Machine Learning I Assignment 9

December 15, 2018

Group Name : XTBQJ

1 Support Vector Machines

In this exercise sheet, you will experiment with training various support vector machines on a subset of the MNIST dataset composed of digits 5 and 6. First, download the MNIST dataset from http://yann.lecun.com/exdb/mnist/, uncompress the downloaded files, and place them in a `data/` subfolder. Install the optimization library CVXOPT (`python-cvxopt` package, or directly from the website `www.cvxopt.org`). This library will be used to optimize the dual SVM in part A.

1.1 Part A: Kernel SVM and Optimization in the Dual

We would like to learn a nonlinear SVM by optimizing its dual. An advantage of the dual SVM compared to the primal SVM is that it allows to use nonlinear kernels such as the Gaussian kernel, that we define as:

$$k(x, x') = \exp\Big(- \frac{\|x - x'\|^2}{\sigma^2} \Big)$$

The dual SVM consists of solving the following quadratic program:

$$\max_{\alpha} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{ij} \alpha_i \alpha_j y_i y_j k(x_i, x_j)$$

subject to:

$$0 \leq \alpha_i \leq C \qquad \text{and} \qquad \sum_{i=1}^{N} \alpha_i y_i = 0.$$

Then, given the alphas, the prediction of the SVM can be obtained as:

$$f(x) = \begin{cases} 1 & \text{if} \quad \sum_{i=1}^{N} \alpha_i y_i k(x, x_i) + \theta > 0 \\ -1 & \text{if} \quad \sum_{i=1}^{N} \alpha_i y_i k(x, x_i) + \theta < 0 \end{cases}$$

where

$$\theta = \frac{1}{\#SV} \sum_{i \in SV} \Big(y_i - \sum_{j=1}^{N} \alpha_j y_j k(x_i, x_j) \Big)$$

and `SV` is the set of indices corresponding to the unbound support vectors.

1.2 Implementation (25 P)

We will solve the dual SVM applied to the MNIST dataset using the CVXOPT quadratic optimizer. For this, we have to build the data structures (vectors and matrices) to must be passed to the optimizer.

- *Implement* a function `gaussianKernel` that returns for a Gaussian kernel of scale σ, the Gram matrix of the two data sets given as argument.
- *Implement* a function `getQPMatrices` that builds the matrices `P`, `q`, `G`, `h`, `A`, `b` (of type `cvxopt.matrix`) that need to be passed as argument to the optimizer `cvxopt.solvers.qp`.
- *Run* the code below using the functions that you just implemented. (It should take less than 3 minutes.)

In [90]:
```
import scipy,scipy.spatial
from cvxopt import matrix

def gaussianKernel(X1,X2,sigma):

    X_dis = scipy.spatial.distance.cdist(X1,X2,'euclidean')
    K = numpy.exp(numpy.power(X_dis,2)/(-sigma**2))

    return K

def getQPMatrices(X,Y,C):
    N = len(X)

    P = numpy.outer(Y,Y)*X
    P = matrix(P)
    q = -numpy.ones(N)
    q = matrix(q)
    G = -numpy.eye(N)
    G = matrix(G)
    h = numpy.zeros(N)
    h = matrix(h)
    A = Y.reshape(1,1000)
    A = matrix(A)
    b = numpy.zeros(1)
    b = matrix(b)

    return P,q,G,h,A,b
```

