(1a)

In order to state the conditions on the data under which a solution can be found from the Lagrange dual formulation, the Slater's conditions are verified in the following. Because the objective and the inequality constraints are convex in SVM, we search for a point which satisfies these constraints with strict inequalities.

$$\forall i \in 1, .., N : y_i(w^T \Phi(x_i) + b) \geq 1 - \epsilon_i$$
$$\forall i \in 1, .., N : -y_i(w^T \Phi(x_i) + b) + 1 - \epsilon_i \leq 0$$
$$-\epsilon_i \leq 0$$

Setting $w = 0$ and $b = 0$ for all $\epsilon_i > 1$ the inequality constraints are strictly satisfied. For example $\epsilon_i = 2$:

$$-y_i(0 * \Phi(x_i) + 0) - 1 = -1 < 0$$

$$-2 < 0$$

(1b)

Deriving the inner minimization problem of:

$$\max_{\alpha}\{\min_{\omega,b,\zeta}\{\frac{1}{2}||\omega||^2 + C * \sum_{i=1}^{N} \zeta_i + \sum_{i=1}^{N} \alpha_i(1 - \zeta_i - y_i(w^T \Phi(x) + b)) - \sum_{i=1}^{N} u_i \zeta_i\}\}$$

gives us three different equations:

$$\frac{\partial L}{\partial \omega} = \omega - \sum_{i=1}^{N} \alpha_i y_i \phi(x_i) \implies \omega = \sum_{i=1}^{N} \alpha_i y_i \phi(x_i)$$

$$\frac{\partial L}{\partial b} \implies \sum_{i=1}^{N} \alpha_i y_i = 0$$

$$\frac{\partial L}{\partial \zeta} \implies \alpha_i = C + u_i$$

Injecting into L gives us the following quadratic optimization problem:

$$L(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} y_i y_j \alpha_i \alpha_j \langle \phi(x_i \phi(x_j) \rangle$$

$$\begin{gathered} \max L(\alpha) s.t. \\ \alpha_i \geq 0 \\ \alpha_i \leq C \\ \sum_{i=1}^{N} \alpha_i y_i = 0 \end{gathered}$$

(1c)

For $\omega = \sum_{i=1}^{N} \alpha_i y_i \phi(x_i)$ we could get α_i from dual optimization problem. Here we'll state the method to get b.
The KKT-conditions as below:

$$\alpha_i \geq 0 \tag{1}$$
$$\mu_i \geq 0 \tag{2}$$
$$\zeta_i \geq 0 \tag{3}$$
$$y_i f(x_i) - 1 + \zeta_i \geq 0 \tag{4}$$
$$\alpha_i (y_i f(x_i) - 1 + \zeta_i) = 0 \tag{5}$$
$$\mu_i \zeta_i = 0 \tag{6}$$

If $0 < a_i < C$, for which x_i should be support vector, because $a_i = C - u_i$, then u_i must be greater than 0. Because $\zeta_i = 0$ due to (6) and $a_i > 0$, therefore $y_i f(x_i) = 1$ from (5). Using this relationship we have $y_i(\omega^T \phi(x_i) + b) = 1$, so that we conclude the formula to get b is:

$$b = \frac{1}{y_i} - \sum_{m \in S} \alpha_m y_m k(x_i, x_m)$$

where S is the set of all support vectors, and i subscript means any data points make $0 < \alpha_i < C$. Because $y_i \in \{-1, +1\}$ so the equation above could be written as $b = y_i - \sum_{m \in S} \alpha_m y_m k(x_i, x_m)$.

(1d)

(i)

$$\max_{\alpha} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} y_i y_j \alpha_i \alpha_j \langle \Phi(x_i), \Phi(x_j) \rangle$$

$$= \max_{\alpha} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} y_i y_j \alpha_i \alpha_j K(x_i, x_j)$$

(ii)

$$f(x) = sign(w^T \Phi(x) + b)$$

$$= sign(\sum_{i=1}^{N} y_i \alpha_i \Phi(x_i) \Phi(x) + b)$$

$$= sign(\sum_{i=1}^{N} y_i \alpha_i K(x_i, x) + b)$$

Exercise 2: SVMs and Quadratic Programming

(a) Express the matrices and vectors P, q, G, h, A, b

A quadratic optimization problem can be solved by CVXOPT Python software and the matrix form looks like:

$$\begin{aligned} \min_x \frac{1}{2} x^T P x + q^T x \\ s.t. \quad Gx \preccurlyeq h \\ and \quad Ax = b \end{aligned}$$

The kernel dual SVM reads $\max_\alpha \left\{ -\frac{1}{2} \sum_{i,j=1} \alpha_i \alpha_j y_i y_j k(x_i, x_j) + \sum_{i=1}^{N} \alpha_i \right\}$.
The maximization problem can be transformed into a minimization problem by multiplying by -1:

$$\min_\alpha \left\{ \frac{1}{2} \sum_{i,j=1} \alpha_i \alpha_j y_i y_j k(x_i, x_j) - \sum_{i=1}^{N} \alpha_i \right\}$$

$\underline{\frac{1}{2} x^T P x + q^T x}$:

$$x = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{bmatrix} \qquad q = \begin{bmatrix} -1 \\ -1 \\ \vdots \\ -1 \end{bmatrix}$$

$$P = \begin{bmatrix} y_1 y_1 k(x_1, x_1) & \cdots & y_1 y_N k(x_1, x_N) \\ \vdots & \ddots & \vdots \\ y_N y_1 k(x_N, x_1) & \cdots & y_N y_N k(x_N, x_N) \end{bmatrix}$$

$\underline{Ax = b}$:

$$\sum_{i=1}^{N} \alpha_i y_i = 0 \qquad A^T = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \qquad b = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$\underline{Gx \preccurlyeq h}$:
We have two inequalities which can be concatenated to one matrix G and to a vector h.

