Kernel methods for machine learning
Homework 2

Med Chiheb Yaakoubi

Exercice 1: Sobolev Spaces

Let's suppose that $\mathcal{H}$ is an RKHS and its reproducing kernel being K
We know that the functions : $t \mapsto \mathbf{1}_{[x,1]}(t)$ are in $\mathcal{H}$ hence:

$$\begin{aligned}\mathbf{1}_{[x,1]}(t) =< K_t, \mathbf{1}_{[x,1]}(.) >_{\mathcal{H}} &= \int_x^1 K(t,u)du + \int_0^1 \delta_x \partial_u K_t(u)du \\ &= \int_x^1 K(t,u)du + \partial_u K_t(x)\end{aligned}$$

We can now differentiate by x and we get:

$$-\delta(x-t) = -K(t,x) + \partial_u^2 K(t,x)$$

We can then solve this equation and find the reproducing kernels. Having the expression of the kernels we finally check that they verify the reproducing kernel properties and we get that $\mathcal{H}$ is an r.k

Exercice 2: Gaussian RKHS

(Q1) *K is positive definite*
We know that K writes :

$$K(x,t) = Const \times exp(-const \times (||x||^2) exp(-const \times (||y||^2) exp(const \times x^T y)$$

The first two exponentials constitute a p.d kernel since they are independant. The second term is of the form e^K where K is a p.d kernel. Since the multiplication of two p.d Kernels gives out a p.d Kernel we get the result.
(Q2) $H_\tau \subset H_\sigma \subset L_2(\mathbb{R}^d)$
We have $H_\sigma \subset L_2(\mathbb{R}^d)$ because the $(t \mapsto K_\sigma(x,t))_x \subset L_2(\mathbb{R}^d)$ and that $\mathcal{H}_\sigma$ is closure of the span of these functions. we can also notice that for any f $\in \mathcal{H}_\sigma$, since it's in $L_2(\mathbb{R}^d)$:

$$< f, K_x >_{\mathcal{H}} = f(x) = \frac{1}{(2\pi)^d} \int_{\mathbb{R}^d} \hat{f}(t) e^{itx} dt \tag{1}$$

We can also see that:

$$\widehat{K_\sigma(x,.)} = e^{-ixt} e^{-\sigma^2 \frac{||t||^2}{2}}$$

From this we can see that:

$$< f, K_x >_{\mathcal{H}_\sigma} = \frac{1}{(2\pi)^d} \int_{\mathbb{R}^d} \hat{f}(t) \overline{\widehat{K}_x(t)} e^{\sigma^2 \frac{||t||^2}{2}} dt$$

Since we got the expression of the dot product for the r.k functions (take $f = K_y$) we can generalize to the whole space and get:

$$< f, g >_{\mathcal{H}_\sigma} = \frac{1}{(2\pi)^d} \int_{\mathbb{R}^d} \hat{f}(t) \overline{\widehat{g}(t)} e^{\sigma^2 \frac{||t||^2}{2}} dt$$

Meaning that:

$$||f||_{\mathcal{H}_\sigma} = \int_{\mathbb{R}^d} \hat{f}(t)^2 e^{\sigma^2 ||t||^2/2} dt$$

for a function f $\in \mathcal{H}_\tau$ i.e $\hat{f}(t)^2 e^{\tau^2 ||t||^2/2}$ is integrable, hence $hatf(t)^2 e^{\sigma^2 ||t||^2/2}$ is also integrable by $\tau > \sigma$ which means f is in $\mathcal{H}_\sigma$
(Q3) from the above characterization, we can directly deduce the first inequality. for the second inequality we have to see $||f||_{L_2(\mathbb{R}^d)}$ using the parseval equality which leads us to:

$$||f||_{\mathcal{H}_\sigma} - ||f||_{L_2(\mathbb{R}^d)} = \int_{\mathbb{R}^d} \hat{f}(t)^2 (e^{\sigma^2 ||t||^2/2} - 1) dt$$

But since a simple differentiation can help us conclude that:

$$e^{\sigma^2 ||t||^2/2} - 1 \leq \frac{\sigma^2}{\tau^2} (e^{\tau^2 ||t||^2/2} - 1)$$

We get the second inequality
(Q4)for $\tau > 0$ let $f \in \mathcal{H}_\tau$, let also σ such that $\sigma < \tau$ if we use the previous inequality we get the result needed.

