Lab 2: Support Vector Machines

DD2431 - Machine Learning

Mélanie Sedda

October 6, 2015

1 Linear classifier

The goal of a linear classifier is to find a vector w such that all the points of the first class ($t = 1$) are such that $w^T x \geq 0$ and all the points of the second class ($t = -1$) are such that $w^T x \leq 0$. The hyperplane $w^T x = 0$ is the boundary between the two classes.

We can notice that multiplying w by some constant doesn't change the classifier. So an equivalent problem is to say that we want $w^T x \geq 1$ for the points such that $t = 1$ and $w^T x \leq -1$ for the points such that $t = -1$ (we just need to scale w) which can be summarized in the equation

$$tw^t x \geq 1.$$

2 Maximal margin

If several boundaries satisfy this property, we would like to find the best one. The best one is defined as the one that has the bigger margin, where the margin is the distance between the separating hyperplane and the closest point to it. If some points p and q are such that $w^T p = 1$ and $w^T q = -1$ then the projection of the length of the vector $p - q$ on the axis perpendicular to the plane must be equal to $2d$ since both p and q are at distance d from the hyperplane. We can then write

$$2d = \frac{w^T(p-q)}{||w||} = \frac{w^T p - w^T q}{||w||} = \frac{2}{||w||}$$

and see that maximizing d is equivalent to minimizing $||w||$.

So if we want to find the best linear classifier we need to solve the following optimization problem

$$\begin{aligned} \min_w \quad & ||w|| \\ & t_i w^T x_i \geq 1 \quad \forall i \end{aligned}$$

which can also be written as

$$\begin{aligned} \min_w \quad & \frac{1}{2} w^T w \\ & t_i w^T x_i \geq 1 \quad \forall i. \end{aligned} \tag{1}$$

3 Non-linear classifier

If we want to create a non-linear classifier a trick is to transform the input data. So instead of consider x we will consider $\phi(x)$ for some function ϕ and we will try to separate these values linearly.

4 Kernel trick

If ϕ maps the points into a very high dimensional space, we intuitively feel that the chance that the problem becomes linearly separable are higher (actually if we have as many dimensions as points, the problem is always linearly separable because we can simply use the first dimension to separate one point from the others, then the second one to separate another point from the remaining ones, etc). The kernel trick is to utilize the advantages of a high-dimensional space without actually representing anything high-dimensional. For this, we need that the only operation done in the high-dimensional space is to compute scalar products between pairs of items. The trick is then to express the scalar product in the high dimensional space in function of the original (low-dimensional) representation, i.e.

$$\phi(x) \cdot \phi(y) = \mathcal{K}(x, y).$$

The function $\mathcal{K}$ is called a kernel. So kernels are functions that correspond to a scalar product is some high dimensional space.

Common kernels are

Linear kernel

$$\mathcal{K}(x, y) = x \cdot y + 1$$

Polynomial kernel

$$\mathcal{K}(x, y) = (x \cdot y + 1)^p$$

RBF kernel

$$\mathcal{K}(x, y) = e^{-\frac{||x-y||^2}{2\sigma^2}}$$

Sigmoid kernel

$$\mathcal{K}(x, y) = \tanh(kx \cdot y - \delta)$$

Then additional +1 terms are here to allow hyperplanes with a biais, i.e. that that do not necessarily contain the origin.

But how do we make some scalar products appear? For this, we will consider the dual problem of (2). There are as many dual variables as primal constraints. Let us call α_i those variables. The constraints on these variables are simply $\alpha_i \geq 0$ and the objective becomes

$$\max_{\alpha} \min_{w} L = \max_{\alpha} \min_{w} \frac{1}{2} w^T w - \sum_i \alpha_i (t_i w^T \phi(x_i) - 1).$$

Since

$$\frac{\partial L}{\partial w} = w - \sum_i \alpha_i t_i \phi(x_i)$$

the w that minimizes L is

$$w = \sum_i \alpha_i t_i \phi(x_i),$$

$$\min_w \; L = \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j t_i t_j \phi(x_i)^T \phi(x_j) - \sum_{i,j} \alpha_i \alpha_j t_i t_j \phi(x_i)^T \phi(x_j) + \sum_i \alpha_i$$

and the dual problem is

$$\begin{aligned} \max_\alpha \; & \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j t_i t_j \phi(x_i)^T \phi(x_j) \\ & \alpha_i \geq 0 \quad \forall i. \end{aligned}$$

We see that we indeed only have a scalar in the high dimensional space and it can be replaced by a kernel function. We can also express this problem is a more compact form using vectors and matrices. The canonical form of the dual problem is

$$\begin{aligned} \min_\alpha \; & \frac{1}{2}\alpha^T P \alpha - \alpha \cdot e \\ & \alpha \geq 0. \end{aligned} \tag{2}$$

where $P_{i,j} = t_i t_j \mathcal{K}(x_i, x_j)$.

