sheet08-programming-group49

December 11, 2025

1 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.

1.1 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.**

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

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

1.2 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:

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

which is of similar form to our dual SVM (note that 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`.**

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

def getQPMatrices(K,T,C):
    N = K.shape[0]    # number of training samples
    # ---- P matrix ----
    # P_ij = y_i * y_j * K_ij
    # CVXOPT requires P to be positive semidefinite
    P = cvxopt.matrix(np.outer(T, T) * K, tc='d')

    # ---- q vector ----
    # q_i = -1  (because the dual maximizes sum(a_i), so in minimization:␣
 ↪-sum(a_i))
    q = cvxopt.matrix(-np.ones(N), tc='d')

    # ---- G matrix and h vector ----
    # Box constraints: 0   a_i   C
    # 1) a_i >= 0    ->    -a_i <= 0
    # 2) a_i <= C    ->     a_i <= C
    #
    # SG:
    #    [-I]    (for a >=0)
    #    [ I]    (for a <=C)
    #
    # h:
    #    [0]
    #    [C]

    G = cvxopt.matrix(
        np.vstack([-np.eye(N), np.eye(N)]),
        tc='d'
    )
    h = cvxopt.matrix(
        np.hstack([np.zeros(N), C * np.ones(N)]),
        tc='d'
    )

    # ---- A matrix and b vector ----
    # Equality constraint: sum_i a_i y_i = 0
    # Single row matrix A = [y_1, ..., y_N]
    A = cvxopt.matrix(T.reshape(1, -1), tc='d')
    b = cvxopt.matrix(0.0, tc='d')

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

1.3 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:

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

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 hyper-parameter C. The function should return the parameter θ.**

[ ]:
```
import numpy as np

def getTheta(K, y, a, C):
    """
        theta = y_M - Σ_j (a_j * y_j * K_{M,j})
    """

    # flatten vectors
    a = np.ravel(a)        # a_j (dual variables)
    y = np.ravel(y)        # y_j (labels)

    # margin support vectors: 0 < _i < C
    margin_mask = (a > 1e-6) & (a < C - 1e-6)

    if not np.any(margin_mask):
        raise ValueError("No margin support vector found to compute  ")
    # pick one margin SV index M
    M = np.where(margin_mask)[0][0]

    # y_M = y[M]
    # a_j = a[j]
    # y_j = y[j]
    # K_{M,j} = K[M, j]

    y_M = y[M]
    K_Mj = K[M]                 # row M → K_{M,j} over all j
    sum_term = np.sum(a * y * K_Mj)   # Σ_j(a_j y_j K_{M,j})

    theta = y_M - sum_term

    return theta
```

1.4 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**

[ ]:
```
# import numpy as np
# import cvxopt
class GaussianSVM:

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

    def fit(self, X, T):
        """
        Train the Gaussian kernel SVM:
            1. Compute Gram matrix K
            2. Build QP matrices
            3. Solve dual with cvxopt.solvers.qp
            4. Store support vectors and parameters
            5. Compute   (bias)
         """

        # Store training data
        self.X = X
        self.T = np.ravel(T)

        # 1) Compute Gram matrix using Gaussian kernel
        K = getGaussianKernel(X, X, self.scale)

        # 2) Build QP matrices
        P, q, G, h, A, b = getQPMatrices(K, self.T, self.C)

        # 3) Solve the QP (dual SVM)
        sol = cvxopt.solvers.qp(P, q, G, h, A, b)
        alpha = np.ravel(sol['x'])

        # 4) Identify support vectors (a > 0)
        sv_mask = alpha > 1e-6
        self.sv_alpha = alpha[sv_mask]
        self.sv_X = X[sv_mask]
        self.sv_T = self.T[sv_mask]

        # 5) Compute   using full a, K
        self.theta = getTheta(K, self.T, alpha, self.C)

        # Store   for debugging
        self.alpha = alpha

    def predict(self, X):
        """
            f(x) = sign(Σ a_i y_i k(x, x_i) +  )
        """

        # Compute kernel between test points and support vectors
        K_test = getGaussianKernel(X, self.sv_X, self.scale)

        # Decision function
        decision = np.dot(K_test, self.sv_alpha * self.sv_T) + self.theta

        # Return predictions: sign(...)
        return np.sign(decision)
```

1.5 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.

```
[ ]: 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:  284  train: 0.997  test: 0.921
scale=     30.0  C=    100.0  nSV:  284  train: 1.000  test: 0.917
scale=     30.0  C=   1000.0  nSV:  284  train: 1.000  test: 0.917
scale=     30.0  C=  10000.0  nSV:  284  train: 1.000  test: 0.918

scale=    100.0  C=     10.0  nSV:  284  train: 0.966  test: 0.934
scale=    100.0  C=    100.0  nSV:  284  train: 0.987  test: 0.938
scale=    100.0  C=   1000.0  nSV:  284  train: 0.997  test: 0.929
scale=    100.0  C=  10000.0  nSV:  284  train: 0.965  test: 0.894

scale=    300.0  C=     10.0  nSV:  284  train: 0.937  test: 0.925
scale=    300.0  C=    100.0  nSV:  284  train: 0.961  test: 0.942
scale=    300.0  C=   1000.0  nSV:  284  train: 0.944  test: 0.919
scale=    300.0  C=  10000.0  nSV:  284  train: 0.891  test: 0.855

scale=   1000.0  C=     10.0  nSV:  284  train: 0.930  test: 0.915
scale=   1000.0  C=    100.0  nSV:  284  train: 0.925  test: 0.913
```

```
scale=   1000.0  C=   1000.0  nSV:  284  train: 0.895  test: 0.888
scale=   1000.0  C=  10000.0  nSV:  284  train: 0.852  test: 0.840

scale=   3000.0  C=     10.0  nSV:  284  train: 0.912  test: 0.895
scale=   3000.0  C=    100.0  nSV:  284  train: 0.910  test: 0.909
scale=   3000.0  C=   1000.0  nSV:  284  train: 0.885  test: 0.874
scale=   3000.0  C=  10000.0  nSV:  284  train: 0.867  test: 0.854
```

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.