

Exercice 1:

On cherche à écrire le dual de: $\min_{\substack{x \in \mathbb{R}^n \\ s.c}} \|x\|_1$
 $Ax = b.$

On écrit ce problème sous sa forme LP:

$$\begin{array}{ll} \min_{\substack{x \in \mathbb{R}^n \\ Ax = b \\ t \in \mathbb{R}}} & t \\ & \forall i \in \mathbb{R}^n, |x_i| \leq t. \end{array} \Rightarrow \begin{array}{ll} \min_{\substack{x \in \mathbb{R}^n \\ t \in \mathbb{R}}} & t \\ \text{s.c.} & Ax = b \\ & \forall i \in \mathbb{R}^n, x_i \leq t \\ & \forall i \in \mathbb{R}^n, -x_i \leq t \end{array} \quad (\text{LP})$$

On écrit le lagrangien associé à (LP).

$$\begin{aligned} \mathcal{L}(t, x, \mu, \lambda_1, \lambda_2) &= t + \mu^T(Ax - b) + \lambda_1^T(x - t\mathbf{1}_m) \\ &\quad + \lambda_2^T(-t\mathbf{1}_m - x). \\ &= t(1 - \mathbf{1}_m^T \lambda_1 - \mathbf{1}_m^T \lambda_2) - \mu^T b \\ &\quad + x^T(A^T \mu + \lambda_1 - \lambda_2) \end{aligned}$$

Cette fonction est linéaire en les variables t et x et aussi :

$$\min_t t(1 - \mathbf{1}_m^T \lambda_1 - \mathbf{1}_m^T \lambda_2) = \begin{cases} 0 & \text{si } (\lambda_1 + \lambda_2)^T \mathbf{1}_m = 1 \\ -\infty & \text{sinon.} \end{cases}$$

$$\text{et } \min_x x^T(A^T \mu + \lambda_1 - \lambda_2) = \begin{cases} 0 & \text{si } A^T \mu + \lambda_1 - \lambda_2 = 0 \\ -\infty & \text{sinon.} \end{cases}$$

D'où la fonction duale s'écrit :

$$\begin{aligned} g(\mu, \lambda_1, \lambda_2) &= \min_{\substack{x, t}} \mathcal{L}(t, x, \mu, \lambda_1, \lambda_2) \\ &= \begin{cases} 0 & \text{si } (\lambda_1 + \lambda_2)^T \mathbf{1}_m = 1 \\ -\infty & \text{et } A^T \mu + \lambda_1 - \lambda_2 = 0 \text{ sinon.} \end{cases} \end{aligned}$$

le problème dual s'écrit donc :

$$\begin{array}{ll}\max & -\mu^T b \\ \text{s.c} & \lambda_1 \geq 0 \\ & \lambda_2 \geq 0 \\ & A^T \mu + \lambda_1 - \lambda_2 = 0 \\ & \underline{1 - (\lambda_1 + \lambda_2) \mathbf{1}_n = 0}\end{array}$$

2

Exercice 2:

Montrons que $D_{KL}(u, v) \geq 0$, pour $u, v \in \mathbb{R}_{++}^n$ et $\sum_{i=1}^n u_i = \sum_{i=1}^n v_i = 1$.

Vérifions que $D_{KL}(u, v) = f(u) - f(v) - \nabla f(v)^T(u - v)$
avec $f(v) = \sum_{i=1}^n v_i \log(v_i)$.

$$\text{Par définition, } D_{KL}(u, v) = \sum_{i=1}^n u_i \log\left(\frac{u_i}{v_i}\right)$$

$$\text{Or } f(u) - f(v) - \nabla f(v)^T(u - v) = \sum_{i=1}^n (u_i \log(u_i) - v_i \log(v_i))$$

$$- \sum_{i=1}^n (\log(v_i) + 1)(u_i - v_i)$$

$$= \sum_{i=1}^n u_i \log(u_i) - \underbrace{\sum_{i=1}^n u_i \log(v_i)}_{= 0 \text{ car } \sum_i u_i = \sum_i v_i = 1} - \sum_{i=1}^n (u_i - v_i)$$

$$= \sum_{i=1}^n u_i \log\left(\frac{u_i}{v_i}\right).$$

D'où le résultat.

Remarquons que f est strictement convexe car $\nabla^2 f(v) = \begin{pmatrix} \frac{1}{v_1} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{v_n} \end{pmatrix}$
et $v_i \in [1, n]$, $v_i > 0$.

Donc $\forall (u, v) \in \mathbb{R}_{++}^n, \quad f(u) > f(v) + \nabla f(v)^T(u - v)$
 $u \neq v$

Soit $\forall (u, v) \in \mathbb{R}_{++}^n, \quad u \neq v, \quad D_{KL}(u, v) > 0$

Or pour $u = v$, ($u, v \in \mathbb{R}_{++}^n$), $D_{KL}(u, v) = \sum_{i=1}^n u_i \log\left(\frac{u_i}{v_i}\right)$
 $= \sum_{i=1}^n u_i \log(1) = 0$

D'où $\forall u, v \in \mathbb{R}_{++}^n \quad D_{KL}(u, v) \geq 0$ avec égalité si $u = v$.

Exercice 3 :

Considérons le problème : $\min_{\mathbf{x}} \mathbf{c}^T \mathbf{x}$, $\mathbf{A} \in \mathbb{S}_n^{++}$

s.c. $\mathbf{x}^T (\mathbf{A} - \mathbf{b}\mathbf{b}^T) \mathbf{x} \leq 0$

$\mathbf{b}^T \mathbf{x} \geq 0$

$$\mathbf{Dx} = \mathbf{g}.$$

Réécrivons ce problème pour établir la convexité (notamment de l'ensemble des contraintes).