In [101]:
```
import utils,numpy,cvxopt,cvxopt.solvers
import time

Xtrain,Ttrain,Xtest,Ttest = utils.getMNIST56()

cvxopt.solvers.options['show_progress'] = False

for scale in [10,30,100]:
```

```python
    for C in [1,10,100]:

        t_ini = time.clock()
        # Prepare kernel matrices
        ### TODO: REPLACE BY YOUR OWN CODE
        Ktrain = gaussianKernel(Xtrain,Xtrain,scale)
        Ktest  = gaussianKernel(Xtest,Xtrain,scale)
        ###


        # Prepare the matrices for the quadratic program
        ### TODO: REPLACE BY YOUR OWN CODE
        P,q,G,h,A,b = getQPMatrices(Ktrain,Ttrain,C)
        ###


        # Train the model (i.e. compute the alphas)
        alpha = numpy.array(cvxopt.solvers.qp(P,q,G,h,A,b)['x']).flatten()

        # Get predictions for the training and test set
        SV = (alpha>1e-6)
        uSV = SV*(alpha<C-1e-6)
        theta = 1.0/sum(uSV)*(Ttrain[uSV]-numpy.dot(Ktrain[uSV,:],alpha*Ttrain)).sum()
        Ytrain = numpy.sign(numpy.dot(Ktrain[:,SV],alpha[SV]*Ttrain[SV])+theta)
        Ytest  = numpy.sign(numpy.dot(Ktest [:,SV],alpha[SV]*Ttrain[SV])+theta)

        # Print accuracy and number of support vectors
        Atrain = (Ytrain==Ttrain).mean()
        Atest  = (Ytest ==Ttest ).mean()

        t_end = time.clock()
        t = t_end - t_ini
        print('Scale=%3d  C=%3d  SV: %4d  Train: %.3f  Test: %.3f   Time:
        %.3f' %(scale,C,sum(SV),Atrain,Atest,t))
    print('')
```

```
Scale= 10  C=  1  SV: 1000  Train: 1.000  Test: 0.937   Time:2.561
Scale= 10  C= 10  SV: 1000  Train: 1.000  Test: 0.937   Time:2.547
Scale= 10  C=100  SV: 1000  Train: 1.000  Test: 0.937   Time:2.540

Scale= 30  C=  1  SV:  256  Train: 1.000  Test: 0.986   Time:2.642
Scale= 30  C= 10  SV:  256  Train: 1.000  Test: 0.986   Time:2.676
Scale= 30  C=100  SV:  256  Train: 1.000  Test: 0.986   Time:2.665

Scale=100  C=  1  SV:  134  Train: 1.000  Test: 0.975   Time:2.716
Scale=100  C= 10  SV:  134  Train: 1.000  Test: 0.975   Time:2.689
Scale=100  C=100  SV:  134  Train: 1.000  Test: 0.975   Time:2.827
```

1.3 Analysis (10 P)

- *Explain* which combinations of parameters σ and C lead to good generalization, underfitting or overfitting?

Answer : As we can see from the results above, that the combination of $\sigma = 30$ and $C = 1, 10, 100$ lead to good generalization with underfitting.

- *Explain* which combinations of parameters σ and C produce the fastest classifiers (in terms of amount of computation needed at prediction time)?

Answer : We can simply print the time spent after each iteration. The combination of parameters $\sigma = 10$ and $C = 100$ produce the fastest classifiers.

1.4 Part B: Linear SVMs and Gradient Descent in the Primal

The quadratic problem of the dual SVM does not scale well with the number of data points. For large number of data points, it is generally more appropriate to optimize the SVM in the primal. The primal optimization problem for linear SVMs can be written as

$$\min_{w,\theta} ||w||^2 + C \sum_{i=1}^{N} \xi_i \qquad \text{where} \qquad \forall_{i=1}^{N} : y_i(w \cdot x_i + \theta) \geq 1 - \xi_i \qquad \text{and} \qquad \xi_i \geq 0.$$

It is common to incorporate the constraints directly into the objective and then minimizing the unconstrained objective

$$J(w, \theta) = ||w||^2 + C \sum_{i=1}^{N} \max(0, 1 - y_i(w \cdot x_i + \theta))$$

using simple gradient descent.

