters, centroid_set):
2     return np.array([X[points_in_clusters[:,0] == k].mean(axis
3                       =0) for k in range(len(centroid_set))])
4 new_position_centr = np.array([])
5 for i in range(len(d)):
6     new_position_centr = np.append(new_position_centr,
7                                     centroids_moving(X[i], points_in_clusters[i], centroids[i]
8                                                         ))
9 new_position_centr = new_position_centr.reshape(len(d),K,2)
10 print new_position_centr
11
12 [[[ -13.82919874  19.64638503]
13   [ 44.62527485 -17.08127271]
14   [ 49.2583624  -2.58098819]
15   [ 12.81568895   5.20702855]
16   [ 26.75005756 -25.69362956]
17   [ 13.03522465 -21.93359926]]
18 ...
19   [ 15.65318603  18.39961297]
20   [-23.20695024  12.64083583]
21   [  9.79272096 -22.96052392]
22   [ 31.122512   -32.82182453]
23   [ 48.62381334 -21.15032426]
24   [-10.3448574   24.93610957]]]

```

Using previous step update the centroids coordinates till the difference between new and old ones is less than 0.1:

```

1 new_position_centr = np.array([])
2 iterations = np.array([])
3 for i in range(len(d)):
4     iter = 0
5     old_position_centr = centroids[i]
6     dists = 100
7     cluster_points = assign_points_to_centroids(X[i],
8                                                  old_position_centr)

```

```

8     while(dists > 0.1):
9         iter+=1
10        #update cluster
11        new_position_c = centroids_moving(X[i], cluster_points,
12                                           old_position_centr)
13        #update points
14        cluster_points = assign_points_to_centroids(X[i],
15                                                    new_position_c)
16        dists = 0
17        for j in range(K):
18            dists += scipy.spatial.distance.euclidean(
19                old_position_centr[j], new_position_c[j])
20        old_position_centr = new_position_c
21    iterations = np.append(iterations, iter)
22    new_position_centr = np.append(new_position_centr,
23                                   new_position_c)
24
25    new_position_centr = new_position_centr.reshape(len(d),K,2)
26    print "iterations"
27    print iterations
28    print "centroids"
29    print new_position_centr
30
31    iterations
32    [ 24.  30.  25.   8.  17.  12.  11.   7.]
33    centroids
34    [[[ 49.40952006 -12.93490955]
35     [-21.9539597   14.08117102]
36     [ 25.82518642 -25.17856609]
37     [ 23.05741957  12.17402433]
38     [  4.14254172 -11.15832509]
39     [  0.45427766  26.09350205]]
40
41    ...
42
43    [[  4.98230105 -19.26547119]
44     [ 50.47929746 -18.7701339 ]
45     [  2.37019789  25.9815445 ]
46     [ 28.61314336 -32.18748749]
47     [ 23.42569304  11.6670059 ]
48     [-21.33635336  14.82847459]]]

```

The resulting centroids for the first and last ds are shown in Fig. 5.

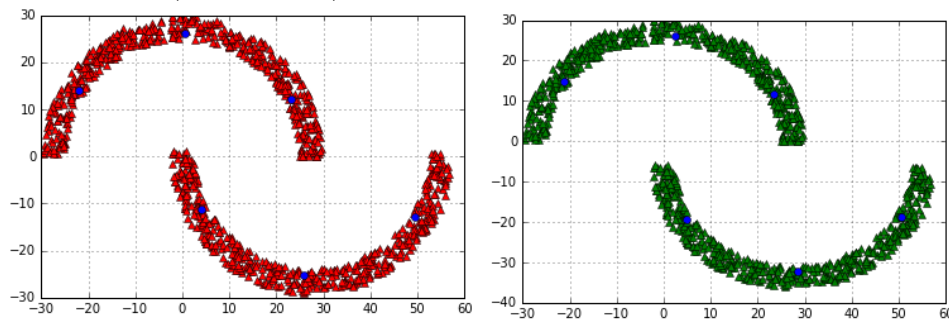
(a) For mean and variance calculation we used the following code:

```

1  #a mean and variance calculation
2
3  def pair_dist(arr1, arr2):

```

Figure 5: The double moon (red - for $d = 1$ and green - for $d = -6$ triangles) with final calculated centroids (blue points)