Puisque $\mathbf{A} \in \mathbb{S}_n$, $\exists \mathbf{V} \in \mathbb{S}_n$ tel que $\mathbf{A} = \mathbf{V}^T \mathbf{V}$ (cholesky).

$$\text{Alors } \mathbf{x}^T (\mathbf{A} - \mathbf{b}\mathbf{b}^T) \mathbf{x} = \mathbf{x}^T (\mathbf{V}^T \mathbf{V} - \mathbf{b}\mathbf{b}^T) \mathbf{x}.$$

$$= \|\mathbf{Vx}\|_2^2 - \|\mathbf{b}^T \mathbf{x}\|_2^2 = \|\mathbf{Vx}\|_2^2 - (\mathbf{b}^T \mathbf{x})^2$$

$$\text{car } \mathbf{b}^T \mathbf{x} \in \mathbb{R}.$$

$$\begin{aligned} \mathbf{Dx} \quad & \left\{ \begin{array}{l} \mathbf{x}^T (\mathbf{A} - \mathbf{b}\mathbf{b}^T) \mathbf{x} \leq 0 \\ \mathbf{b}^T \mathbf{x} \geq 0 \end{array} \right. \Leftrightarrow \left\{ \begin{array}{l} \|\mathbf{Vx}\|_2^2 \leq (\mathbf{b}^T \mathbf{x})^2 \\ \mathbf{b}^T \mathbf{x} \geq 0 \end{array} \right. \\ & \Leftrightarrow \|\mathbf{Vx}\|_2^2 \leq \mathbf{b}^T \mathbf{x} \end{aligned}$$

Le problème se réécrit donc :

$$\begin{aligned} \min_{\mathbf{x}} \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.c.} \quad & \|\mathbf{Vx}\|_2 \leq \mathbf{b}^T \mathbf{x} \quad \Leftrightarrow \quad \|\mathbf{y}\|_2 \leq \mathbf{b}^T \mathbf{x} \\ & \mathbf{Dx} = \mathbf{g} \quad \text{et} \quad \mathbf{y} = \mathbf{Vx}. \\ & \mathbf{Dx} = \mathbf{g}. \end{aligned}$$

Ce problème est un SOC et est donc convexe.

Écrivons le dual de ce problème ; le Lagrangien s'écrit :

$$\mathcal{L}(y, x, \mu_1, \mu_2, \lambda) = c^T x + \lambda_1 (\|y\|_2 + b^T y) + \mu_2^T (Dx - g) \\ + \mu_2^T (y - Vx).$$

On a

$$= x^T (c - \lambda_1 b + D^T \mu_2 - V^T \mu_2) \\ + \lambda_1 \|y\|_2 + \mu_2^T y - \mu_2^T g.$$

Or

$$\min_x x^T (c - \lambda_1 b + D^T \mu_2 - V^T \mu_2) = \begin{cases} 0 & \text{si } c - \lambda_1 b + D^T \mu_2 - V^T \mu_2 = 0 \\ -\infty & \text{sinon.} \end{cases}$$

et

$$\min_y \lambda_1 \|y\|_2 + \mu_2^T y = \begin{cases} 0 & \|\mu_2\|_2 \leq \lambda_1 \\ -\infty & \text{sinon.} \end{cases} \quad \text{(résultat du cours)}$$

Donc

$$g(\mu_1, \mu_2, \lambda) = \begin{cases} -\mu_2^T g & \text{si } c - \lambda_1 b + D^T \mu_2 - V^T \mu_2 = 0 \\ -\infty & \text{et } \|\mu_2\|_2 \leq \lambda_1 \\ -\infty & \text{sinon.} \end{cases}$$

Le problème dual s'écrit donc

$$\max_{\mu_1, \mu_2, \lambda} -\mu_2^T g \\ \text{s.t. } \begin{cases} c - \lambda_1 b + D^T \mu_2 - V^T \mu_2 = 0 \\ \|\mu_2\|_2 \leq \lambda_1 \\ \lambda_1 \geq 0 \end{cases}$$

Exercice 4:

Écris le dual de :

$$\text{minimize} \quad - \sum_{i=1}^m \log(b_i - a_i^T x).$$

Posons $\forall i \in [1, m], y_i = b_i - a_i^T x$.

Le problème s'écrit :

$$\begin{array}{ll} \min & - \sum_{i=1}^m \log(y_i) \\ \text{s.c} & \\ & y_i = b_i - a_i^T x \end{array}$$

On écrit le Lagrangien :

$$\begin{aligned} L(y, x, \mu) &= - \sum_{i=1}^m \log(y_i) + \sum_{i=1}^m \mu_i (y_i - b_i + a_i^T x) \\ &= - \sum_{i=1}^m \{\log(y_i) + \mu_i y_i\} - b^T \mu + \sum_{i=1}^m \mu_i a_i^T x. \end{aligned}$$

Or la fonction $f(y) \mapsto \sum_{i=1}^m \mu_i y_i - \log(y_i)$ est convexe pour $\mu \geq 0$, différentiable et coercive donc

la fonction atteint son minimum pour $\frac{\partial f}{\partial y_i} = 0$ i.e. $\mu_i = \frac{1}{y_i}$.

$$\text{Dès } \inf \sum_{i=1}^m -\log(y_i) + \mu_i y_i = \begin{cases} -\infty & \text{si } \mu \neq 0 \\ m + \sum_{i=1}^m \log(\mu_i) & \text{sinon.} \end{cases}$$

$$\text{De plus } \min_x \sum_{i=1}^m \mu_i a_i^T x = \begin{cases} -\infty & \text{sinon.} \end{cases}$$

$$\text{Finalement, } g(\mu) = \begin{cases} -b^T \mu + m + \sum_{i=1}^m \log(\mu_i) & \mu > 0 \\ -\infty & \text{sinon} \end{cases}$$

7

Le dual s'écrit alors,

$$\max_{\mu \geq 0} -b^T \mu + m + \sum_{i=1}^m \log(\mu_i)$$

Exercice 5:

Considérons le problème ; $\min f_0(x) \quad (1)$
s.c $Ax = b$.

et considérons $\phi(x) = f_0(x) + \lambda \|Ax - b\|_2^2$.

Soit \tilde{x} une solution de ϕ , alors $\nabla \phi(\tilde{x}) = 0$
i.e $\nabla f_0(\tilde{x}) + 2\lambda A^T(Ax - b) = 0$

Considérons le problème dual de (1):

Le Lagrangien s'écrit: $\lambda(x, \mu) = f_0(x) + \mu^T(Ax - b)$.

La fonction $x \mapsto \lambda(x, \mu)$ est convexe, différentiable

donc un point μ admissible du dual de (1) doit

vérifier : (car on minimise $x \mapsto \lambda(x, \mu)$).

$$\nabla f_0(x) + \mu A^T = 0 \quad (2)$$

Or $\tilde{\mu} = 2\lambda(A\tilde{x} - b)$ vérifie (2) d'après la condition de
premier ordre sur ϕ et par définition de \tilde{x} .

Alors par dualité faible, soit x^* la solution optimale de (1)

x^* vérifie $Ax^* = b$ et $f_0(x^*) \geq \lambda(\tilde{x}, \tilde{\mu}) \geq \inf_x \lambda(x, \tilde{\mu})$.

