MVA: Kernel methods in machine learning (2022/2023)

Assignment 2

Solution by Matias Etcheverry March 1, 2023

1 Exercice 1. Support Vector Classifier

We want to solve the following minimization problem:

$$\begin{aligned} \min_{f,b,\xi} \quad & \frac{1}{2}\|f\|_{\mathcal{H}}^2 + C\sum_{i=1}^n \xi_i \\ \text{s.t.} \quad & y_i(f(x_i)+b)) \geq 1-\xi_i, \\ & \xi_i \geq 0 \end{aligned}$$

1. (a) Computing the Lagrangian gives the following expression: let $\alpha_i, \mu_i \in \mathbb{R}$

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

(b) Using the optimality conditions on the Lagrangian, we obtain the following constraints:

- on ξ_i:

$$\frac{\partial \mathcal{L}}{\partial \xi_i} = 0 \Longrightarrow \boxed{\alpha_i + \mu_i = C}$$

This equation also had the constraint $0 \leq \alpha_i \leq C$

- on f:

$$\frac{\partial \mathcal{L}}{\partial f} = 0 \Longrightarrow \boxed{f = \sum_{i=1}^n \alpha_i y_i k_{x_i}}$$

with $k_{x_i} \in \mathcal{H} : t \mapsto k(t, x_i)$

- on b:

$$\frac{\partial \mathcal{L}}{\partial b} = 0 \Longrightarrow \boxed{-\sum_{i=1}^n \alpha_i y_i = 0}$$

We obtain the following dual problem:

$$\begin{aligned} \min_{\alpha \in \mathbb{R}} \quad & \frac{1}{2}\sum_{i,j=1}^n \alpha_i\alpha_j y_i y_j k_{x_j}(x_i) - \sum_{i=1}^n \alpha_i \\ \text{s.t.} \quad & -\sum_{i=1}^n \alpha_i y_i = 0, \\ & 0 \leq \alpha_i \leq C \end{aligned}$$

(c) By applying the KKT conditions (complementary slackness) on the Lagrangian, we have: for i such that x_i is on the margin:

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

By additionning the 2 lines and knowing that $\alpha_i + \mu_i = C$, we get $\xi_i = 0$. This leads to $\boxed{\alpha_i > 0}$ when x_i is on the margin.

2. (a) code for the RBF and Linear kernels:

```python
class RBF:
    def __init__(self, sigma=1.0):
        self.sigma = sigma  ## the variance of the kernel

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

class Linear:
    def __init__(self):
        pass

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

(b) In the `fit` method of the `KernelSVC` class, I added a second constraint to the optimizer. This second constraint encodes the positivity of α.

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

    # Lagrange dual problem
    def loss(alpha):
        alpha_y = alpha * y
        norm_term = alpha_y.T @ kernel @ alpha_y
        alpha_term = np.sum(alpha)
        return -alpha_term + 0.5 * norm_term  #'''--------------dual loss
    ------------------ '''

    # Partial derivate of Ld on alpha
    def grad_loss(alpha):
        alpha_y = alpha * y
        return (alpha_y[None, :] * kernel).sum(axis=1) * y - 1  #
    '''----------------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 * y).sum()  # '''----------------function
    defining the equality constraint-----------------'''
    jac_eq = lambda alpha: -y  #'''----------------jacobian wrt alpha of the
    equality constraint-----------------'''
    fun_ineq = lambda alpha: self.C - alpha  # '''---------------function
    defining the inequality constraint-------------------'''
    jac_ineq = lambda alpha: -np.eye(len(alpha))  # '''---------------jacobian
    wrt alpha of the  inequality constraint-------------------'''
    fun_ineq2 = lambda alpha: alpha  # '''---------------function defining the
    positivity constraint-------------------'''
    jac_ineq2 = lambda alpha: np.eye(len(alpha))  # '''---------------jacobian
    wrt alpha of the  positivity constraint-------------------'''

    constraints = (
        {"type": "eq", "fun": fun_eq, "jac": jac_eq},
        {"type": "ineq", "fun": fun_ineq, "jac": jac_ineq},
        {"type": "ineq", "fun": fun_ineq2, "jac": jac_ineq2},
    )
    optRes = optimize.minimize(
```