```

4     summ = 0;
5     for i in range(len(arr1)):
6         summ += scipy.spatial.distance.euclidean(arr1[i], arr2)
7     return summ/len(arr1)
8 def var_for_cluster(arr1, arr2, mean):
9     summ = 0;
10    for i in range(len(arr1)):
11        summ += (scipy.spatial.distance.euclidean(arr1[i], arr2)
12                - mean)**2
13    return summ/(len(arr1) - 1)
14
15 def mean_calculation(X, points_in_clusters, centroid_set):
16     return np.array([pair_dist(X[points_in_clusters[:,0] == k],
17                               centroid_set[k]) for k in range(len(centroid_set))])
18
19 def variance_calculation(X, points_in_clusters, centroid_set,
20                          mean):
21     return np.array([var_for_cluster(X[points_in_clusters[:,0]
22                               == k], centroid_set[k], mean[k]) for k in range(len(
23                               centroid_set))])
24
25 mean = np.array([])
26 variance = np.array([])
27 for i in range(len(d)):
28     points = assign_points_to_centroids(X[i], new_position_centr
29                                         [i])
30     mean_tmp = mean_calculation(X[i], points, new_position_centr
31                                [i])
32     mean = np.append(mean, mean_tmp)
33     variance_tmp = variance_calculation(X[i], points,
34                                         new_position_centr[i], mean_tmp)
35     variance = np.append(variance, variance_tmp)
36 mean = mean.reshape(len(d), len(mean_tmp))
37 variance = variance.reshape(len(d), len(variance_tmp))
38 print mean
39 print variance

```


So, we've obtained the following means : $[[8.11551501 6.89620468 7.05776613 7.38103096$
 $7.7538904 7.69813644] [5.45685966 5.78160551 4.91614198 10.67401462 10.94663218 5.7996544$
 $] [10.94663218 10.67401462 5.7996544 4.91614198 5.45685966 5.78160551] [5.12399471$
 $5.9802248 10.83794335 10.47906663 5.65293167 5.31845917] [5.96931319 10.83794335$
 $5.03875964 5.58564629 5.44074919 10.47906663] [5.52657573 4.91614198 10.67401462$
 $5.70120506 10.94663218 5.7996544] [7.7538904 7.05776613 7.31133375 7.46133993 8.11551501$
 $7.25492557] [7.31133375 8.29599059 7.46133993 6.74378599 7.25492557 7.71256627]]$

and variances:

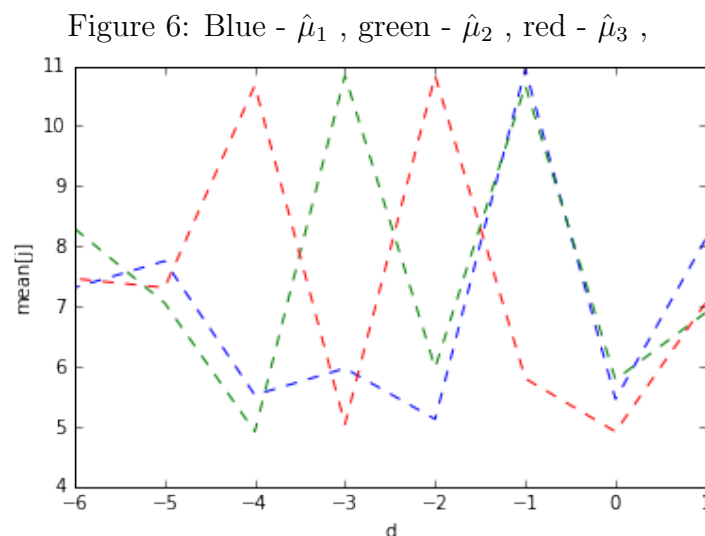
$[[14.62263462 16.0437904 14.86025695 15.14036738 12.38129185 15.15804983] [7.84305872$
 $10.28209152 6.38976484 29.80877893 26.90941454 9.23635644] [26.90941454 29.80877893$
 $9.23635644 6.38976484 7.84305872 10.28209152] [6.0968976 7.93027525 32.81736714 27.11795529$
 $8.33838342 7.67798775] [8.42205311 32.81736714 5.78362442 7.97020471 8.42114792 27.11795529]$
 $[8.15256191 6.38976484 29.80877893 10.01222759 26.90941454 9.23635644] [12.38129185$
 $14.86025695 13.81686673 17.58098457 14.62263462 13.80546138] [13.81686673 14.57946065$
 $17.58098457 13.62149918 13.80546138 14.16774844]]$

for $(d_1 - d_8)$.

(b) In Fig. 6 we show the mean - d dependency.

```

1 #b
2 plt.xlabel('d')
3 plt.ylabel('mean[j]')
4 colors = ['b--', 'g--', 'r--']
5
6 for i in range(3):
7     plt.plot(d, mean[:,i], colors[i])
8 plt.show()
```



The mean values show symmetry.

(c) See Fig. 7.