D'où $f_0(x^*) \geq f_0(\tilde{x}) + 2\lambda(A\tilde{x} - b)^T(Ax - b)$

Soit $f_0(x^*) \geq f_0(\tilde{x}) + 2\lambda \|A\tilde{x} - b\|^2$.

Exercice 6

On étudie le problème de classification suivant :

$$\begin{aligned} & \underset{w,z}{\text{minimize}} && \frac{1}{2} \|w\|_2^2 + C\mathbf{1}_m^T z \\ & \text{subject to} && y_i(w^T x_i) \leq 1 - z_i, \text{ for } i = 1, \dots, m \\ & && z \geq 0 \end{aligned}$$

On écrit le dual associé à ce problème :

$$\mathcal{L}(w, z, \lambda_1, \lambda_2) = \frac{1}{2} \|w\|_2^2 + C\mathbf{1}_m^T z + \sum_{i=1}^m \lambda_{1i}(1 - z_i - y_i(w^T x_i)) - (\lambda_2)^T z$$

$$\mathcal{L}(w, z, \lambda_1, \lambda_2) = \frac{1}{2} \|w\|_2^2 - \sum_{i=1}^m \lambda_{1i} y_i (w^T x_i) + C\mathbf{1}_m^T z - \lambda_1^T z - \lambda_2^T z + \lambda_1^T \mathbf{1}_m$$

Or la fonction $w \mapsto \frac{1}{2} \|w\|_2^2 - \sum_{i=1}^m \lambda_{1i} y_i (w^T x_i)$ est différentiable, convexe, coercive donc on peut utiliser la condition de premier ordre pour la minimiser :

$$w = \sum_{i=1}^m \lambda_{1i} y_i x_i$$

$$\text{Puis } \inf_w \left\{ \frac{1}{2} \|w\|_2^2 - \sum_{i=1}^m \lambda_{1i} y_i x_i \right\} = -\frac{1}{2} \left\| \sum_{i=1}^m \lambda_{1i} y_i x_i \right\|_2^2$$

$$\text{Ensuite, } \inf_z \left\{ z^T (C\mathbf{1}_m - \lambda_1 - \lambda_2) \right\} = \begin{cases} 0 & \text{if } C\mathbf{1}_m - \lambda_1 - \lambda_2 = 0 \\ -\infty & \text{otherwise.} \end{cases}$$

Donc,

Ensuite,

$$g(\mu) = \inf_{w,z} \mathcal{L}(w, z, \lambda_1, \lambda_2) = \begin{cases} \lambda_1^T \mathbf{1}_m - \frac{1}{2} \left\| \sum_{i=1}^m \lambda_{1i} y_i x_i \right\|_2^2 & \text{if } C\mathbf{1}_m - \lambda_1 - \lambda_2 = 0 \\ -\infty & \text{otherwise.} \end{cases} \text{ Le}$$

problème dual s'écrit donc :

$$\begin{aligned} & \underset{\mu, \lambda_1, \lambda_2}{\text{maximize}} && \lambda_1^T \mathbf{1}_m - \frac{1}{2} \left\| \sum_{i=1}^m \lambda_{1i} y_i x_i \right\|_2^2 \\ & \text{subject to} && C\mathbf{1}_m - \lambda_1 - \lambda_2 = 0 \\ & && \lambda_1 \geq 0 \\ & && \lambda_2 \geq 0 \end{aligned}$$

Cela se réécrit :

$$\begin{aligned} & \underset{\mu, \lambda_1, \lambda_2}{\text{minimize}} && \frac{1}{2} \left\| \sum_{i=1}^m \lambda_{1i} y_i x_i \right\|_2^2 - \lambda^T \mathbf{1}_m \\ & \text{subject to} && C\mathbf{1}_m - \lambda \geq 0 \\ & && \lambda \geq 0 \end{aligned}$$

On peut le réécrire comme pour le précédent DM :

$$\begin{aligned} & \underset{v}{\text{minimize}} && v^T Q v + v^T p \\ & \text{subject to} && A v \preceq b \end{aligned}$$

Avec : $A = \begin{bmatrix} 1_{m,m} \\ -1_{m,m} \end{bmatrix}$ $Q_{i,j} = \frac{1}{2} y_i y_j x_i^T x_j$ $p = -1_m$ $b = \begin{bmatrix} CI_m \\ 0_m \end{bmatrix}$

Pour chaque centering_step, on minimise : $f_t : v \mapsto t(v^T Q v + v^T p) - \mathbf{1}^T \log(-Av + b)$, où le logarithme est appliqué à chaque composante du vecteur $-Av + b$

Nous devons ∇f_t et $\nabla^2 f_t$. On a :

$$\nabla f_t = t(2Qv + p) - \sum_i \frac{1}{-A_i v + b_i} A_i^T = t(2Qv + p) - \sum_i \frac{1}{A_i v - b_i} A_i^T$$

On peut le réécrire avec des opérations vectorielles:

$$\nabla f_t = t(2Qv + p) - A^T \frac{1}{(Av - b)}, \text{ where } \frac{1}{Av - b} = \begin{bmatrix} \frac{1}{A_0 v - b_0} \\ \dots \\ \frac{1}{A_n v - b_n} \end{bmatrix}$$

Et :

$$\nabla^2 f_t = 2tQ + \sum_i \frac{1}{(A_i v - b_i)^2} A_i^T \cdot A_i$$

$$\nabla^2 f_t = 2tQ + A^T \text{diag} \left(\frac{1}{(Av - b)^2} \right) A, \text{ where } \frac{1}{(Av - b)^2} = \begin{bmatrix} \frac{1}{(A_0 v - b_0)^2} \\ \dots \\ \frac{1}{(A_n v - b_n)^2} \end{bmatrix}$$

```
In [1]: import numpy as np
## We create objective function, gradient, and hessian using the above result
def f_t(t, v, Q, p, A, b):
    f_0 = (v.T).dot(Q.dot(v)) + (v.T).dot(p)
    log_ = b - A.dot(v)
    log_barrier = - np.log(log_).sum()
    return np.squeeze(t * f_0 + log_barrier)

def gradient_t(t, v, Q, p, A, b):
    grad_0 = 2*Q.dot(v) + p
    grad_barrier = (A.T).dot(1 / (b - A.dot(v)))
    return t * grad_0 + grad_barrier

def hessian_t(t, v, Q, p, A, b):
    hessian_0 = 2*Q
    diag_ = np.diag(np.squeeze(1 / (b - A.dot(v))**2))
    hessian_barrier = (A.T).dot(diag_).dot(A)
    return t * hessian_0 + hessian_barrier
```

On écrit la centering step, qui est une méthode de Newton pour minimiser f_t à t fixé.

