Kernel Support Vector Machines

In this exercise sheet, we will implement a kernel SVM. Our implementation will be based on a generic quadratic programming optimizer provided in CVXOPT (`python-cvxopt` package, or directly from the website `www.cvxopt.org`). The SVM will then be tested on the UCI breast cancer dataset, a simple binary classification dataset accessible via the `scikit-learn` library.

1. Building the Gaussian Kernel (5 P)

As a starting point, we would like to implement the Gaussian kernel, which we will make use of in our kernel SVM implementation. It is defined as:

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

- **Implement a function `getGaussianKernel` that returns for a Gaussian kernel of scale σ, the Gram matrix of the two data sets given as argument.**

```
In [1]: import numpy,scipy,scipy.spatial

        def getGaussianKernel(X1,X2,scale):
            ### TODO: REPLACE BY YOUR OWN CODE
            import solutions
            K = solutions.getGaussianKernel(X1,X2,scale)
            return K
            ###
```

2. Building the Matrices for the CVXOPT Quadratic Solver (20 P)

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. The dual SVM consists of solving the following quadratic program:

$$\max_{\alpha} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{ij} \alpha_i \alpha_j y_i y_j k(x_i, x_j) \qquad \text{subject to:} \qquad 0 \leq \alpha_i \leq C \qquad \text{and} \qquad \sum_{i=1}^{N} \alpha_i y_i = 0.$$

We would like to rely on a CVXOPT solver to obtain a solution to our SVM dual. The function `cvxopt.solvers.qp` solves an optimization problem of the type:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \frac{1}{2}\boldsymbol{x}^\top P \boldsymbol{x} + \boldsymbol{q}^\top \boldsymbol{x} \\ \text{subject to} \quad & G\boldsymbol{x} \preceq \boldsymbol{h} \\ \text{and} \quad & A\boldsymbol{x} = \boldsymbol{b}. \end{aligned}$$

which is of similar form to our dual SVM (note that $\boldsymbol{x}$ will correspond to the parameters $(\alpha_i)_i$ of the SVM). We need to build the data structures (vectors and matrices) that makes solving this quadratic problem equivalent to solving our dual SVM.

- **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`.**

```
In [2]: import cvxopt,cvxopt.solvers
        cvxopt.solvers.options['show_progress'] = False

        def getQPMatrices(K,T,C):
            ### TODO: REPLACE BY YOUR CODE
            import solutions
            P,q,G,h,A,b = solutions.getQPMatrices(K,T,C)
            return P,q,G,h,A,b
            ###
```

3. Computing the Bias Parameters (10 P)

Given the parameters $(\alpha_i)_i$ the optimization procedure has found, the prediction of the SVM is given by:

$$f(x) = \text{sign}\Big(\sum_{i=1}^{N} \alpha_i y_i k(x, x_i) + \theta\Big)$$

Note that the parameter θ has not been computed yet. It can be obtained from any support vector that lies exactly on the margin, or equivalently, whose associated parameter α is not equal to 0 or C. Calling one such vector "x_M", the parameter θ can be computed as:

$$\theta = y_M - \sum_{j=1}^{N} \alpha_j y_j k(x_M, x_j)$$

- **Implement a function `getTheta` that takes as input the Gram Matrix used for training, the label vector, the solution of our quadratic program, and the hyperparameter C. The function should return the parameter θ.**

```
In [3]: def getTheta(K,T,alpha,C):
            ### TODO: REPLACE BY YOUR CODE
            import solutions
            theta = solutions.getTheta(K,T,alpha,C)
            return theta
            ###
```

4. Implementing a class `GaussianSVM` (15 P)

All functions that are needed to learn the SVM have now been built. We would like to implement a SVM class that connects them and make the SVM easily usable. The class structure is given below and contains two functions, one for training the model, and one for applying it to test data.

