Kernel methods for machine learning
Homework 2

Simon Queric simon.queric@telecom-paris.fr

February 25, 2024

Exercise 1. Support Vector Classifier

1. (a) The Lagrangian writes :

$$L(f, b, \xi, \alpha, \mu) = \frac{1}{2} f^T K f + C\xi^T \mathbf{1} \ - \xi^T(\alpha + \mu) + \alpha^T \mathbf{1} \ - b\alpha^T y - (\text{diag}(y)\alpha)^T K f$$

   (b) We get the dual problem by taking the infimum on (f, b, ξ) :

$$\max_{\alpha \in \mathbb{R}^n} \alpha^T \mathbf{1} \ - \frac{1}{2}(\text{diag}(y)\alpha)^T K(\text{diag}(y)\alpha)$$

   with constraints :

$$0 \leqslant \alpha_i \leqslant C, \alpha^T y = 0$$

   We can express $f(x)$ in function of α :

$$f(x) = \sum_{i=1}^{n} y_i \alpha_i K(x, x_i)$$

   (c) With strong duality we get

$$\begin{cases} \alpha_i(1 - x_i - y_i(f(x_i) + b)) = 0 \\ (C - \alpha_i)\xi_i = 0 \end{cases}$$

   It implies that

$$0 < \alpha_i < C \iff y_i(f(x_i) + b) = 1$$

   which characterizes the support vector points.

2. (a)

```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
        X2 = (X**2).sum(axis=-1) # size N
        Y2 = (Y**2).sum(axis=-1) # size M
        XdotY = X.dot(Y.T)
        diff2 = X2[:,None] + Y2[None,:] - 2*XdotY
        return np.exp(-diff2/(2*self.sigma**2))    ## Matrix of shape NxM
```

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

Figure 1: RBF and Linear kernels

(b)

```python
def fit(self, X, y):
   #### You might define here any variable needed for the rest of the code
    self.X = X
    self.y = y
    K = self.kernel(X, X)
    N = len(y)

    # Lagrange dual problem
    def loss(alpha):
        return  -np.sum(alpha) + (y*alpha).T@K@(y*alpha) / 2

    # Partial derivate of Lagrange dual on alpha
    def grad_loss(alpha):
        return -np.ones_like(N) + np.diag(y)@K@(y*alpha)

    # Constraints on alpha of the shape :
    # -  d - C*alpha  = 0
    # -  b - A*alpha >= 0

    fun_eq = lambda alpha: alpha@y  # '''----------------function defining the equality constraint-------------
    jac_eq = lambda alpha:  y  #'''----------------jacobian wrt alpha of the  equality constraint--------------
    fun_ineq = lambda alpha: np.concatenate((alpha, self.C-alpha))  # '''---------------function defining the in
    I = np.zeros((2*N, N))
    I[:N,:] = np.eye(N)
    I[N:,:] = -np.eye(N)
    jac_ineq = lambda alpha: I  # '''---------------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

    self.support = self.X[(self.epsilon<self.alpha) & (self.alpha < self.C)]
    fx = (self.alpha[:,None]*y[:,None]*K).sum(axis=-1)

    self.b = np.mean(y[(self.epsilon<self.alpha) & (self.alpha < self.C)] \
                - fx[(self.epsilon<self.alpha) & (self.alpha < self.C)]) #''' ----------------offset of the
    self.norm_f = np.sum(((y*self.alpha)@K@(y*self.alpha)) **2) # '''-----------------------RKHS norm of the fu

    self.y = y[(self.epsilon<self.alpha) & (self.alpha < self.C)]
    self.alpha = self.alpha[(self.epsilon<self.alpha) & (self.alpha < self.C)]
```

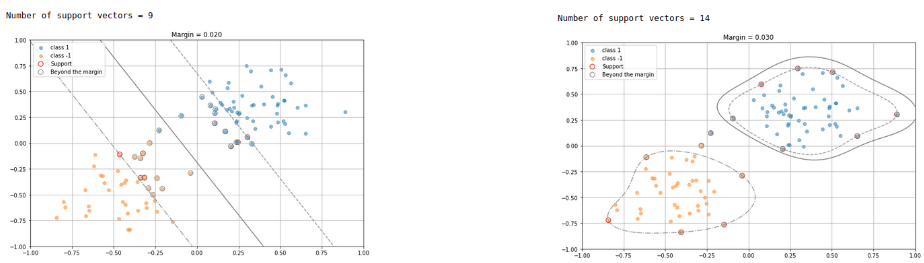
Figure 2: Method *fit* of the KernelSVC class

(c)

```python
### 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
    similarity_matrix = self.kernel(x, self.support)
    separating_fun = (similarity_matrix*self.y[None,:]*self.alpha[None,:]).sum(axis=-1)
    return separating_fun
```