Exercice 3: Support Vector Classifier

(Q1\a) The lagrangian can be written as:

$$\begin{aligned}
\mathcal{L}(x, \alpha, \mu) &= \frac{1}{2}||f||^2 + C \sum_{i=1}^{N} \xi_i + \sum_{i=1}^{N} \alpha_i (1 - \xi_i - y_i(f(x_i) + b)) - \sum_{i=1}^{N} \mu_i \xi_i \\
&= \frac{1}{2}||f||^2 + \sum_{i=1}^{N} (C - \alpha_i - \mu_i) \xi_i + \sum_{i=1}^{N} \alpha_i (1 - y_i(f(x_i) + b))
\end{aligned}$$

where $x = (f, b, (\xi_i))$
(Q1\b) The dual problem is :

$$\begin{aligned} \max_{\alpha_i} \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j k(x_i, x_j) \\ \text{s.t.} \quad & \sum_{i=1}^{n} \alpha_i y_i = 0, \\ & 0 \leq \alpha_i \leq C \quad \text{for } i = 1, 2, \ldots, n. \end{aligned}$$

and

$$f = \sum_{i=1}^{N} \alpha_i y_i k(x_i, .)$$

(Q1\c) Using the complentary slackness conditions we get that the support vector points are the points x_i s.t. $\alpha_i > 0$
(Q2\a)

```python
class RBF:
    def __init__(self, sigma=1.):
        self.sigma = sigma  ## the variance of the kernel
    def kernel(self,X,Y):
        # We use here vectorized operations instead of loops for optimization
        # Compute pairwise squared Euclidean distances
        X_norm = np.sum(X**2, axis=1)[:, np.newaxis]  # Shape (N, 1)
        Y_norm = np.sum(Y**2, axis=1)[np.newaxis, :]  # Shape (1, M)
        distances = X_norm + Y_norm - 2 * np.dot(X, Y.T)  # Shape (N, M)

        # Compute the RBF kernel
        G = np.exp(-distances / (2 * self.sigma**2))
        return  G
```
✓ 0.0s

```python
class Linear:
    def kernel(self,X,Y):
        G = np.dot(X,Y.T)
        return G
```
✓ 0.0s

(Q2\b)

```python
def fit(self, X, y):
   #### You might define here any variable needed for the rest of the code
    N = len(y)
    K = self.kernel(X,X) # the gram matrix
    self.X = X
    self.y = y
    # Lagrange dual problem
    def loss(alpha):
        s_1 = np.sum(alpha)

        weighted_labels = y*alpha
        s_2=np.dot(weighted_labels.T,np.dot(K,weighted_labels))

        return  s_1-(1/2)*s_2

    # Partial derivate of Ld on alpha
    def grad_loss(alpha):
        weighted_labels = y*alpha
        return 1 - y * np.dot(K,weighted_labels)

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

    fun_eq = lambda alpha:  np.dot(y,alpha) # '''----------------function de
    jac_eq = lambda alpha:  y   #'''----------------jacobian wrt alpha of th
    fun_ineq = lambda alpha:  np.hstack([self.C-alpha,alpha]) # '''--------
    jac_ineq = lambda alpha:  np.vstack([-np.eye(N),np.eye(N)]) # '''------
```

```
#'''------------------- A matrix with each row corresponding to support vectors -

self.support =  X[self.alpha>0]

#''' -----------------offset of the classifier----------------- '''

ind1 = self.alpha>0
ind2 = self.alpha<self.C
ind_verif = ind1*ind2 # datapoints having their respective 0<alpha_i<C
indices = np.arange(N)[ind_verif]
f_opt = lambda i: np.dot(K[i],y*self.alpha)

self.b = np.mean(1/y[indices]-f_opt(indices))

# '''-----------------------RKHS norm of the function f -----------------------

weighted_labels = y*self.alpha

self.norm_f = np.dot(weighted_labels.T,np.dot(K,weighted_labels))
```