Pour la backtracking line search, on ajoute une condition : A chaque étape $v + tdv$ doit être dans le domaine de f_t (notamment dans le log). Cela s'écrit $b - A(v + tdv) > 0$, si cette condition n'est pas vérifiée, on continue de mettre à jour t en βt

- t commence égal à 1

- tant que $f(v + t\mathbf{d}v) \geq f(v) + \alpha t \nabla f_t(v) \mathbf{d}v$ or $b - A(v + t\mathbf{d}v) \leq 0$

répéter $t = \beta t$

```
In [2]: def backtracking_line_search_t(t, v, dv, Q, p, A, b):
    rate = 1
    while (b - A.dot(v + rate*dv) <= 0).any() or \
           f_t(t, v + rate*dv, Q, p, A, b) > f_t(t, v, Q, p, A, b) + _alpha*rate:
        rate *= _beta
    return rate

def centering_step(t, v0, eps, Q, p, A, b, max_iter = 100):
    v = v0
    v_seq = []
    newton_crit = []
    count = 0
    while True :
        count += 1
        ## Append v
        v_seq.append(v)
        ## Compute hessian and gradient
        hessian_t_v = hessian_t(t, v, Q, p, A, b)
        gradient_t_v = gradient_t(t, v, Q, p, A, b)

        ## Compute inverse of the hessian matrix:
        inv_hess_t_v = np.linalg.solve(hessian_t_v, np.eye(hessian_t_v.shape[0]))
        ## Compute newton step :
        dv = - inv_hess_t_v.dot(gradient_t_v)
        ## Compute lambda :
        lambda_ = np.squeeze( np.dot(gradient_t_v.T, inv_hess_t_v.dot(gradient_t_v)) )
        newton_crit.append(lambda_/2)
        if lambda_ / 2 <= eps or count > max_iter:
            break
        else :
            step = backtracking_line_search_t(t, v, dv, Q, p, A, b)
            v = v + step * dv
    return v_seq, newton_crit
```

On peut écrire la barrier method, qui utilise la centering step à chaque étape t .

```
In [3]: def barr_method(v0, eps_barr, eps_centering, mu, Q, p, A, b):
    t = t_barr
    # Initialise lists
    v_seq_seq = []
    newton_crit_seq = []
    bar_method_crit_seq = []
    v = v0
    # iterate
    while True :
        v_seq, newton_crit = centering_step(t, v, eps_centering, Q, p, A, b)
        v_seq_seq += v_seq
        v_star = v_seq[-1]
        v = v_star
        newton_crit_seq.append(newton_crit)
        bar_method_crit_seq.append(2*n/t)
        if 2*n/t < eps_barr :
            break
        else :
            t = mu*t
    return v_seq_seq, newton_crit_seq, bar_method_crit_seq
```

On génère des points issus de gaussiennes bivariées.

In [4]:

```

from matplotlib.colors import ListedColormap

## Initializing X, lambda_, y, n, d
#n, m = 100, 20
#X = 20*np.random.rand(m, n)
#C = 10
#y = 20*np.random.rand(m, 1)

C = 10
n, m = 2, 200
# first gaussian
mu_1 = np.array([-2,-2])
cov_1 = np.array([[1,0], [0,1]])
X1 = np.random.multivariate_normal(mu_1, cov_1, size = m//2)
y1 = np.ones((m//2, 1))

# second gaussian
mu_2 = np.array([2,2])
cov_2 = np.array([[1,0], [0,1]])
X2 = np.random.multivariate_normal(mu_2, cov_2, size = m//2)
y2 = - np.ones((m//2, 1))

# Creating X, y
X = np.vstack((X1, X2))
y = np.vstack((y1, y2))

## Defining A, Q, p
A = np.concatenate((np.eye(m), -np.eye(m)), axis = 0)
C = 10
b = np.concatenate((C * np.ones((m, 1)), np.zeros((m,1))), axis = 0)

Q = (X*y).dot((X*y).T)*1/2
p = -np.ones((m, 1))
## parameters for backtracking line search :
_alpha = 0.25
_beta = 1/2
## parameters for barr_method :
t_barr = 1

def compute_dual(v, Q, p, A, b):
    return (v.T).dot(Q.dot(v)) + (v.T).dot(p)

```

On garde en mémoire les résultats

Etude en fonction de μ

In [5]:

```

import matplotlib.pyplot as plt
v_seq_stored = dict()
f_seq_stored = dict()
newton_crit_stored = dict()
barr_method_crit_stored = dict()
mus = [2, 5, 10, 20, 50, 100, 150, 200, 300]
for mu in mus:
    v_ini = C*0.5*np.ones((m, 1))
    v_seq, newton_crit, barr_method_crit = barr_method(v0 = v_ini, eps_barr =
    v_seq_stored[mu] = v_seq
    f_seq_stored[mu] = [compute_dual(v_inner, Q, p, A, b) for v_inner in v_se
    newton_crit_stored[mu] = newton_crit
    barr_method_crit_stored[mu] = barr_method_crit

```

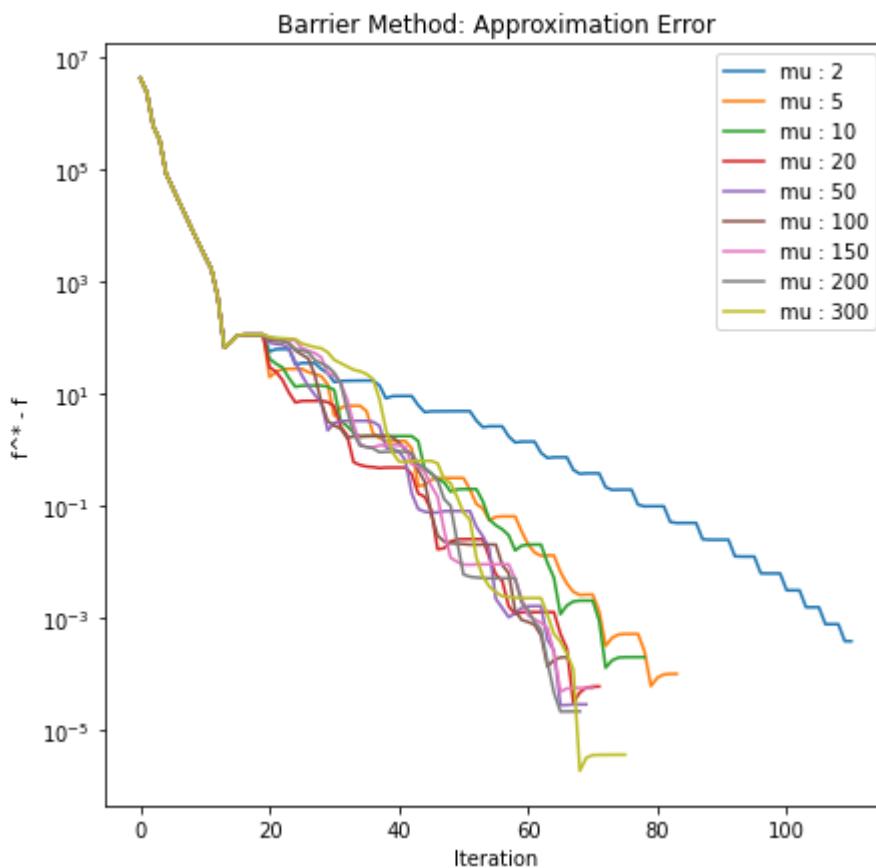
On affiche l'erreur entre la meilleure approximation de f^* et la valeur courante f , en fonction

