sheet09

December 16, 2017

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:

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

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

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

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 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 [3]: import utils,numpy,cvxopt,cvxopt.solvers
        import scipy,scipy.spatial

        na = numpy.newaxis

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

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

        def gaussianKernel(X1,X2,scale):
            D = scipy.spatial.distance.cdist(X1,X2,'sqeuclidean')
            K = numpy.exp(-D/scale**2)
            return K

        def getQPMatrices(K,Y,C):
            n = Y.shape[0]

            # Prepare matrices (regarding exercise 2 of theory part)
            P = 0.5*Y[:,na]*K*Y[na,:]
            q = -numpy.ones([n])
            G = numpy.concatenate([numpy.identity(n), -numpy.identity(n)])
            h = numpy.concatenate([C*numpy.ones([n]), numpy.zeros([n])])
            A = Y
            b = numpy.array([0.0])

            # Convert to CVXOPT matrices
            P = cvxopt.matrix(P)
            q = cvxopt.matrix(q)
            G = cvxopt.matrix(G)
            h = cvxopt.matrix(h)
            A = cvxopt.matrix(A, (1,n))
            b = cvxopt.matrix(b)

            return 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) # Count support vectors
        uSV = SV*(alpha<C-1e-6)
        B = 1.0/sum(uSV)*(Ttrain[uSV]-numpy.dot(Ktrain[uSV,:],alpha*Ttrain)
        Ytrain = numpy.sign(numpy.dot(Ktrain[:,SV],alpha[SV]*Ttrain[SV])+B)
        Ytest  = numpy.sign(numpy.dot(Ktest [:,SV],alpha[SV]*Ttrain[SV])+B)

        # 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
    print('')
```

```
Scale= 10  C=  1  SV: 1000  Train: 1.000  Test: 0.754
Scale= 10  C= 10  SV: 1000  Train: 1.000  Test: 0.853
Scale= 10  C=100  SV: 1000  Train: 1.000  Test: 0.853

Scale= 30  C=  1  SV:  321  Train: 0.992  Test: 0.985
Scale= 30  C= 10  SV:  281  Train: 1.000  Test: 0.986
Scale= 30  C=100  SV:  256  Train: 1.000  Test: 0.986

Scale=100  C=  1  SV:  433  Train: 0.973  Test: 0.968
Scale=100  C= 10  SV:  311  Train: 0.984  Test: 0.971
Scale=100  C=100  SV:  163  Train: 1.000  Test: 0.975
```

1.1.2 Analysis (10 P)

- *Explain* which combinations of parameters σ and C lead to good generalization, underfitting or overfitting?
- *Explain* which combinations of parameters σ and C produce the fastest classifiers (in terms of amount of computation needed at prediction time)?

σ controls the smoothing that is done by the kernel:

- A low σ leads to a higher number of support vectors and increases the overfit.
- A high σ increases the calculation time.

$\Rightarrow$ There is a tradeoff between good generalization and computation time.
C is the magnitude of slack that is allowed for the soft margin. A higher slack leads to more support vectors.

- If the amount of support vectors is too low, underfitting might occur. If the amount of support vectors is too high, overfitting might occur.
- The computation time increases with the amount of support vectors.

$\Rightarrow$ *C* needs to be chosen in a way such that the number of support vectors leads to a good generalization and such that the calculation does not take an overbearing amount of time.

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 θ.
- *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.zeros([d])
    b = 1e-9


    def J(w,b,C,x,y):
        z = numpy.dot(x,w)+b
        return (w**2).sum()+C*numpy.maximum(0,1-y*z).sum()


    def DJ(w,b,C,x,y):
        z = numpy.dot(x,w)+b
        dw = 2*w+C*(((1-y*z)>0)[:,na]*x*y[:,na]).sum(axis=0)
        db = -C*(((1-y*z)>0)*y).sum(axis=0)
        return dw,db


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

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

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

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

        w = w - lr*dw
        b = b - lr*db
```

```
It=   0   J: 10000.000  Train: 0.471  Test: 0.482
It=   5   J: 38160181.082  Train: 0.064  Test: 0.061
It=  10   J: 76463688.162  Train: 0.068  Test: 0.061
It=  15   J: 114755927.488  Train: 0.068  Test: 0.061
It=  20   J: 153026193.442  Train: 0.068  Test: 0.062
It=  25   J: 191267580.768  Train: 0.068  Test: 0.062
It=  30   J: 229473394.193  Train: 0.068  Test: 0.062
It=  35   J: 267637143.538  Train: 0.068  Test: 0.062
It=  40   J: 305753079.102  Train: 0.069  Test: 0.062
It=  45   J: 343814909.651  Train: 0.069  Test: 0.062
```

```
It= 50   J: 381816388.535  Train: 0.069  Test: 0.062
It= 55   J: 419751798.934  Train: 0.069  Test: 0.062
It= 60   J: 457615702.694  Train: 0.070  Test: 0.062
It= 65   J: 495403014.324  Train: 0.070  Test: 0.062
It= 70   J: 533108182.775  Train: 0.070  Test: 0.062
It= 75   J: 570726194.392  Train: 0.070  Test: 0.062
It= 80   J: 608252201.047  Train: 0.070  Test: 0.062
It= 85   J: 645681516.203  Train: 0.070  Test: 0.062
It= 90   J: 683009611.055  Train: 0.070  Test: 0.062
It= 95   J: 720232110.759  Train: 0.070  Test: 0.062
It=100   J: 757344790.742  Train: 0.070  Test: 0.062
```