```

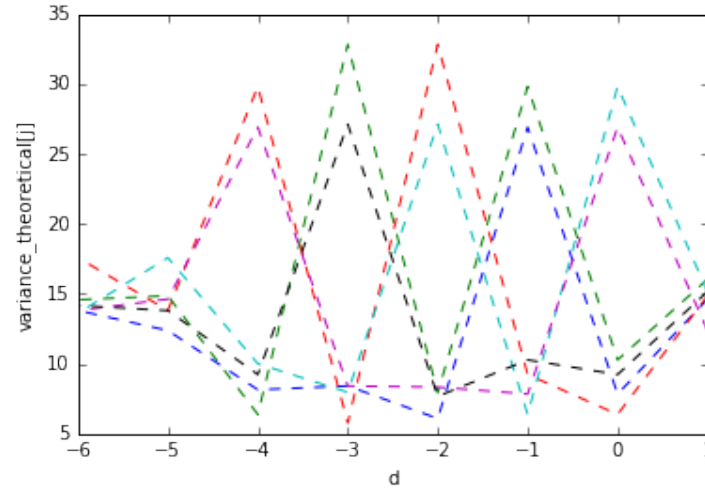
1 #c
2 plt.xlabel('d')
```

```

3 plt.ylabel('variance[j]')
4 colors = ['b--', 'g--', 'r--', 'c--', 'm--', 'k--']
5
6 for i in range(6):
7     plt.plot(d, variance[:,i], colors[i])
8 plt.show()

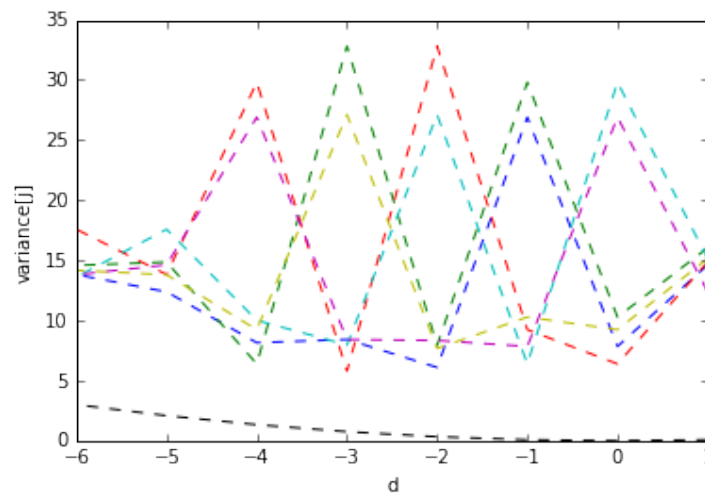
```

Figure 7: Blue - $\hat{\sigma}_1^2$, green - $\hat{\sigma}_2^2$, red - $\hat{\sigma}_3^2$, cian - $\hat{\sigma}_4^2$, magenta - $\hat{\sigma}_5^2$, black - $\hat{\sigma}_6^2$



(d) In Fig. 8 we show the theoretical σ^2 and experimental $\hat{\sigma}^2$ s.

Figure 8: Blue - $\hat{\sigma}_1^2$, green - $\hat{\sigma}_2^2$, red - $\hat{\sigma}_3^2$, cian - $\hat{\sigma}_4^2$, magenta - $\hat{\sigma}_5^2$, yellow - $\hat{\sigma}_6^2$, black - $\hat{\sigma}_{theor}^2$



2.2 Exercise 5.11

The purpose of this second experiment is to compare the classification performance of two hybrid learning algorithms: the "K-means,RLS" algorithm investigated in Section 5.8 and the "K-means,LMS" algorithm investigated in this problem.

As in section 5.8, assume the following:

Number of hidden Gaussian units:20

Number of training samples:1,000 data points

Number of testing samples: 2,000 data points

Let the learning-rate parameter of the LMS algorithm be annealed linearly from 0.6 down to 0.01.

- (a) Construct the decision boundary computed for the "K-means,LMS" algorithm for the vertical separation between the two moons in Fig.2 set at $d = -5$.
- (b) Repeat the experiment for $d = -6$.
- (c) Compare the classification results obtained using the "K-means,LMS" algorithm with those of the "K-means,RLS" algorithm studied in Section 5.8.
- (d) Discuss how, in general, the complexity of the "K-means,LMS" algorithm compares with that of the "K-means,RLS" algorithm.