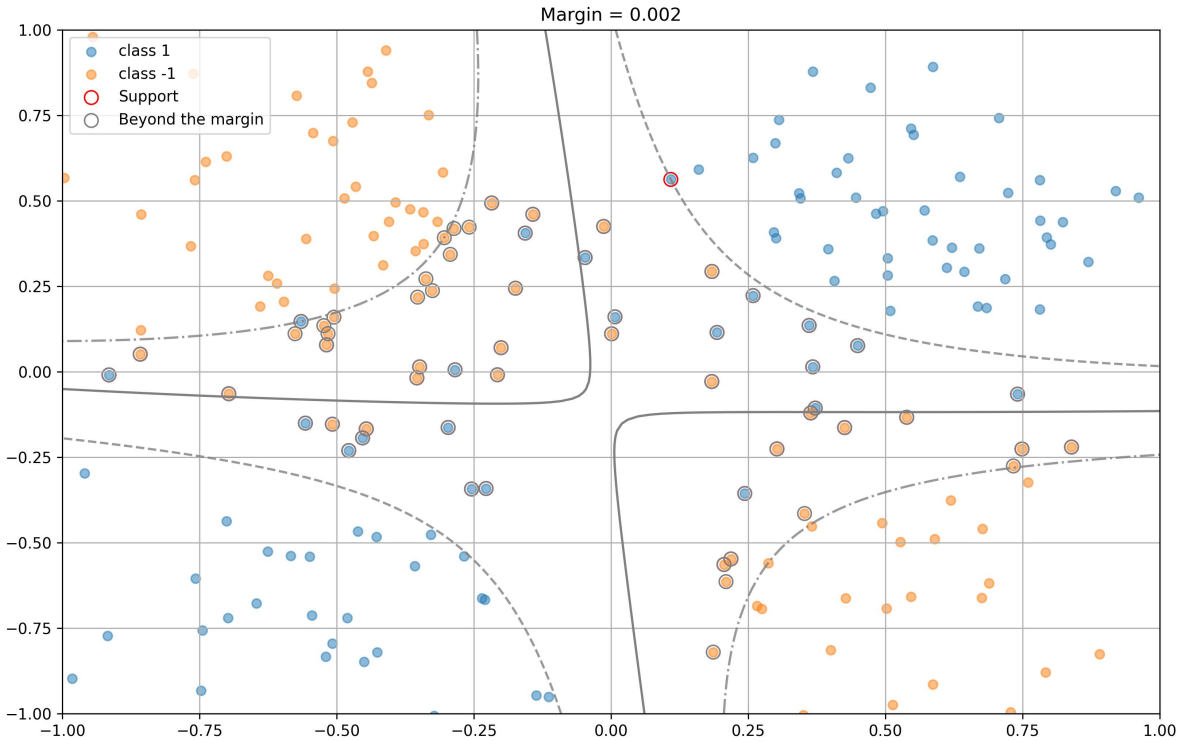
```
        fun=lambda alpha: loss(alpha),
        x0=np.ones(N),
        method="SLSQP",
        jac=lambda alpha: grad_loss(alpha),
        constraints=constraints,
    )
    self.alpha = optRes.x

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

(c) Code for the `separating_function` method:

```
def separating_function(self, x):
    # Input : matrix x of shape N data points times d dimension
    # Output: vector of size N
    return (self.alpha_y_support[None, :] * self.kernel(x, self.support)).sum(
    axis=1)
```

(d) Below are the results I obtained on the 3rd dataset, using the RBF kernel. The result is quite good.

`Number of support vectors = 45`



2 Exercice 2. Kernel PCA

1. We want to study vector v such that $Cv = \lambda v$, for positive λ and $\|v\| = 1$. Based on the definition of C, we have:

$$\begin{aligned} \lambda v &= Cv \\ &= \frac{1}{n}\sum_{i=1}^{n}(\tilde{\varphi}(X_i) \otimes \tilde{\varphi}(X_i))v \\ &= \frac{1}{n}\sum_{i=1}^{n}\langle \tilde{\varphi}(X_i) \mid v\rangle \, \tilde{\varphi}(X_i) \\ \text{i.e.} \quad v &= \sum_{i=1}^{n}\frac{1}{\lambda n}\langle \tilde{\varphi}(X_i) \mid v\rangle \, \tilde{\varphi}(X_i) \end{aligned}$$

This means that v is a combination of features $\tilde{\varphi}(X_i)$. We denote α_i such that $v = \sum_{i=1}^{n}\alpha_i\tilde{\varphi}(X_i)$. Thus, we have:

$$\begin{aligned} Cv &= \frac{1}{n}\sum_{i=1}^{n}(\tilde{\varphi}(X_i) \otimes \tilde{\varphi}(X_i))\sum_{j=1}^{n}\alpha_j\tilde{\varphi}(X_j) \\ &= \frac{1}{n}\sum_{i,j=1}^{n}\alpha_j\langle \tilde{\varphi}(X_i) \mid \tilde{\varphi}(X_j)\rangle \, \tilde{\varphi}(X_i) \end{aligned}$$

Thus, the condition $Cv = \lambda v$ is met when:

$$\frac{1}{n}\sum_{j=1}^{n}\alpha_j\langle \tilde{\varphi}(X_i) \mid \tilde{\varphi}(X_j)\rangle = \lambda\alpha_i$$

$$\text{i.e. when} \quad \boxed{(\frac{1}{n}G)\alpha = \lambda\alpha}$$

with $[G]_{i,j} = \langle \tilde{\varphi}(X_i) \mid \tilde{\varphi}(X_j)\rangle$

Warning: this condition is sufficient but not necessary. I didn't find how to make this a necessary condition.

Finally, the condition $\|v\| = 1$ is transformed in $\boxed{\alpha^T G\alpha = 1}$

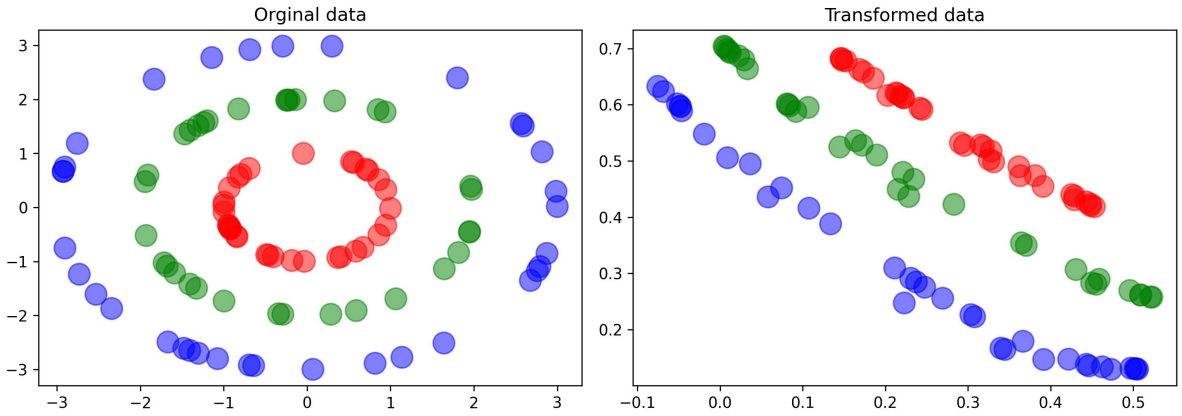
2. (a) `compute_PCA` method which finds the top r eigenvectors: of the matrix G.

```python
def compute_PCA(self, X):
    # assigns the vectors
    n = len(X)
    G = self.kernel(X, X)
    G = (
        (np.eye(n) - 1 / n * np.ones((n, n)))
        @ G
        @ (np.eye(n) - 1 / n * np.ones((n, n)))
    )
    eigenvalues, eigenvectors = np.linalg.eig(G / n)
    indices = np.argsort(eigenvalues)
    self.lmbda = eigenvalues[indices][: self.r]
    self.alpha = eigenvectors[indices][: self.r]
    self.support = X

```

(b) Implementation of the method `transform`:

```python
def transform(self, x):
    # Input : matrix x of shape N data points times d dimension
    # Output: vector of size N
    G = self.kernel(x, self.support)
    return G @ self.alpha.T

```

(c) With the kernel PCA, the classes are now linearly separable:



3. First of all, here is my implementation of `MultivariateKernelRR`

```
class MultivariateKernelRR:
    def __init__(self, kernel, lmbda):
        self.lmbda = lmbda
        self.kernel = kernel
        self.support = None
        self.alpha = None
        self.b = None
        self.type = "ridge"

    def fit(self, X, y):
        self.support = X
        kernel = self.kernel(X, X)
        n = len(y)
        self.alpha = np.linalg.solve(kernel + self.lmbda * n * np.eye(n), y)
        self.b = - np.mean(self.regression_function(X), axis=0)

    ### 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
        kernel = self.kernel(x, self.support)
        images = [(kernel[i, :][:, None] * self.alpha).sum(axis=0) for i in range(
    len(x))]
        images = np.stack(images, axis=0)
        return images

    def predict(self, X):
        """Predict y values in {-1, 1}"""
        return self.regression_function(X) + np.expand_dims(self.b, axis=0)

```

(a) Implementation of `fit` method:

```
def fit(self,train):
    n = len(train)
    self.pca.compute_PCA(train.reshape(n, -1))
    pca_components = self.pca.transform(train.reshape(n, -1))
    self.ridge_reg.fit(pca_components, train.reshape(n, -1))
```

(b) Implementation of `denoise` method:

```
def denoise(self,test):
    n = len(test)
    pca_components = self.pca.transform(test.reshape(n, -1))
    prediction = self.ridge_reg.predict(pca_components)
    return prediction.reshape((n, 28, 28))
```

(c) I cannot train on the full dataset due to memory limitations, but below are the results I obtained. We can safely say that my denoiser is not working.