5 Slack variables

Even if the problem is not separable, we could want to find a reasonable boundary. We allow some points to be misclassified but we had a penalty in that case. The primal problem can then be formulated as

$$\begin{aligned} \min_w \quad & \frac{1}{2} w^T w + C \sum_i \xi_i \\ & t_i w^T x_i \geq 1 - \xi_i \quad \forall i. \end{aligned} \tag{3}$$

where the ξ_i are called slack variables and the dual problem becomes

$$\begin{aligned} \min_\alpha \; & \frac{1}{2}\alpha^T P \alpha - \alpha \cdot e \\ & 0 \leq \alpha \leq C. \end{aligned} \tag{4}$$

with the same P as before. C is a parameter that can be tuned. If C is large, then the misclassifications are highly penalized.

6 Implementation in Python

Listing 1: svm.py

```
__author__ = 'melaniesedda'
import numpy as np
import pylab
import random
from kernels import *
from cvxopt.solvers import qp
from cvxopt.base import matrix
```

```
8
9  def svm(classA, classB, kernel_function, args, C):
10     # Shuffle data
11     data = classA + classB
12     random.shuffle(data)
13     pylab.hold(True)
14     pylab.plot([p[0] for p in classA], [p[1] for p in classA], 'bo')
15     pylab.plot([p[0] for p in classB], [p[1] for p in classB], 'ro')
16     N = np.shape(data)[0]
17
18     # Create matrices (rajouter des matrix partout)
19     P = [[p[2]*q[2]*kernel_function(p[0:2], q[0:2], args) for p in data] \
20          for q in data]
21     print P
22     q = -1*np.ones(N)
23     G = -1*np.identity(N)
24     h = np.zeros(N)
25
26     # Solve the optimization problem
27     r = qp(matrix(P) , matrix(q) , matrix(G) , matrix(h))
28     alpha = list(r['x'])
29     for i in range(len(alpha)):
30         if alpha[i] <= math.pow(10, -5):
31             alpha[i] = 0
32     print [data[i] for i in range(len(data)) if alpha[i] != 0]
33
34     # Plot the boundary
35     xrange = np.arange(-4, 4, 0.05)
36     yrange = np.arange(-4, 4, 0.05)
37     grid = matrix([[indicator([x, y], alpha, data, kernel_function, args) \
38                     for y in yrange] for x in xrange])
39     pylab.contour(xrange, yrange, grid, (-1.0, 0.0, 1.0), \
40                   colors=('red', 'black', 'blue'), linewidths=(1, 3, 1))
41     pylab.show()
42
43
44 if __name__ == "__main__":
45     test = 7
46
47     # Linear
48     if test == 1:
49         classA = [(-1.0, 1.0, 1.0), (-1.0, 2.0, 1.0)]
50         classB = [(1.0, -1.0, -1.0), (2.0, 0.0, -1.0)]
51         svm(classA, classB, linear_kernel, [0], 0)
52
53     if test == 2:
54         classA = [(random.normalvariate(-1.5, 1.0), \
55                    random.normalvariate(0.5, 1.0), 1.0) for i in range(5)]
56         classB = [(random.normalvariate(0.0, 0.5), \
57                    random.normalvariate(-0.5, 0.5), -1.0) for i in range(10)]
58         svm(classA, classB, linear_kernel, [0], 0)
59
60     # Polynomial
61     if test == 3:
62         classA = [(random.normalvariate(2.0, 0.5), \
63                    random.normalvariate(2.0, 0.5), 1.0) for i in range(5)] + \
64                  [(random.normalvariate(0.0, 0.5), \
65                    random.normalvariate(2.5, 0.5), 1.0) for i in range(5)]
66         classB = [(random.normalvariate(0.0, 0.5), \
67                    random.normalvariate(0.0, 0.5), -1.0) for i in range(10)]
68         svm(classA, classB, polynomial_kernel, [2], 0)