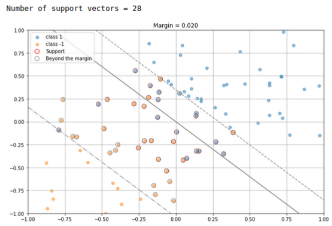
Figure 3: Method *separating_function* of the KernelSVC class

(d)

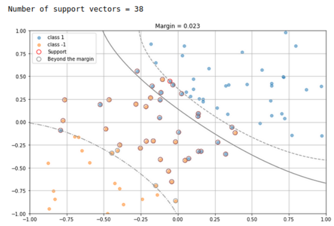


(a) Linear kernel. $C = 0.5$

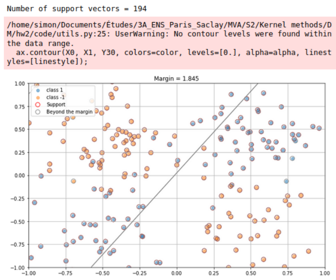
(b) RBF kernel. $\sigma = 0.3, C = 2$



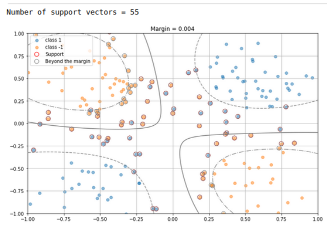
(a) Linear kernel. $C = 3, \varepsilon = 10^{-11}$



(b) RBF kernel. $\sigma = 1, C = 1$



(c) Linear kernel. $C = 1$



(d) RBF kernel. $\sigma = 0.6, C = 1.5$

Figure 5: SVM classification for several point clouds, using linear and RBF kernels.

Exercise 2. Kernel Ridge Regression

1. According to the representer theorem, the regression function f can be expressed as

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

with $\alpha \in \mathbb{R}^N$ and $b \in \mathbb{R}$

The optimization problem can be rewritten as

$$\min_{\alpha, b} \frac{1}{N} \|K\alpha - y + b\mathbf{1}\|_2^2 + \frac{\lambda}{2} \alpha^T K \alpha$$

Let K be the Gram matrix of $(x_i)_{1 \leq i \leq N}$.

Then, let A be the square matrix of size $N + 1$ with K in the top left corner and zero entries otherwise.

Let B the matrix K with an additional column of ones.

Then the solution of the problem is $(\alpha^*, b^*)^T = (B^T B + \frac{N\lambda}{2} A)^{-1} B^T y$

2. (a)

   (b)

```python
def fit(self, X, y):
    N = len(y)
    K = self.kernel(X, X)
    A = np.zeros((N+1, N+1))
    A[:N,:N] = K
    B = np.zeros((N, N+1))
    B[:,:N] = K
    B[:,-1] = 1
    self.support = X
    theta = np.linalg.solve(B.T@B + self.lmbda*N / 2 * A, B.T@y)
    self.b = theta[-1]
    self.alpha = theta[:-1]

### Implementation of the separting function $f$
def regression_function(self,x):
    # Input : matrix x of shape N data points times d dimension
    # Output: vector of size N
    similarity_matrix = kernel(x, self.support)
    regression = similarity_matrix.dot(self.alpha)
    return regression
```

```python
def fit(self, X, y):
    N, q = y.shape
    self.support = X
    K = self.kernel(X, X)
    self.b = np.zeros(q)
    self.alpha = np.zeros((q, N))
    A = np.zeros((N+1, N+1))
    A[:N,:N] = K
    B = np.zeros((N, N+1))
    B[:,:N] = K
    B[:,-1] = 1
    theta = np.linalg.solve(B.T@B + self.lmbda*N / 2 * A, B.T@y)
    self.b = theta[-1,:]
    self.alpha = theta[:-1,:]

### Implementation of the separting function $f$
def regression_function(self,x):
    # Input : matrix x of shape N data points times d dimension
    # Output: vector of size N
    q = len(self.b)
    n, d = x.shape
    similarity_matrix = kernel(x, self.support)
    outputs = similarity_matrix.dot(self.alpha)
    return outputs
```

Figure 6: Methods *fit* and *regression_function* for the classes KernelRR and MultivariateKernelRR

   (c)

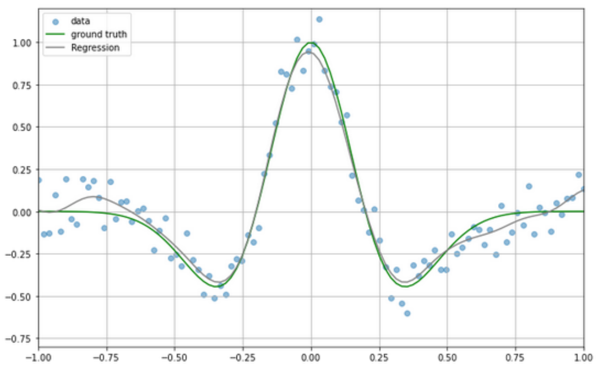


Figure 7: Univariate kernel ridge regression with RBF kernel. $\sigma = 0.1, \lambda = 0.01$