du nombre total cumulé d'itération de newton (inner).

```
In [6]: fig, ax = plt.subplots(figsize = (7, 7))
f_star = np.min(np.concatenate([f_seq_stored[mu] for mu in mus]))
argmin = np.argmin(np.concatenate([f_seq_stored[mu] for mu in mus]))
v_star = np.concatenate([v_seq_stored[mu] for mu in mus])
for mu in mus :
    delta_star = f_seq_stored[mu] - f_star
    ## Remove where f_star == f_seq (ie where delta_star == 0)
    delta_star = np.array(delta_star)[np.where(delta_star != 0)]
    ax.plot(range(len(delta_star)), delta_star, label = 'mu : %s' % mu)

ax.set(xlabel='Iteration', ylabel='f^* - f',
       title='Barrier Method: Approximation Error')
ax.set_yscale('log')
ax.legend()
ax.plot()
```

Out[6]: []

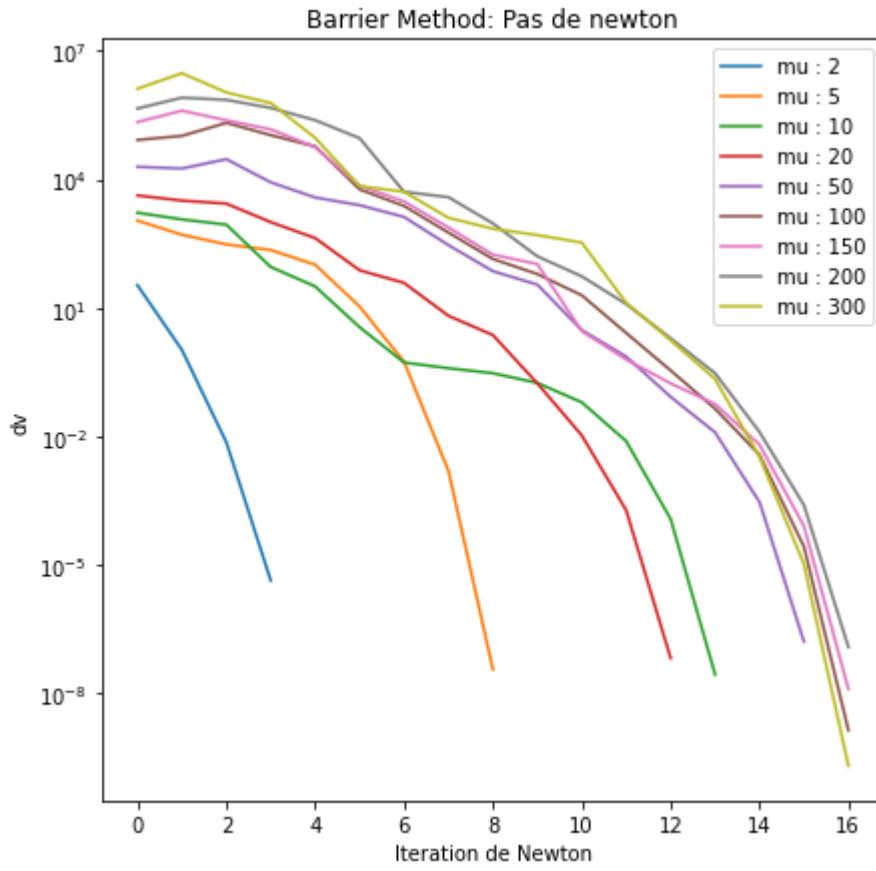


On peut tracer l'évolution du pas à l'intérieur de l'algorithme de Newton, pour un t donné de la barrier method. Ce pas devrait décroître quadratiquement.

```
In [7]: fig, ax = plt.subplots(figsize = (7, 7))
for mu in mus :
    ax.plot(range(len(newton_crit_stored[mu][2])), newton_crit_stored[mu][2],
            label = 'mu : %s' % mu)

ax.set(xlabel='Iteration de Newton', ylabel='dv',
       title='Barrier Method: Pas de newton')
ax.set_yscale('log')
ax.legend()
ax.plot()
```

Out[7]: []



Etude en fonction de C

```
In [8]: import matplotlib.pyplot as plt
v_seq_stored = dict()
f_seq_stored = dict()
newton_crit_stored = dict()
barr_method_crit_stored = dict()
Cs = [1, 5, 10, 15, 50, 100]
for c in Cs:
    print(c)
    b = np.concatenate((c * np.ones((m, 1)), np.zeros((m, 1))), axis = 0)
    v_ini = np.random.rand(m, 1)
    v_seq, newton_crit, barr_method_crit = barr_method(v0 = v_ini, eps_barr =
    v_seq_stored[c] = v_seq
    f_seq_stored[c] = [compute_dual(v_inner, Q, p, A, b) for v_inner in v_seq]
    newton_crit_stored[c] = newton_crit
    barr_method_crit_stored[c] = barr_method_crit
```