69         svm(classA, classB, polynomial_kernel, [3], 0)
70         svm(classA, classB, polynomial_kernel, [4], 0)
```

```
71
72      # RBF
73      if test == 4:
74          classA = [(random.normalvariate(2.0, 0.5), \
75                     random.normalvariate(2.0, 0.5), 1.0) for i in range(5)] + \
76                   [(random.normalvariate(0.0, 0.5), \
77                     random.normalvariate(2.5, 0.5), 1.0) for i in range(5)]
78          classB = [(random.normalvariate(0.0, 0.5), \
79                     random.normalvariate(0.0, 0.5), -1.0) for i in range(10)]
80          svm(classA, classB, rbf_kernel, [1], 0)
81          svm(classA, classB, rbf_kernel, [2], 0)
82          svm(classA, classB, rbf_kernel, [3], 0)
83
84      if test == 5:
85          classA = [(random.normalvariate(-1.5, 0.5), \
86                     random.normalvariate(1.5, 0.5), 1.0) for i in range(5)] + \
87                   [(random.normalvariate(1.5, 0.5), \
88                     random.normalvariate(-1.5, 0.5), 1.0) for i in range(5)] + \
89                   [(random.normalvariate(-1.5, 0.5), \
90                     random.normalvariate(-1.5, 0.5), 1.0) for i in range(5)]
91          classB = [(random.normalvariate(1.5, 0.5), \
92                     random.normalvariate(1.5, 0.5), -1.0) for i in range(5)] + \
93                   [(-1.0, -1.0, -1.0)]
94          svm(classA, classB, rbf_kernel, [1], 0)
95          svm(classA, classB, rbf_kernel, [2], 0)
96          svm(classA, classB, rbf_kernel, [3], 0)
97
98      if test == 6:
99          classA = [(random.normalvariate(-1.5, 0.5), \
100                    random.normalvariate(1.5, 0.5), 1.0) for i in range(5)] + \
101                  [(random.normalvariate(1.5, 0.5), \
102                    random.normalvariate(-1.5, 0.5), 1.0) for i in range(5)] + \
103                  [(random.normalvariate(-1.5, 0.5), \
104                    random.normalvariate(-1.5, 0.5), 1.0) for i in range(5)] + \
105                  [(0.0, 0.0, 1.0)]
106          classB = [(random.normalvariate(1.5, 0.5), \
107                    random.normalvariate(1.5, 0.5), -1.0) for i in range(5)] + \
108                  [(-1.0, -1.0, -1.0)]
109          svm(classA, classB, rbf_kernel, [1], 0)
110          svm(classA, classB, rbf_kernel, [2], 0)
111          svm(classA, classB, rbf_kernel, [3], 0)
112
113      # Sigmoid
114      if test == 7:
115          classA = [(random.normalvariate(-1.5, 0.5), \
116                    random.normalvariate(1.5, 0.5), 1.0) for i in range(5)] + \
117                  [(random.normalvariate(1.5, 0.5), \
118                    random.normalvariate(-1.5, 0.5), 1.0) for i in range(5)] + \
119                  [(random.normalvariate(-1.5, 0.5), \
120                    random.normalvariate(-1.5, 0.5), 1.0) for i in range(5)]
121          classB = [(random.normalvariate(1.5, 0.5), \
122                    random.normalvariate(1.5, 0.5), -1.0) for i in range(5)] + \
123                  [(-0.5, -0.5, -1.0)]
124          svm(classA, classB, sigmoid_kernel, [0.15, -0.5], 0)
125          """svm(classA, classB, sigmoid_kernel, [0.1, -0.5], 0)
126          svm(classA, classB, sigmoid_kernel, [0.1, 0.0], 0)
127          svm(classA, classB, sigmoid_kernel, [0.2, -1], 0)
128          svm(classA, classB, sigmoid_kernel, [0.3, -1], 0)"""
```