Max: $\alpha_i \geq 0$ $\qquad\qquad$ $\alpha_i \leq C$

Min: $-\alpha_i \leq 0$

$$\begin{bmatrix} -1 & & \\ & \ddots & \\ & & -1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{bmatrix} \leq \begin{bmatrix} C \\ C \\ \vdots \\ C \end{bmatrix}$$

And this leads to G and h.

$$G = \begin{bmatrix} -1 & & & & \\ & -1 & & & \\ & & \ddots & & \\ & & & & -1 \\ 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & & 1 \end{bmatrix} \qquad h = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ C \\ C \\ \vdots \\ C \end{bmatrix}$$

sheet7

January 11, 2021

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.**

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

def getGaussianKernel(X1,X2,scale):
    ### TODO: REPLACE BY YOUR OWN CODE
    Gram = scipy.spatial.distance.cdist(X1,X2) ** 2
    K = numpy.exp(Gram/(-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:

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

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`.**

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

def getQPMatrices(K,T,C):
    ### TODO: REPLACE BY YOUR CODE
    N = K.shape[0]
    T = T.reshape(N,1)

    P = K*T*T.reshape(1,N)

    q = numpy.ones((N,1)) * -1

    G = numpy.concatenate((numpy.eye(N),-1*numpy.eye(N)),axis=0)

    h = numpy.concatenate((numpy.ones((N,1))*C,numpy.zeros((N,1))),axis=0)

    A = numpy.transpose(T)

    b = numpy.zeros((1,1))

    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:

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

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

[3]:
```
def getTheta(K,T,alpha,C):
    ### TODO: REPLACE BY YOUR CODE

    M=numpy.nonzero((alpha>0) * (alpha<C))[0][0]

    k_M=K[M]

    theta=T[M]-numpy.sum(k_M*alpha*T)

    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**

[4]:
```
class GaussianSVM:

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

        self.C, self.scale = C*1.0, scale*1.0

    def fit(self,X,T):

        ### TODO: REPLACE BY YOUR CODE
        K = getGaussianKernel(X,X,self.scale)
        P,q,G,h,A,b = getQPMatrices(K,T,self.C)
```

```
        P,q,G,h,A,b=cvxopt.matrix(P),cvxopt.matrix(q),cvxopt.matrix(G),cvxopt.
↪matrix(h),cvxopt.matrix(A),cvxopt.matrix(b)

        sol=cvxopt.solvers.qp(P,q,G,h,A,b)
        self.alpha=sol['x']

        self.alpha=numpy.array(self.alpha).flatten()
        self.alpha[self.alpha < 0.5*1e-2] = 0
        M = (self.alpha>0.0)*(self.alpha<self.C)
        self.theta=getTheta(K,T,self.alpha,self.C)
        self.data = X
        self.X=X[M]

        self.T=T
        self.alpha=self.alpha
        ###

    def predict(self,X):

        ### TODO: REPLACE BY YOUR CODE
        Y = numpy.sign(getGaussianKernel(X,self.data,self.scale)@(self.
↪alpha*self.T)+self.theta)
        return Y
        ###
```

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.

[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:  164  train: 0.997  test: 0.921
scale=     30.0  C=    100.0  nSV:  162  train: 1.000  test: 0.918
scale=     30.0  C=   1000.0  nSV:  162  train: 1.000  test: 0.918
scale=     30.0  C=  10000.0  nSV:  162  train: 1.000  test: 0.918

scale=    100.0  C=     10.0  nSV:   82  train: 0.965  test: 0.932
scale=    100.0  C=    100.0  nSV:   69  train: 0.987  test: 0.940
scale=    100.0  C=   1000.0  nSV:   67  train: 0.998  test: 0.932
scale=    100.0  C=  10000.0  nSV:   60  train: 1.000  test: 0.926

scale=    300.0  C=     10.0  nSV:   58  train: 0.802  test: 0.793
scale=    300.0  C=    100.0  nSV:   45  train: 0.946  test: 0.927
scale=    300.0  C=   1000.0  nSV:   34  train: 0.970  test: 0.938
scale=    300.0  C=  10000.0  nSV:   29  train: 0.991  test: 0.941

scale=   1000.0  C=     10.0  nSV:   62  train: 0.841  test: 0.835
scale=   1000.0  C=    100.0  nSV:   51  train: 0.800  test: 0.794
scale=   1000.0  C=   1000.0  nSV:   44  train: 0.889  test: 0.877
scale=   1000.0  C=  10000.0  nSV:   36  train: 0.939  test: 0.920

scale=   3000.0  C=     10.0  nSV:   78  train: 0.873  test: 0.849
scale=   3000.0  C=    100.0  nSV:   61  train: 0.855  test: 0.848
scale=   3000.0  C=   1000.0  nSV:   51  train: 0.856  test: 0.855
scale=   3000.0  C=  10000.0  nSV:   45  train: 0.874  test: 0.856
```

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.