1
5
10
15
50
100

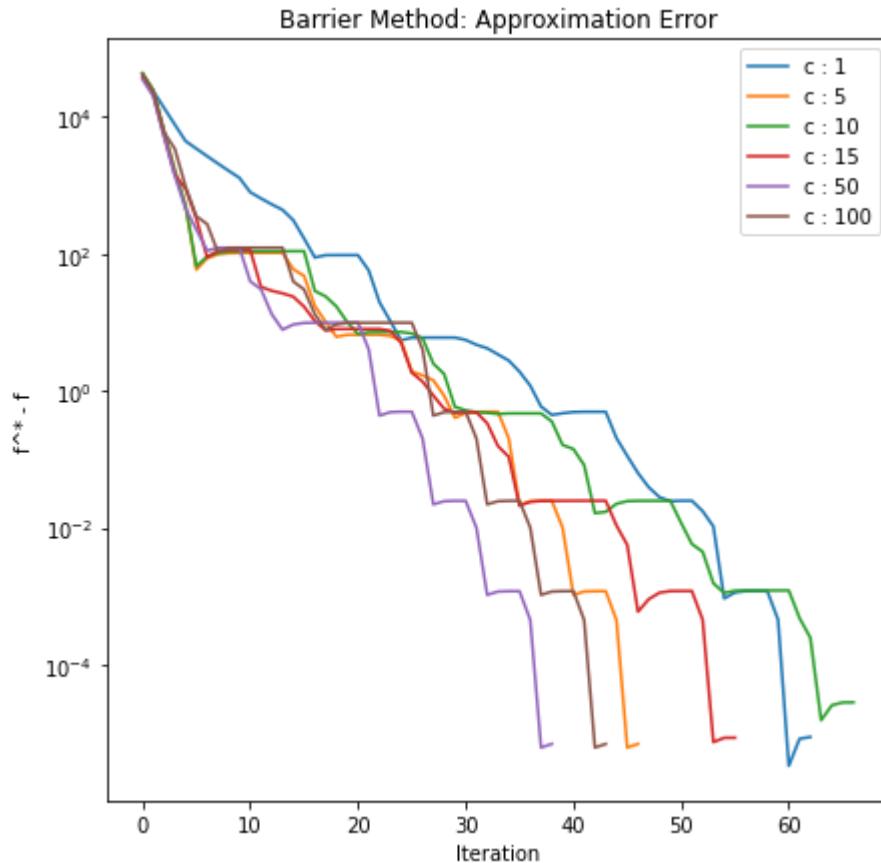
```
In [9]: fig, ax = plt.subplots(figsize = (7, 7))

for c in Cs :
    f_star = np.min([f_seq_stored[c]])
    delta_star = f_seq_stored[c] - f_star
    ## Remove where f_star == f_seq (ie where delta_star == 0)
    delta_star = np.array(delta_star)[np.where(delta_star != 0)]
    ax.plot(range(len(delta_star)), delta_star, label = 'c : %s' % c)

ax.set(xlabel='Iteration', ylabel='f^* - f',
       title='Barrier Method: Approximation Error')
ax.set_yscale('log')
```

```
ax.legend()
ax.plot()
```

Out[9]: []

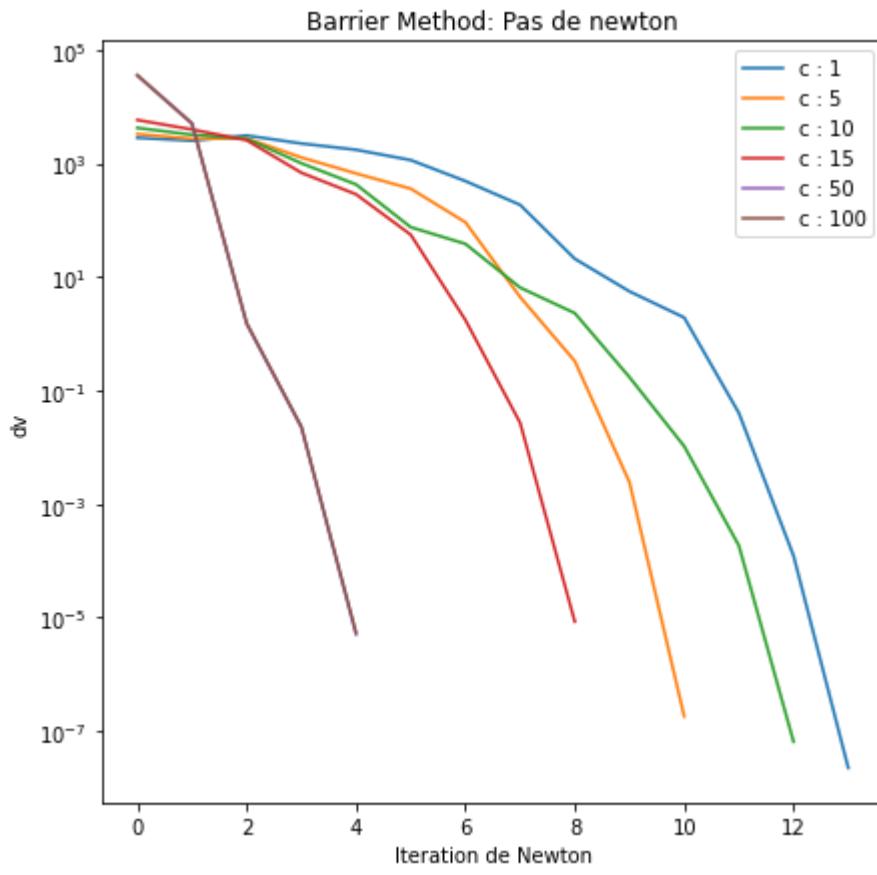


On observe que l'erreur d'approximation diminue plus vite pour des grandes valeurs de C.

```
In [10]: fig, ax = plt.subplots(figsize = (7, 7))
for c in Cs :
    ax.plot(range(len(newton_crit_stored[c][2])), newton_crit_stored[c][2], label=c)

ax.set(xlabel='Iteration de Newton', ylabel='dv',
       title='Barrier Method: Pas de newton')
ax.set_yscale('log')
ax.legend()
ax.plot()
```

Out[10]: []



On retrouve une convergence quadratique du pas de Newton pour n'importe quelle valeur de C .

On cherche à retrouver la solution du primal étant donné la solution du dual.

Pour rappel le primal s'écrit :

$$\begin{aligned} & \underset{w,z}{\text{minimize}} && \frac{1}{2} \|w\|_2^2 + C\mathbf{1}_m^T z \\ & \text{subject to} && y_i(w^T x_i) \leq 1 - z_i, \text{ for } i = 1, \dots, m \\ & && z \geq 0 \end{aligned}$$

Et le dual :

$$\begin{aligned} & \underset{\mu, \lambda_1, \lambda_2}{\text{maximize}} && \lambda_1^T \mathbf{1}_m - \frac{1}{2} \left\| \sum_{i=1}^m \lambda_{1i} y_i x_i \right\|_2^2 \\ & \text{subject to} && C\mathbf{1}_m - \lambda_1 - \lambda_2 = 0 \\ & && \lambda_1 \geq 0 \\ & && \lambda_2 \geq 0 \end{aligned}$$

Le point $w = 0_n$, $z = 2\mathbf{1}_m$ vérifie les contraintes du primal (condition de Slater) donc on peut utiliser KKT.