Listing 2: kernels.py

```
1  __author__ = 'melaniesedda'
2  import math
```

```
import numpy as np


def linear_kernel(x, y, args):
    return np.dot(x, y) + 1


def polynomial_kernel(x, y, args):
    return math.pow(np.dot(x, y), args[0]) + 1


def rbf_kernel(x, y, args):
    return math.exp(-1/(2*math.pow(args[0], 2)) * \
                    math.pow(np.linalg.norm([x[i]-y[i] for i in range(len(x))]), 2))


def sigmoid_kernel(x, y, args):
    return math.tanh(args[0]*np.dot(x, y) - args[1])

def indicator(x, alpha, data, kernel, args):
    return sum([alpha[i]*data[i][2]*kernel(x, data[i][0:2], args) \
                for i in range(len(alpha))])

if __name__ == "__main__":
    x = np.array([1, 2])
    y = np.array([3, 4])
    print linear_kernel(x, y)
    print polynomial_kernel(x, y, 2)
    print rbf_kernel(x, y, 2)
    print sigmoid_kernel(x, y, 2, 5)
```

7 Results

7.1 Linear kernel

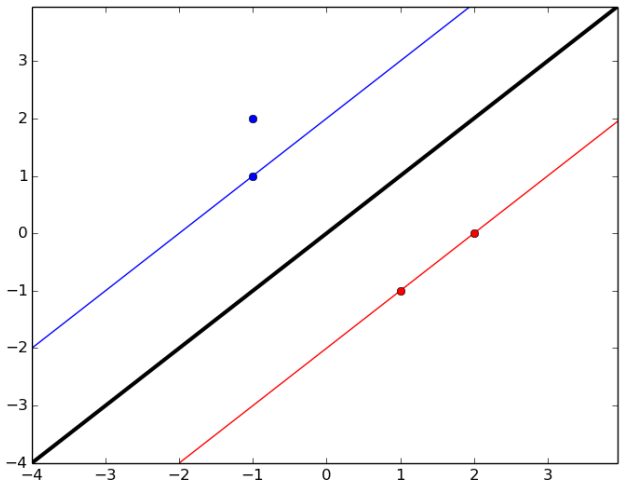


Figure 1: Linear kernel : test 1

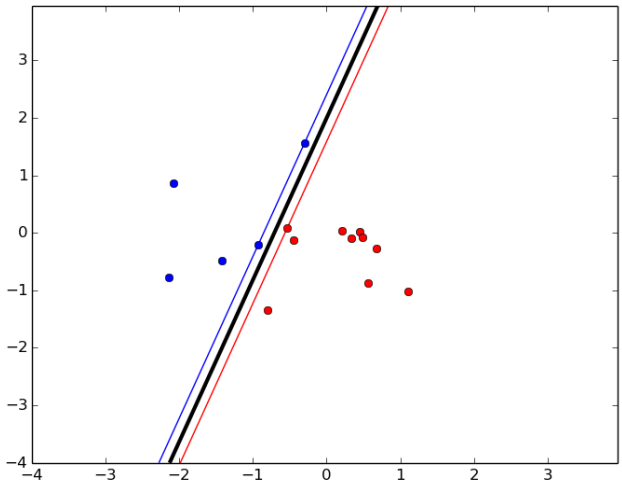
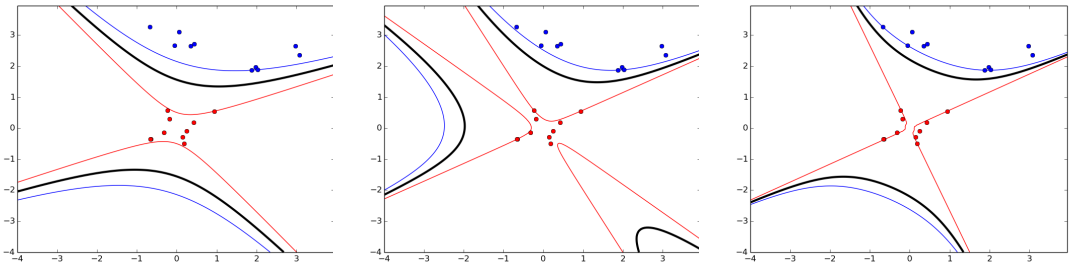


