# sheet09_prog

December 16, 2018

## 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:

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

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

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

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

### 1.1.1 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 $\sigma$, 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 [1]:
```
import utils,numpy,cvxopt,cvxopt.solvers,scipy.spatial
Xtrain,Ttrain,Xtest,Ttest = utils.getMNIST56()
cvxopt.solvers.options['show_progress'] = False
```

In [2]:
```
#print("Xtrain,Ttrain,Xtest,Ttest", Xtrain.shape,Ttrain.shape,Xtest.shape,Ttest.shape)
def gaussianKernel(X1, X2, sigma):
    return numpy.exp(-scipy.spatial.distance.cdist(X1, X2, 'euclidean')**2/sigma**2)

def getQPMatrices(K, T, C):
    ones = numpy.ones(K.shape[0])
    zeros = ones*.0

    # maximizer: therefore negative minimizer --> max qx - xPx = min -qx + xPx
    Y = numpy.outer(T, T)
    P = Y*K
    q = -ones

    # constraint: 0  a_i  C --> Gx  h
    G = numpy.concatenate([numpy.diag(ones), -numpy.diag(ones)])
    h = numpy.concatenate([C*ones, zeros])

    # constraint: sum a_i*y_i = 0 --> Ax = b
    A = T.reshape(1,-1)
    b = .0

    # numpy matrices needs to be converted before accepted by cvxopt
    return map(cvxopt.matrix, (P, q, G, h, A, b))

for scale in [10,30,100]:
    for C in [1,10,100]:
        # Prepare kernel matrices
        Ktrain = gaussianKernel(Xtrain, Xtrain, scale)
        Ktest  = gaussianKernel(Xtest, Xtrain, scale)


        # Prepare the matrices for the quadratic program
```

```
            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()
            print('Scale=%3d  C=%3d  SV: %4d  Train: %.3f  Test: %.3f'%(scale,C,sum(SV),Atra
        print('')
```

```
Scale= 10  C=  1  SV: 1000  Train: 1.000  Test: 0.937
Scale= 10  C= 10  SV: 1000  Train: 1.000  Test: 0.937
Scale= 10  C=100  SV: 1000  Train: 1.000  Test: 0.937

Scale= 30  C=  1  SV:  254  Train: 1.000  Test: 0.985
Scale= 30  C= 10  SV:  274  Train: 1.000  Test: 0.986
Scale= 30  C=100  SV:  256  Train: 1.000  Test: 0.986

Scale=100  C=  1  SV:  317  Train: 0.973  Test: 0.971
Scale=100  C= 10  SV:  159  Train: 0.990  Test: 0.975
Scale=100  C=100  SV:  136  Train: 1.000  Test: 0.975
```

### 1.1.2 Analysis (10 P)

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

Looking at the results, the best generalisation seems to come from scale=30 and c=10 or c=100, where we got the highest test score (although it might be slightly overfitted). Scale=10 seems to lead to some overfitting, seing that the test scores are significantly lower than the training score. And Scale=100 seems to be a case of underfitting, where both test and training scores are smaller than 1 in general. Judging by the number of support vectors used, this seems to further confirm this idea.

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

Generally speaking, a lower amount of support vectors should lead to a faster computation, which means that sigma and c, should be rather large.

### 1.2 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.

#### 1.2.1 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 $\theta$.
- *Run* the code below using the functions that you just implemented. (It should take less than 1 minute.)

In [4]:
```
import utils,numpy

C = 10.0
lr = 0.001

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

n,d = Xtrain.shape

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

def J(w, theta, C, Xtrain, Ttrain):
    comp = 1.0-Ttrain*(w.dot(Xtrain.T))+theta

    return numpy.linalg.norm(w)**2 + C*numpy.maximum(.0, comp).sum()

def DJ(w,theta,C,Xtrain,Ttrain):
    comp = 1.0-Ttrain*(w.dot(Xtrain.T))+theta
    select = (numpy.maximum(0,comp)/comp)
    dw = 2*w+C*-numpy.dot(Ttrain*select, Xtrain)
    dtheta = C*(-Ttrain*select).sum()

    return dw, dtheta
```

```python
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)
    ###
    #print(dw.shape)
    w = w - lr*dw
    theta = theta - lr*dtheta
```

```
It=  0   J: 552154.345  Train: 0.443  Test: 0.434
It=  5   J: 32779.497  Train: 0.971  Test: 0.961
It= 10   J: 21448.473  Train: 0.972  Test: 0.968
It= 15   J: 14459.996  Train: 0.975  Test: 0.968
It= 20   J:  9907.475  Train: 0.980  Test: 0.972
It= 25   J:  6781.030  Train: 0.986  Test: 0.972
It= 30   J:  4920.204  Train: 0.986  Test: 0.971
It= 35   J:  3523.371  Train: 0.992  Test: 0.970
It= 40   J:  2779.522  Train: 0.996  Test: 0.969
It= 45   J:  2445.678  Train: 0.998  Test: 0.970
It= 50   J:  2112.371  Train: 0.998  Test: 0.970
It= 55   J:  1992.196  Train: 0.998  Test: 0.970
It= 60   J:  1960.825  Train: 0.997  Test: 0.970
It= 65   J:  1770.648  Train: 1.000  Test: 0.970
It= 70   J:  1735.552  Train: 1.000  Test: 0.970
It= 75   J:  1701.152  Train: 1.000  Test: 0.970
It= 80   J:  1667.434  Train: 1.000  Test: 0.970
It= 85   J:  1634.384  Train: 1.000  Test: 0.970
It= 90   J:  1601.988  Train: 1.000  Test: 0.970
It= 95   J:  1570.236  Train: 1.000  Test: 0.970
It=100   J:  1539.112  Train: 1.000  Test: 0.970
```