On trouve pour les conditions de premier ordre :

- $C\mathbf{1}_m - \lambda_1 - \lambda_2 = 0$

- $w = \sum_{i=1}^m \lambda_{1i} y_i x_i$

Et les équations de relâchement supplémentaires :

- $\lambda_{1i}(y_i(w^T x_i) - 1 + z_i) = 0$, for $i = 1, \dots, m$
- $\lambda_{2i}z_i = 0$, for $i = 1, \dots, m$

Ainsi étant donné une estimation de λ_1 optimale par notre algorithme, on peut calculer $w = \sum_{i=1}^m \lambda_{1i} y_i x_i$. Et on peut calculer λ_2 grâce à $C1_m - \lambda_1 - \lambda_2 = 0$

On remarque que les composantes de λ_1 et λ_2 ne peuvent pas être nulles simultanément, puisque $C > 0$.

Ainsi pour remonter à z on effectue la procédure suivante : Pour $i = 1, \dots, m$ on regarde λ_{1i} .

- Si $\lambda_{1i} = 0$, cela veut dire que $\lambda_{2i} \neq 0$ et donc $z_i = 0$
- Si $\lambda_{1i} \neq 0$, alors $\lambda_{2i} = 0$, et donc $z_i = 1 - y_i(w^T x_i)$. Dans le code suivant j'effectue cette procédure avec un paramètre ϵ , une tolérance pour voir si $\lambda_{1i} = 0$.

```
In [11]: def compute_primal_solution_given_dual(lambda_, C = 10, eps_ = 1e-8):
    w_ = (lambda_*y*X).sum(axis = 0)[ :, None]
    mu_ = C*np.ones((m, 1)) - lambda_
    z = np.zeros((m, 1))
    zero_lambda = np.isclose(np.abs(lambda_), 0, atol = eps_, rtol = 0)[ :, 0]
    z[zero_lambda] = 0
    zero_mu = np.isclose(np.abs(mu_), 0, atol = eps_, rtol = 0)[ :, 0]
    z[zero_mu] = (1 - y * X.dot(w_))[zero_mu]
    return w_, z

def compute_primal(w, z, X, y, C):
    return 0.5*np.linalg.norm(w, ord = 2)**2 + C * (z.T).dot(np.ones((m, 1)))

def output_given_w(w, X, coef):
    mini_, maxi_ = X.min(), X.max()
    in_ = np.arange(mini_, maxi_, 0.2)
    out_ = [coef/w[1] - w[0]*x/w[1] for x in in_]
    return in_, out_
```

Comme on l'a vu dans le DM n°2, résoudre ce problème revient à utiliser la règle de classification suivante : $w^T x_i \leq -1 \implies y_i = -1$ et $w^T x_i \geq 1 \implies y_i = 1$, pour le w obtenu. Dans ce qui suit j'affiche les différents hyperplans pour différentes valeurs de C .

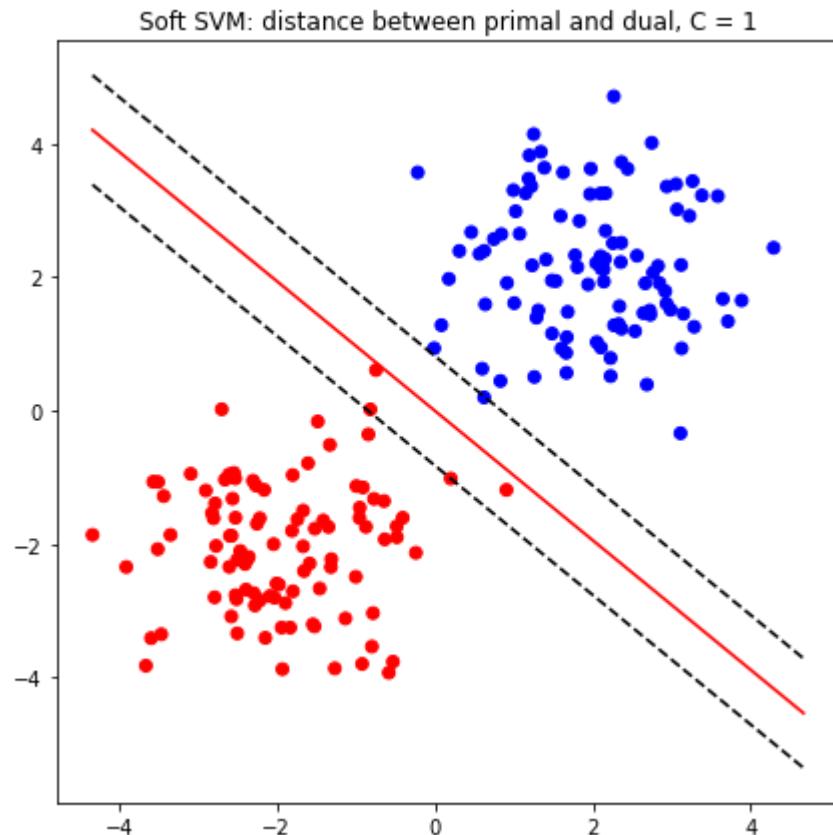
```
In [12]: color = ['blue', 'red']
dual_gap = []
for c in Cs :
    print(c)
    fig, ax = plt.subplots(figsize = (7, 7))
    ax.set(title='Soft SVM: distance between primal and dual, C = %s' % c)
    ax.scatter(X[:, 0], X[:, 1], c = y, cmap = ListedColormap(color))
    f_star = np.min([f_seq_stored[c]])
    argmin = np.argmin([f_seq_stored[c]])
    delta_star = f_seq_stored[c] - f_star
    v_star = v_seq_stored[c][argmin]

    w, z = compute_primal_solution_given_dual(v_star, c, eps_ = 1e-3)
    in_0, out_0 = output_given_w(w, X, 0)
    in_m, out_m = output_given_w(w, X, -1)
    in_p, out_p = output_given_w(w, X, 1)
```