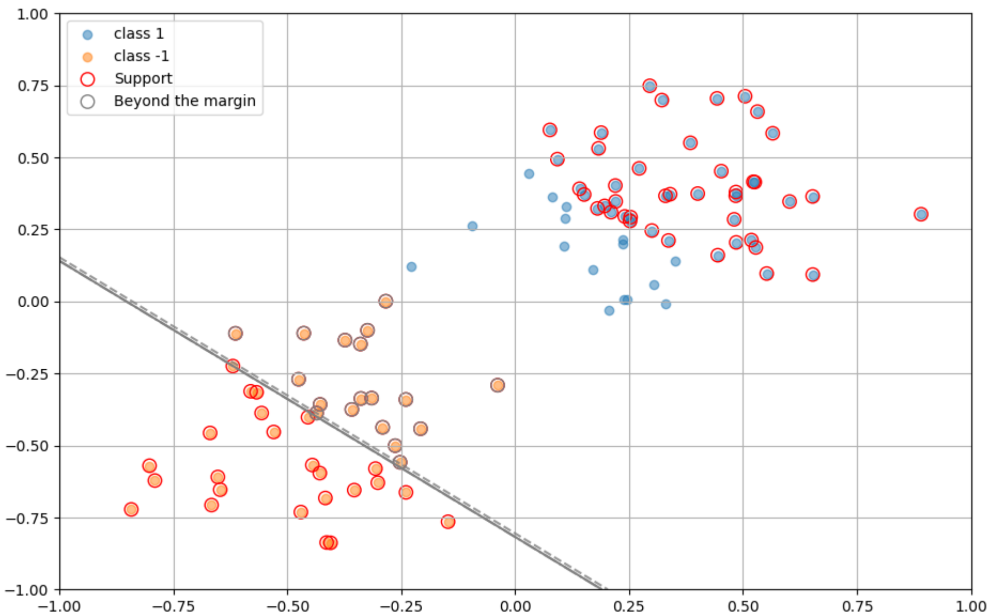
(Q2\c)

```
### 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
    K_pred = self.kernel(self.X,x).T
    return np.dot(K_pred,self.alpha*self.y)
```

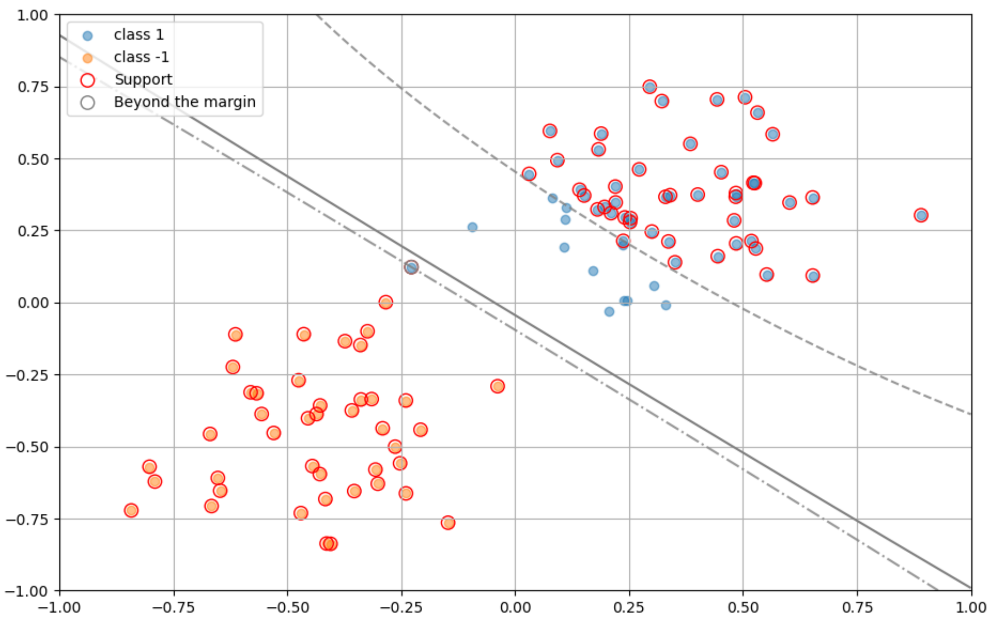
(Q2\d)

```python
def plotClassification(X, y, model=None, label='',  separatorLabel='Separator',
            ax=None, bound=[[-1., 1.], [-1., 1.]]):
    """ Plot the SVM separation, and margin """
    colors = ['blue','red']
    labels = [1,-1]
    cmap = pltcolors.ListedColormap(colors)
    if ax is None:
        fig, ax = plt.subplots(1, figsize=(11, 7))
    for k, label in enumerate(labels):
        im = ax.scatter(X[y==label,0], X[y==label,1],  alpha=0.5,label='class '+str(label))

    if model is not None:
        # Plot the seprating function
        plotHyperSurface(ax, bound[0], model, model.b, separatorLabel)
        if model.support is not None:
            ax.scatter(model.support[:,0], model.support[:,1], label='Support', s=80, facecolors='n
            print("Number of support vectors = %d" % (len(model.support)))
        # Plot the margins
        supp_pred = model.predict(model.support)
        supp_neg = model.support[supp_pred==-1]
        supp_pos = model.support[supp_pred==1]

        intercept_neg = -np.max(model.separating_function(supp_neg))    ### compute the intercept f
        intercept_pos = -np.min(model.separating_function(supp_pos))   ### compute the intercept fo
        xx = np.array(bound[0])
        plotHyperSurface(ax, xx, model, intercept_neg , 'Margin -', linestyle='-.', alpha=0.8)
        plotHyperSurface(ax, xx, model, intercept_pos , 'Margin +', linestyle='--', alpha=0.8)

        # Plot points on the wrong side of the margin
        y_pred = model.predict(X)
        wrong_side_points = X[y_pred!=y]# find wrong points
        ax.scatter(wrong_side_points[:,0], wrong_side_points[:,1], label='Beyond the margin', s=80,
              edgecolors='grey', color='grey')
    ax.legend(loc='upper left')
    ax.grid()
    ax.set_xlim(bound[0])
    ax.set_ylim(bound[1])
```