Exercise 3. Kernel PCA

1. Let v be a non trivial eigenvector of C and $\lambda > 0$ it's associated eigenvalue. Then $\lambda v = Cv = \frac{1}{N}\sum_{i=1}^{N} \underbrace{\langle \widetilde{\varphi}(X_i), v\rangle}_{\alpha_i} \widetilde{\varphi}(X_i)$ It follows that $\lambda \alpha_i = \sum_{k=1}^{N} G_{k,i}\alpha_k$ where $G_{i,j} = \frac{1}{N}\langle \widetilde{\varphi}(X_i), \widetilde{\varphi}(X_j)\rangle$

2. (a)

   (b)

```python
def compute_PCA(self, X):
    # assigns the vectors
    self.support = X
    N = X.shape[0]
    K = self.kernel(X, X)
    G = K - K.mean(axis=-1)[:,None] - K.mean(axis=-1)[None,:] + K.mean()  #

    self.G = G
    eigvalues, eigvectors = np.linalg.eigh(self.G)
    eigvalues = eigvalues[::-1]
    eigvectors = eigvectors[:,::-1]
    self.lmbda = eigvalues[:self.r]
    self.alpha = eigvectors[:,:self.r]/np.sqrt(self.lmbda)[None,:]

def transform(self,x):
    # Input : matrix x of shape N data points times d dimension
    # Output: vector of size N
    N, d = x.shape
    n = self.support.shape[0]
    ones = np.ones((n, n)) / n
    K = self.kernel(self.support, self.support)
    G = self.kernel(x, self.support) # N x n
    output = np.zeros((N, self.r))
    G = G  - G.mean(axis=-1)[:,None] - K.mean(axis=-1)[None,:] + K.mean()

    output = G.dot(self.alpha)

    return output
```

Figure 8: Methods *compute_ PCA* and *transform* of the class KernelPCA

(c)

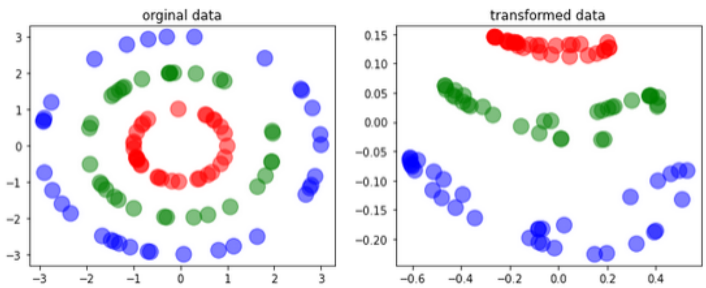


Figure 9: Kernel PCA for a dataset of circles with RBF kernel for $\sigma = 4$. I display the transformed data along the second and third components.

3. (a)

   (b)

```python
class Denoiser:
    def __init__(self, kernel_encoder, kernel_decoder,dim_pca, lmbda):
        self.pca = KernelPCA(kernel_encoder, r=dim_pca)
        self.ridge_reg = MultivariateKernelRR(kernel_decoder, lmbda=lmbda)

    def fit(self,train):
        self.pca.compute_PCA(train)
        encoding = self.pca.transform(train)
        self.ridge_reg.fit(encoding, train)

    def denoise(self,test):
        n, d = test.shape
        encoding = self.pca.transform(test)
        denoised_data = self.ridge_reg.predict(encoding)
        return denoised_data.reshape(n, int(np.sqrt(d)), int(np.sqrt(d)))
```

Figure 10: Implementation of a Denoiser using Kernel PCA and Multivariate Kernel Ridge Regression.

   (c)

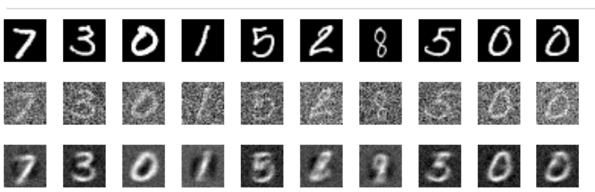


Figure 11: Result of the denoiser for a sample of the MNIST dataset. dim_pca $= 400, \lambda = 10^{-3}/2, \sigma_{\text{encoder}} = 15, \sigma_{\text{decoder}} = 10$.

Tuning hyperparameters to makes the denoiser working is quite challenging. It gives visually satisfying results.