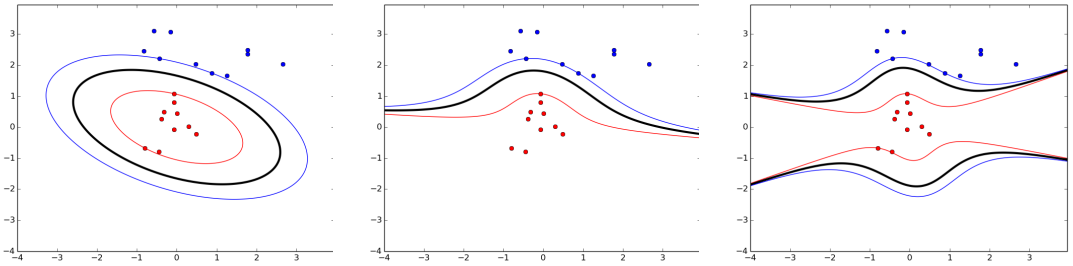
Figure 2: Test 2

7.2 Polynomial kernel



(a) $p = 2$ (b) $p = 3$ (c) $p = 4$

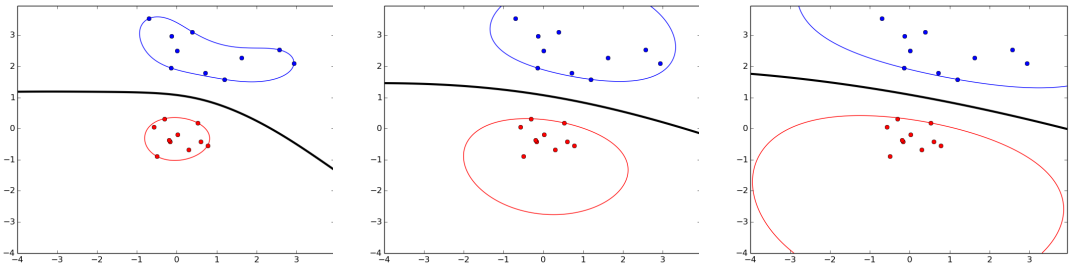
Figure 3: Polynomial kernel : test 1



(a) $p = 2$ (b) $p = 3$ (c) $p = 4$

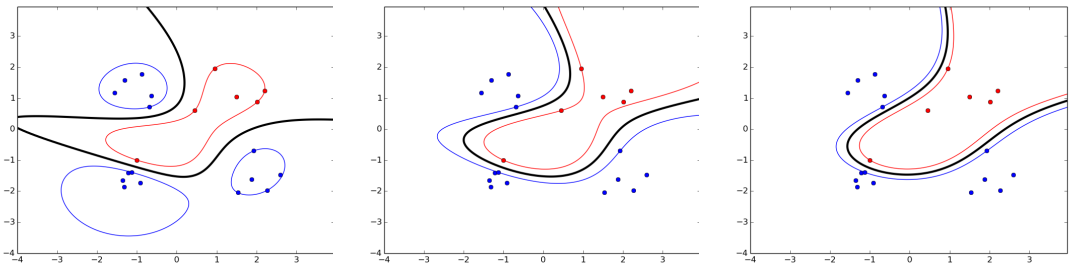
Figure 4: Polynomial kernel : test 2

7.3 RBF kernel



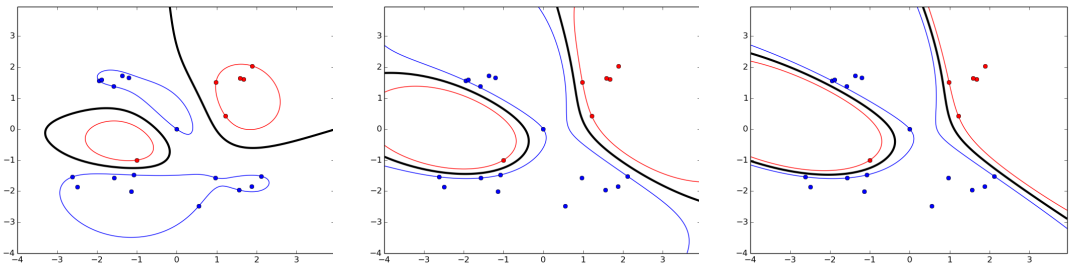
(a) $\sigma = 1$ (b) $\sigma = 2$ (c) $\sigma = 3$

Figure 5: RBF kernel : test 1



(a) $\sigma = 1$ (b) $\sigma = 2$ (c) $\sigma = 3$

Figure 6: RBF kernel : test 2



(a) $\sigma = 1$ (b) $\sigma = 2$ (c) $\sigma = 3$

Figure 7: RBF kernel : test 3

The parameter σ can be used to control the smoothness of the boundary.

7.4 Sigmoid kernel

7.5 Slack variables