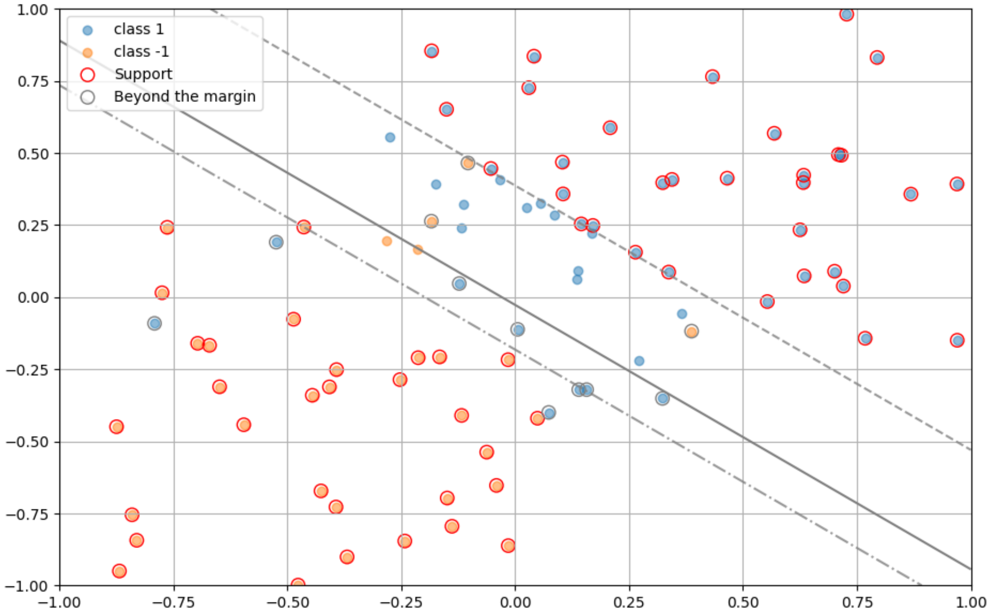
Here are the plots we get from our code in their respective order:

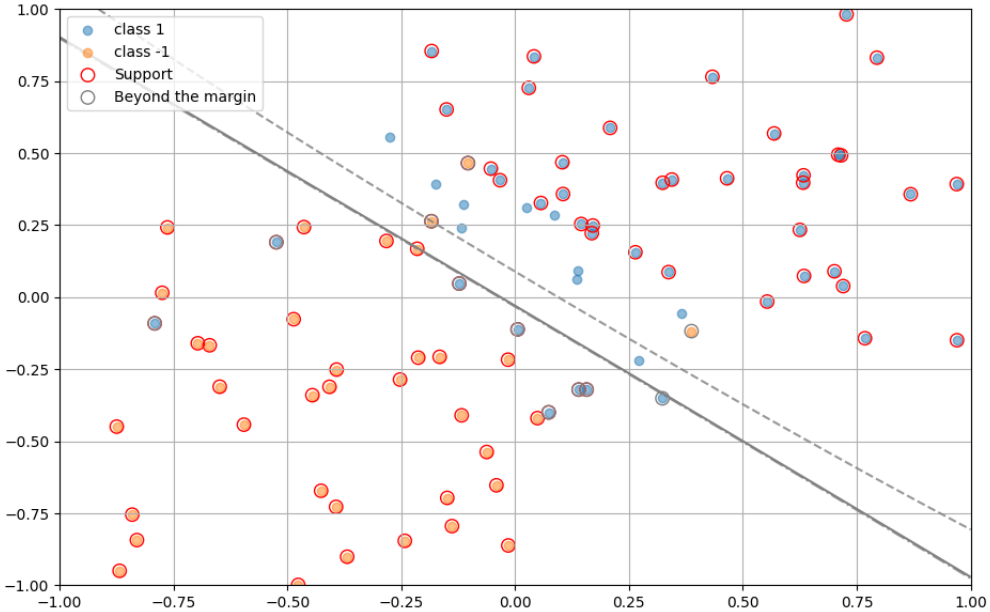


class 1
class -1
Support
Beyond the margin
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

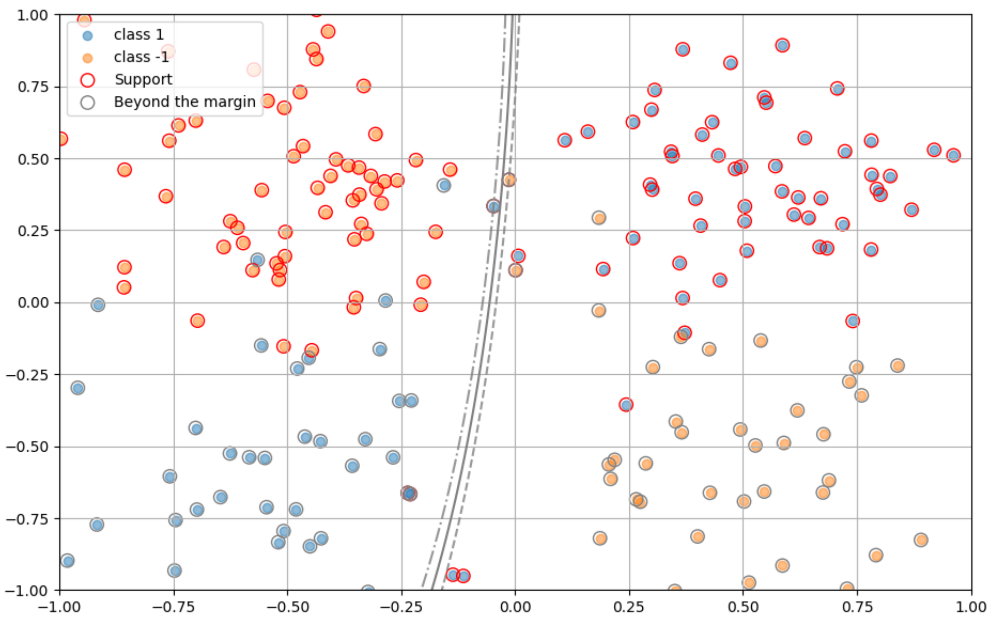


class 1
class -1
Support
Beyond the margin
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


class 1
class -1
Support
Beyond the margin


class 1
class -1
Support
Beyond the margin


class 1
class -1
Support
Beyond the margin
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


class 1
class -1
Support
Beyond the margin
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00