1.5 Implementation (15 P)

- *Implement* the function `J` computing the objective $J(w, \theta)$
- *Implement* the function `DJ` computing the gradient of the objective $J(w, \theta)$ with respect to the parameters w and θ.
- *Run* the code below using the functions that you just implemented. (It should take less than 1 minute.)

```
In [85]: def J(w,theta,C,X,Y):

             N = X.shape[0]
             D = X.shape[1]

             w_n = numpy.linalg.norm(w)
             C_c = numpy.ones([N]) - Y*(numpy.dot(X,w)-theta*numpy.ones([N]))
             C_c = C_c.reshape(N,1)
             m = numpy.amax(numpy.concatenate((C_c,numpy.zeros([N,1])),axis = 1),axis = 1)
             J = w_n**2 + C*m.sum()
             return J
```

```
def DJ(w,theta,C,X,Y):
    N,D = X.shape

    indicator = ((numpy.ones(N) - Y*(numpy.dot(X,w)+theta*numpy.ones(N)))>numpy.zeros(N

    dw = 2*w - C*(((indicator*Y)[:,numpy.newaxis]*X).sum(axis=0))
    dtheta = -C*(indicator*Y).sum(axis = 0)

    return dw,dtheta
```

In [77]:
```
def DJ(w,theta,C,X,Y):

    N = X.shape[0]
    D = X.shape[1]

    C_c = numpy.ones([N]) - Y*(numpy.dot(X,w)-theta*numpy.ones([N]))
    C_c = C_c.reshape(N,1)
    index = numpy.where(C_c > 0.0)
    m = -Y[:,numpy.newaxis]*X
    m = m[index]
    t = -Y*theta
    t = t.reshape(N,1)
    t = t[index]

    dw = 2*w + C*m.sum(axis = 0)
    dtheta = C*t.sum(axis = 0)

    return dw,dtheta
```

In [86]:
```
import utils,numpy

C = 10.0
lr = 0.001

Xtrain,Ttrain,Xtest,Ttest = utils.getMNIST56()

n,d = Xtrain.shape

w = numpy.zeros([d])
theta = 1e-9

for it in range(0,101):

    # Monitor the training and test error every 5 iterations
    if it%5==0:
        Ytrain = numpy.sign(numpy.dot(Xtrain,w)+theta)
```

```
            Ytest  = numpy.sign(numpy.dot(Xtest ,w)+theta)

            ### TODO: REPLACE BY YOUR OWN CODE
            Obj    = J(w,theta,C,Xtrain,Ttrain)
            ###

            Etrain = (Ytrain==Ttrain).mean()
            Etest  = (Ytest ==Ttest ).mean()
            print('It=%3d   J: %9.3f  Train: %.3f  Test: %.3f'%(it,Obj,Etrain,Etest))

        ### TODO: REPLACE BY YOUR OWN CODE
        dw,dtheta = DJ(w,theta,C,Xtrain,Ttrain)
        ###

        w = w - lr*dw
        theta = theta - lr*dtheta
```

```
It=  0   J: 10000.000  Train: 0.471  Test: 0.482
It=  5   J: 68686.731  Train: 0.961  Test: 0.958
It= 10   J: 50000.274  Train: 0.973  Test: 0.961
It= 15   J: 37457.648  Train: 0.973  Test: 0.963
It= 20   J: 28652.040  Train: 0.974  Test: 0.965
It= 25   J: 21727.090  Train: 0.977  Test: 0.967
It= 30   J: 16913.518  Train: 0.980  Test: 0.968
It= 35   J: 13590.484  Train: 0.986  Test: 0.967
It= 40   J: 11119.869  Train: 0.986  Test: 0.967
It= 45   J:  9172.561  Train: 0.991  Test: 0.967
It= 50   J:  7652.186  Train: 0.990  Test: 0.968
It= 55   J:  6418.409  Train: 0.988  Test: 0.966
It= 60   J:  5248.271  Train: 0.995  Test: 0.966
It= 65   J:  4523.320  Train: 0.992  Test: 0.967
It= 70   J:  4033.051  Train: 0.996  Test: 0.966
It= 75   J:  3677.583  Train: 0.997  Test: 0.965
It= 80   J:  3526.082  Train: 0.998  Test: 0.966
It= 85   J:  3404.280  Train: 1.000  Test: 0.966
It= 90   J:  3336.804  Train: 1.000  Test: 0.966
It= 95   J:  3270.665  Train: 1.000  Test: 0.966
It=100   J:  3205.837  Train: 1.000  Test: 0.966
```