- **Implement the function `fit` that makes use of the functions `getGaussianKernel`, `getQPMatrices`, `getTheta` you have already implemented. The function should learn the SVM model and store the support vectors, their label, $(\alpha_i)_i$ and θ into the object (`self`).**
- **Implement the function `predict` that makes use of the stored information to compute the SVM output for any new collection of data points**

```
In [4]: class GaussianSVM:

            def __init__(self,C=1.0,scale=1.0):

                self.C, self.scale = C, scale

            def fit(self,X,T):
```

```
        ### TODO: REPLACE BY YOUR CODE
        import solutions
        solutions.fit(self,X,T)
        ###

    def predict(self,X):

        ### TODO: REPLACE BY YOUR CODE
        import solutions
        Y = solutions.predict(self,X)
        return Y
        ###
```

5. Analysis

The following code tests the SVM on some breast cancer binary classification dataset for a range of scale and soft-margin parameters. For each combination of parameters, we output the number of support vectors as well as the train and test accuracy averaged over a number of random train/test splits. Running the code below should take approximately 1-2 minutes.

```
In [5]: import numpy,sklearn,sklearn.datasets,numpy

        D = sklearn.datasets.load_breast_cancer()
        X = D['data']
        T = D['target']
        T = (D['target']==1)*2.0-1.0

        for scale in [30,100,300,1000,3000]:
            for C in [10,100,1000,10000]:

                acctrain,acctest,nbsvs = [],[],[]

                svm = GaussianSVM(C=C,scale=scale)

                for i in range(10):

                    # Split the data
                    R = numpy.random.mtrand.RandomState(i).permutation(len(X))
                    Xtrain,Xtest = X[R[:len(R)//2]]*1,X[R[len(R)//2:]]*1
                    Ttrain,Ttest = T[R[:len(R)//2]]*1,T[R[len(R)//2:]]*1

                    # Train and test the SVM
                    svm.fit(Xtrain,Ttrain)
                    acctrain += [(svm.predict(Xtrain)==Ttrain).mean()]
                    acctest  += [(svm.predict(Xtest)==Ttest).mean()]
                    nbsvs += [len(svm.X)*1.0]

                print('scale=%9.1f  C=%9.1f  nSV: %4d  train: %.3f  test: %.3f'%(
                    scale,C,numpy.mean(nbsvs),numpy.mean(acctrain),numpy.mean(acctest)))
            print('')
```

```
scale=     30.0  C=     10.0  nSV:  183  train: 0.997  test: 0.921
scale=     30.0  C=    100.0  nSV:  178  train: 1.000  test: 0.918
scale=     30.0  C=   1000.0  nSV:  184  train: 1.000  test: 0.918
```

```
scale=     30.0  C=  10000.0  nSV:  182  train: 1.000  test: 0.918

scale=    100.0  C=     10.0  nSV:  117  train: 0.965  test: 0.935
scale=    100.0  C=    100.0  nSV:   97  train: 0.987  test: 0.940
scale=    100.0  C=   1000.0  nSV:   85  train: 0.998  test: 0.932
scale=    100.0  C=  10000.0  nSV:   71  train: 1.000  test: 0.926

scale=    300.0  C=     10.0  nSV:   88  train: 0.939  test: 0.924
scale=    300.0  C=    100.0  nSV:   48  train: 0.963  test: 0.943
scale=    300.0  C=   1000.0  nSV:   36  train: 0.978  test: 0.946
scale=    300.0  C=  10000.0  nSV:   32  train: 0.991  test: 0.941

scale=   1000.0  C=     10.0  nSV:   66  train: 0.926  test: 0.916
scale=   1000.0  C=    100.0  nSV:   55  train: 0.935  test: 0.929
scale=   1000.0  C=   1000.0  nSV:   49  train: 0.956  test: 0.946
scale=   1000.0  C=  10000.0  nSV:   38  train: 0.971  test: 0.951

scale=   3000.0  C=     10.0  nSV:   87  train: 0.912  test: 0.903
scale=   3000.0  C=    100.0  nSV:   68  train: 0.926  test: 0.919
scale=   3000.0  C=   1000.0  nSV:   58  train: 0.934  test: 0.929
scale=   3000.0  C=  10000.0  nSV:   49  train: 0.953  test: 0.943
```

We observe that the highest accuracy is obtained with a scale parameter that is neither too small nor too large. Best parameters are also often associated to a low number of support vectors.