sheet09_GBBSPL

December 18, 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

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

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 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 [78]:
```
import utils,numpy,cvxopt,cvxopt.solvers
# import solutions

def gaussianKernel(X, X_prime, scale):
    s1 = X.shape[0]
    s2 = X_prime.shape[0]
    nFeatures1 = X.shape[1]
    nFeatures2 = X_prime.shape[1]
    assert(nFeatures1 == nFeatures2)
    K = numpy.zeros((s1,s2))
    for i in range(s1):
        for j in range(s2):
            tmp = X[i] - X_prime[j]
            K[i, j] = numpy.exp(-numpy.linalg.norm(tmp)**2 / (scale ** 2))
    return K

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

    P = cvxopt.matrix(numpy.outer(y,y) * K)
    q = cvxopt.matrix(-numpy.ones((n, 1)))
    A = cvxopt.matrix(y, (1,n))
    b = cvxopt.matrix(0.0)

    tmp1 = -numpy.identity(n)
    tmp2 = numpy.identity(n)
    G = cvxopt.matrix(numpy.vstack((tmp1, tmp2)))
    tmp1 = numpy.zeros(n)
    tmp2 = numpy.ones(n) * C
    h = cvxopt.matrix(numpy.hstack((tmp1, tmp2)))

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


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

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

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),Atr
    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 σ and C lead to good generalization, underfitting or overfitting?

While σ is small(10), we could see overfitting, since training error is always 0 while the test accuracy is not so good. On the otherside, if σ is too big(100), it will lead to underfitting compared to the other 2 groups, as we could see from the 6th to 9th output. When σ is in the middle(30), we got the best result.

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

When doing prediction, the less support vectors we have, the less computation steps needed. From the outputs, we could see that the last item with $\sigma = 100$ and $C = 100$ got the least number of support vectors.

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 [71]: def J(w, theta, C, X, Y):
             tmp = -Y*(X.dot(w) + theta) + 1
             tmp[tmp<0] = 0
             return numpy.linalg.norm(w)**2 + C * numpy.sum(tmp)

         def DJ(w, theta, C, X, Y):
             tmp1 = -Y*(X.dot(w) + theta) + 1
             Y = Y.reshape(1000, 1)
```

```
        tmp2 = X*Y
        tmp2[tmp1 < 0] = 0
        dw = 2*w + C * numpy.sum(-tmp2, axis=0)
        tmp3 = -Y
        tmp3[tmp1 < 0] = 0
        dtheta = numpy.sum(tmp3)
        return dw, dtheta

    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)

            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))

        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: 68694.876  Train: 0.961  Test: 0.958
It= 10   J: 50048.187  Train: 0.973  Test: 0.961
It= 15   J: 37609.110  Train: 0.973  Test: 0.964
It= 20   J: 28556.347  Train: 0.973  Test: 0.966
It= 25   J: 21725.132  Train: 0.979  Test: 0.967
It= 30   J: 16899.386  Train: 0.984  Test: 0.968
It= 35   J: 13693.666  Train: 0.985  Test: 0.967
It= 40   J: 11179.679  Train: 0.985  Test: 0.967
It= 45   J:  9259.750  Train: 0.989  Test: 0.967
It= 50   J:  7720.477  Train: 0.990  Test: 0.968
```

```
It= 55    J:  6396.402  Train: 0.993  Test: 0.966
It= 60    J:  5237.610  Train: 0.996  Test: 0.966
It= 65    J:  4550.211  Train: 0.990  Test: 0.966
It= 70    J:  4017.758  Train: 0.997  Test: 0.966
It= 75    J:  3675.828  Train: 0.997  Test: 0.965
It= 80    J:  3494.036  Train: 0.999  Test: 0.966
It= 85    J:  3396.959  Train: 1.000  Test: 0.966
It= 90    J:  3329.628  Train: 1.000  Test: 0.966
It= 95    J:  3263.632  Train: 1.000  Test: 0.966
It=100    J:  3198.943  Train: 1.000  Test: 0.966
```

In [ ]: