Kernel methods in machine learning
Homework 2

Thibault de SURREL
thibault.de-surrel@ensta-paris.fr

Exercice 1. Support Vector Classifier

1.

(a) The expression of the Lagrangian for the problem given in the exercise is

$$L(f, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\mu}) = \frac{1}{2}\|f\|^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i(y_i(f(x_i)+b)+\xi_i-1) - \sum_{i=1}^{n}\mu_i\xi_i$$

which can be rewritten using matrix notation, if we define $\boldsymbol{Y} = \begin{pmatrix} y_1 & \cdots & y_n \end{pmatrix}^T$ and $\boldsymbol{F} = \begin{pmatrix} f(x_1) & \cdots & f(x_n) \end{pmatrix}^T$, as

$$\boxed{L(f, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\mu}) = \frac{1}{2}\|f\|^2 + C\boldsymbol{\xi}^T\mathbb{1} - (\mathrm{diag}(y_i)\boldsymbol{\alpha})^T\boldsymbol{F} - \boldsymbol{\alpha}^T(b\boldsymbol{Y} + \boldsymbol{\xi} - \mathbb{1}) - \boldsymbol{\mu}^T\boldsymbol{\xi}}$$

(b) The dual problem is

$$\boxed{\begin{array}{ll} \min\limits_{\boldsymbol{\alpha}} & \frac{1}{2}\boldsymbol{\alpha}^T M\boldsymbol{\alpha} - \boldsymbol{\alpha}^T\mathbb{1} \\ \text{s.t.} & \boldsymbol{\alpha}^T\boldsymbol{Y} = 0 \\ & 0 \leq \boldsymbol{\alpha} \leq C\mathbb{1} \end{array}} \tag{1}$$

where $M = (y_i y_j k(x_i, x_j))_{i,jx}$. We can also express f with relevant quantities :

$$\boxed{f = \sum_{i=1}^{n}\alpha_i y_i k(x_i, \cdot)}$$

(c) The support vector point x_i are characterized by $\boxed{0 < \alpha_i < C}$.

2.

You can find my implementation of method `kernel` of the classes RBF and Linear at figure 1 as well as my implementation of the `KernelSVC` at figure 2. The results of the Kernel Support Vector Classifier on the provided dataset is also given at figure3.

```python
class RBF:
    def __init__(self, sigma=1.):
        self.sigma = sigma  ## the variance of the kernel
    def kernel(self,X,Y):
        ## Input vectors X and Y of shape Nxd and Mxd
        return   np.exp(-np.power(np.linalg.norm(X[:,None,:]-Y[None,:,:],axis=2),2)/(2*self.sigma**2)) ## Matrix of shape NxM

class Linear:
    def kernel(self,X,Y):
        ## Input vectors X and Y of shape Nxd and Mxd
        return   X@Y.T ## Matrix of shape NxM
```

Figure 1: My implementation of the method `kernel` of the classes RBF and Linear

```python
class KernelSVC:
    def __init__(self, C, kernel, epsilon = 1e-3):
        self.type = 'non-linear'
        self.C = C
        self.kernel = kernel
        self.alpha = None
        self.support = None
        self.epsilon = epsilon
        self.norm_f = None
        self.X = None
        self.y = None

    def fit(self, X, y):
       #### You might define here any variable needed for the rest of the code
        self.X = X
        self.y = y
        N = len(y)
        M = np.diag(y)@kernel(X,X)@np.diag(y)
        # Lagrange dual problem
        def loss(alpha):
            return  (1/2)*alpha.T@M@alpha - np.sum(alpha)#'''--------------dual loss ------------------ '''

        # Partial derivate of Ld on alpha
        def grad_loss(alpha):
            return  M@alpha - np.ones_like(alpha)# '''----------------partial derivative of the dual loss wrt alpha -----------------'''

        # Constraints on alpha of the shape :
        # -  d - C*alpha  = 0
        # -  b - A*alpha >= 0
        fun_eq = lambda alpha: alpha.T@y # '''----------------function defining the equality constraint------------------'''
        jac_eq = lambda alpha: y   #'''----------------jacobian wrt alpha of the  equality constraint------------------'''
        fun_ineq = lambda alpha: np.concatenate((C*np.ones_like(alpha) - alpha,alpha))  # '''---------------function defining the inequality constraint-------------------'''
        jac_ineq = lambda alpha:  np.concatenate((-np.eye(N),np.eye(N))) # '''---------------jacobian wrt alpha of the  inequality constraint-------------------'''

        constraints = ({'type': 'eq',  'fun': fun_eq, 'jac': jac_eq},
                       {'type': 'ineq',
                        'fun': fun_ineq ,
                        'jac': jac_ineq})

        optRes = optimize.minimize(fun=lambda alpha: loss(alpha),
                                   x0=np.ones(N),
                                   method='SLSQP',
                                   jac=lambda alpha: grad_loss(alpha),
                                   constraints=constraints)
        self.alpha = optRes.x

        ## Assign the required attributes
        indices =  np.where((self.alpha>self.epsilon) & (self.alpha<C))[0]
        self.margin_points = X[indices,:] #'''------------------- A matrix with each row corresponding to a point that falls on the margin ------------------'''
        self.b =  np.mean(y[indices] - self.separating_function(self.margin_points))#''' -----------------offset of the classifier------------------ '''
        self.norm_f = self.alpha.T@kernel(X,X)@self.alpha # '''------------------------RKHS norm of the function f ------------------------------'''
        self.support = X[np.where(self.alpha>self.epsilon)[0],:]

    ### Implementation of the separting function $f$
    def separating_function(self,x):
        # Input : matrix x of shape N data points times d dimension
        # Output: vector of size N
        return np.sum(np.diag(self.y*self.alpha)@kernel(self.X,x),axis=0)

    def predict(self, X):
        """ Predict y values in {-1, 1} """
        d = self.separating_function(X)
        return 2 * (d+self.b> 0) - 1
```

Figure 2: My implementation of the method `KernelSVC`

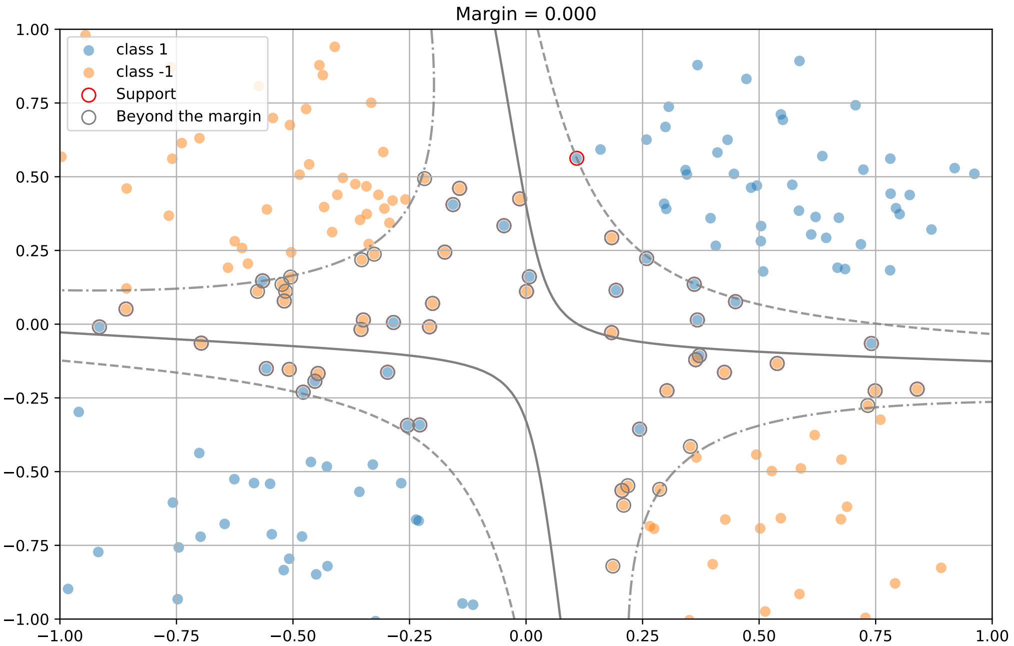


Figure 3: Results of the Kernel Support Vector Classifier

Exercise 2 : Kernel PCA

1.

Let $v \in \mathcal{H}$ be a non-trivial eigenvectors of the operator C, i.e. an element $v \in \mathcal{H}$ such that $Cv = v$ for positive and $\|v\| = 1$. We have that

$$Cv = \frac{1}{N}\sum_{i=1}^{N}\left(\tilde{\varphi}(X_i)\otimes\tilde{\varphi}(X_i)\right)v = \frac{1}{N}\sum_{i=1}^{N}\langle\tilde{\varphi}(X_i), v\rangle\tilde{\varphi}(X_i) = \lambda v. \tag{2}$$

So, as $\lambda \neq 0$, we have

$$v = \frac{1}{\lambda N}\sum_{i=1}^{N}\langle\tilde{\varphi}(X_i), v\rangle\tilde{\varphi}(X_i) = \sum_{i=1}^{N}\alpha_i\tilde{\varphi}(X_i) \tag{3}$$

where $\alpha_i = \frac{1}{\lambda N}\langle\tilde{\varphi}(X_i), v\rangle$. Therefore, we have that $v \in \mathrm{Vect}(\tilde{\varphi}(X_1), ..., \tilde{\varphi}(X_N))$.

For $i \in \{1, ..., N\}$, we have, as $Cv = \lambda v$,

$$\lambda\langle\tilde{\varphi}(X_i), v\rangle = \langle\tilde{\varphi}(X_i), Cv\rangle$$

So, using the expression of v from 3, and the one of Cv from 2, we have

$$\begin{aligned}
\lambda\langle\tilde{\varphi}(X_i), v\rangle = \langle\tilde{\varphi}(X_i), Cv\rangle &\iff \lambda\sum_{j=1}^{N}\alpha_j\langle\tilde{\varphi}(X_i), \tilde{\varphi}(X_j)\rangle = \frac{1}{N}\sum_{j=1}^{N}\langle\tilde{\varphi}(X_j), v\rangle\langle\tilde{\varphi}(X_i), \tilde{\varphi}(X_j)\rangle \\
&\iff N\lambda\sum_{j=1}^{N}\alpha_j\langle\tilde{\varphi}(X_i), \tilde{\varphi}(X_j)\rangle = \sum_{j=1}^{N}\sum_{k=1}^{N}\alpha_k\langle\tilde{\varphi}(X_j), \tilde{\varphi}(X_k)\rangle\langle\tilde{\varphi}(X_i), \tilde{\varphi}(X_j)\rangle \\
\iff N\lambda G\boldsymbol{\alpha} = G^2\boldsymbol{\alpha}
\end{aligned}$$

Where $G = (\langle\tilde{\varphi}(X_i), \tilde{\varphi}(X_j)\rangle)_{i,j\in\{1,...,N\}}$ and $\boldsymbol{\alpha} = (\alpha_1 \cdots \alpha_N)^T$. Therefore, we can solve the following eigenvalue problem

$$G\boldsymbol{\alpha} = N\lambda\boldsymbol{\alpha} \tag{4}$$

for nonzero eigenvalues. The solution of problem 4 are all solutions of the problem $N\lambda G\boldsymbol{\alpha} = G^2\boldsymbol{\alpha}$. Finally, we ask the solutions $\boldsymbol{\alpha}$ belonging to nonzero eigenvalues to be normalized in $\mathcal{H}$ (to be precise, we want their corresponding vector in $\mathcal{H}$ to be normalized). So we impose the following normalization condition :

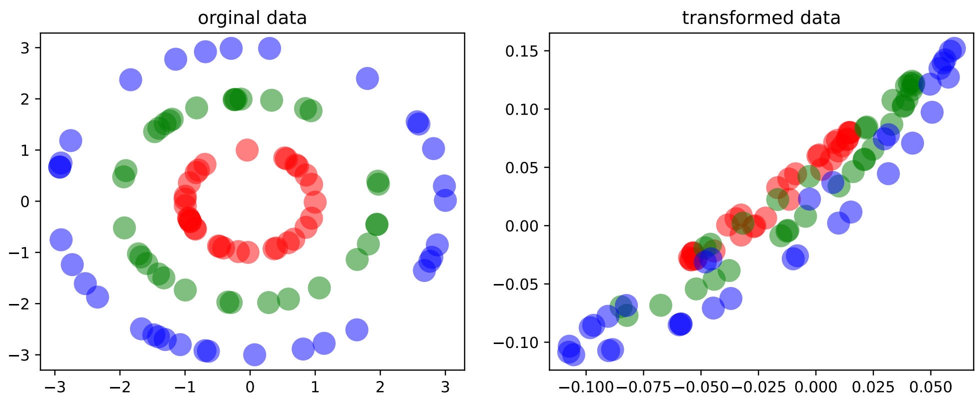
$$\lambda\|\boldsymbol{\alpha}\| = 1.$$

2.

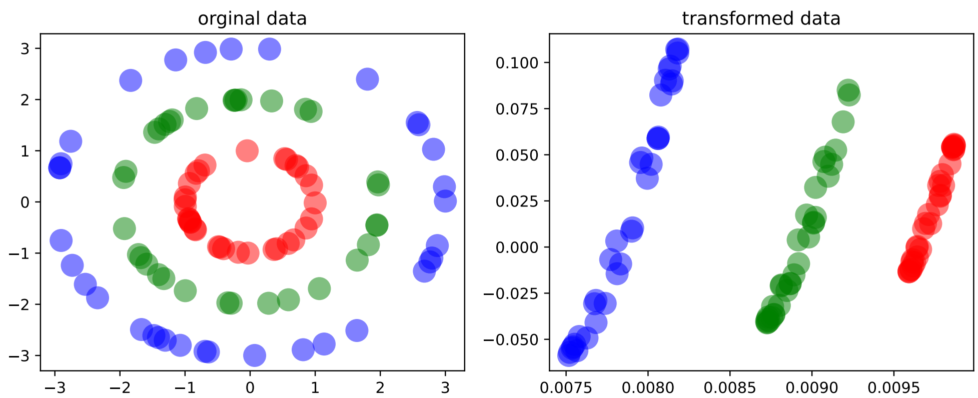
You can find my implementation of the Kernel PCA at figure 4 and the results at figure 5. The results are quite strange. In fact, when I tried to normalize the Gram matrix G in order to have the centered features $\tilde{\varphi}(X_i)$, using the formula from the course ($(I_N - U)G(I_N - U)$ where $U_{i,j} = 1/N$), the results were not very good as one can see at figure 5a. The three components of the dataset are not well separated. However, when I do not normalize the Gram matrix G, the results are better, as one can see on figure 5b. Here, the three components are clearly separated and the PCA seams to work better. I did not achieve to understand what was the problem in my original implementation.

```python
class KernelPCA:

    def __init__(self,kernel, r=2):
        self.kernel = kernel          # <---
        self.alpha = None # Matrix of shape N times d representing the d eingenvectors alpha corresp
        self.lmbda = None # Vector of size d representing the top d eingenvalues
        self.support = None # Data points where the features are evaluated
        self.r = r ## Number of principal components
    def compute_PCA(self, X):
        # assigns the vectors
        self.support = X
        N = X.shape[0]
        K = self.kernel(X,X)

        # center the Kernel matrix
        one_n = np.ones((N,N)) / N
        K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

        #Get the eigenvalues/ eigen vectors
        eigvals, eigvecs = np.linalg.eigh(K)

        # Sort the eigen vectors/values
        idx = eigvals.argsort()[::-1]
        eigvals = eigvals[idx]
        eigvecs = eigvecs[:,idx]

        idx = eigvals>0
        eigvals = eigvals[idx]
        eigvecs = eigvecs[:,idx]

        # Normalize the eigenvalues
        eigvecs /= np.sqrt(eigvals)

        self.alpha = eigvecs
        self.lmbda = eigvals
    def transform(self,x):
        # Input : matrix x of shape N data points times d dimension
        # Output: vector of size N
        K = self.kernel(self.support,x)
        return K@self.alpha[:,:self.r]
```

Figure 4: My implementation of Kernel PCA



(a) Results of the Kernel PCA when the Gram matrix is normalized



(b) Results of the Kernel PCA when the Gram matrix is not normalized

Figure 5: Different results of Kernel PCA