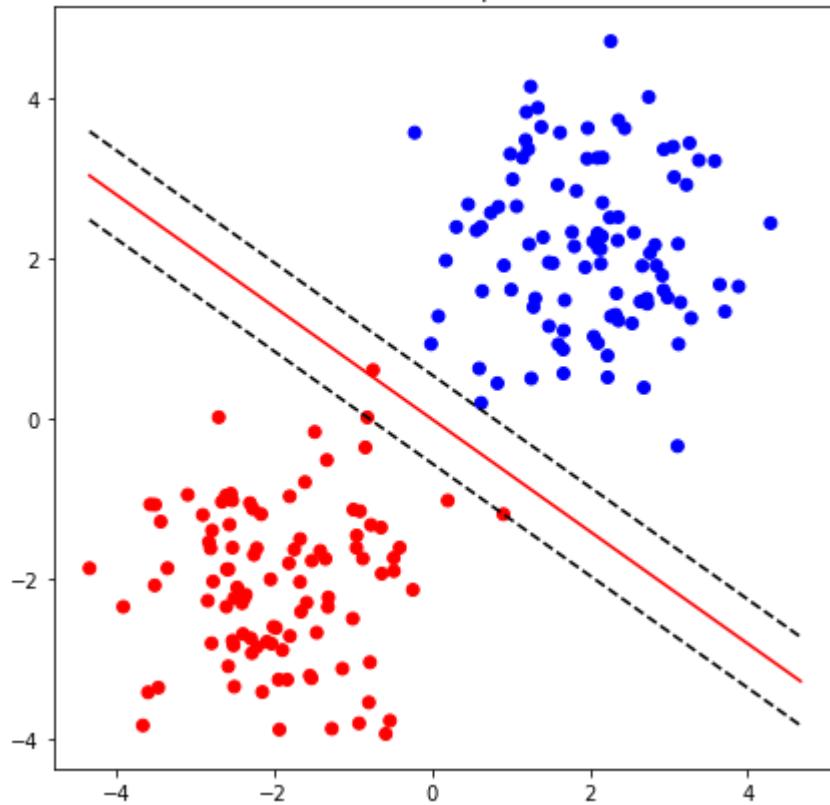
```
l_0 = ax.plot(in_0, out_0, c = 'Red')
l_m = ax.plot(in_m, out_m, '--', c = 'Black')
l_p = ax.plot(in_p, out_p, '--', c = 'Black')

primal = compute_primal(w, z, X = X, y = y, C = c)
dual = compute_dual(v_star, Q = Q, p = p, A = A, b = b)
dual_gap.append(np.squeeze(primal + dual))
```

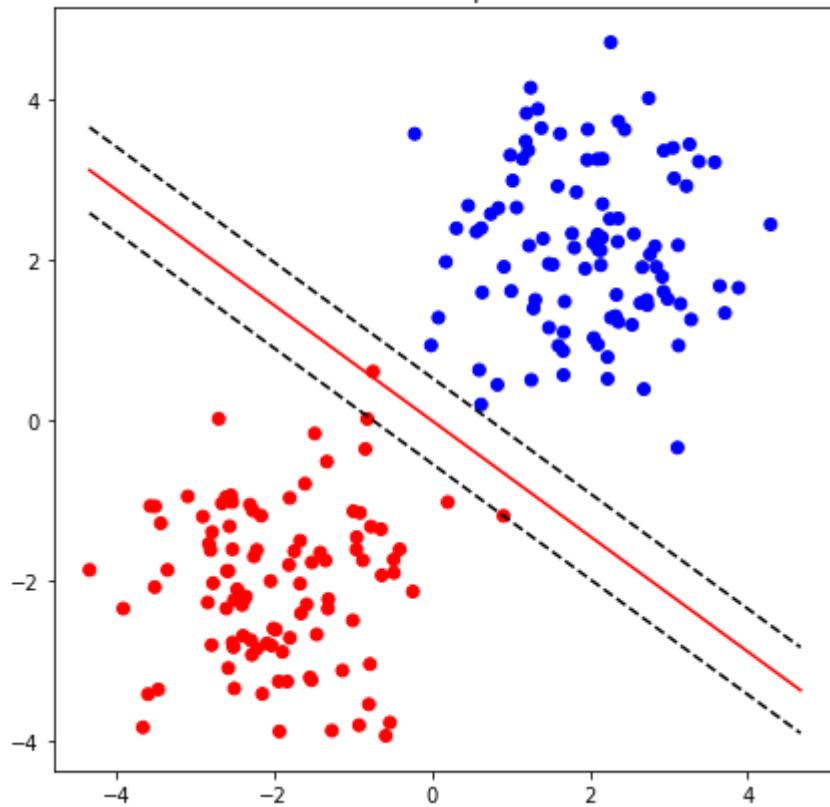
1
5
10
15
50
100



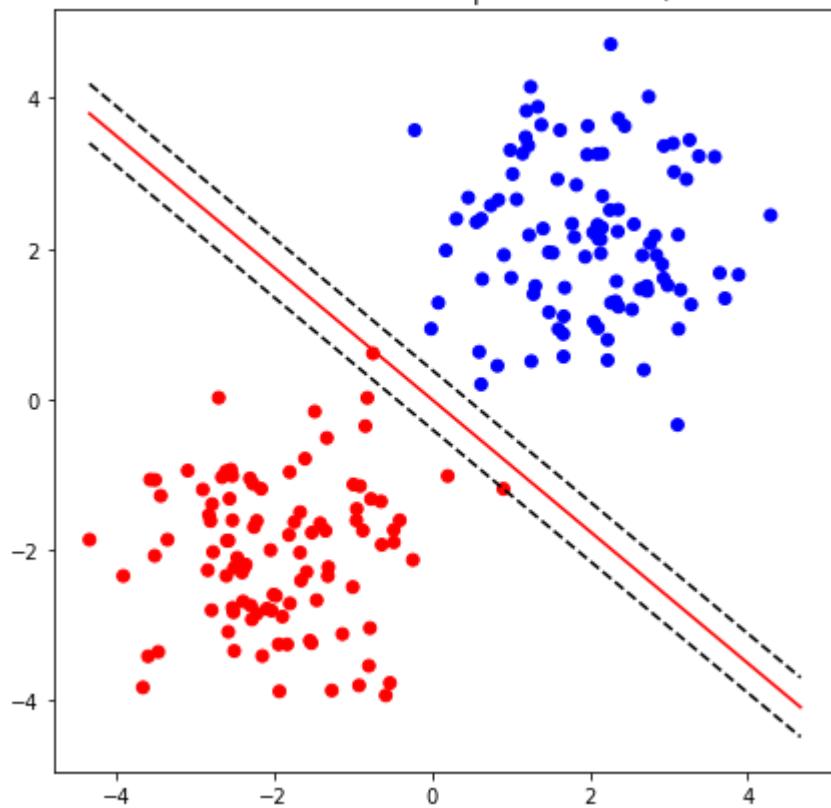
Soft SVM: distance between primal and dual, C = 5



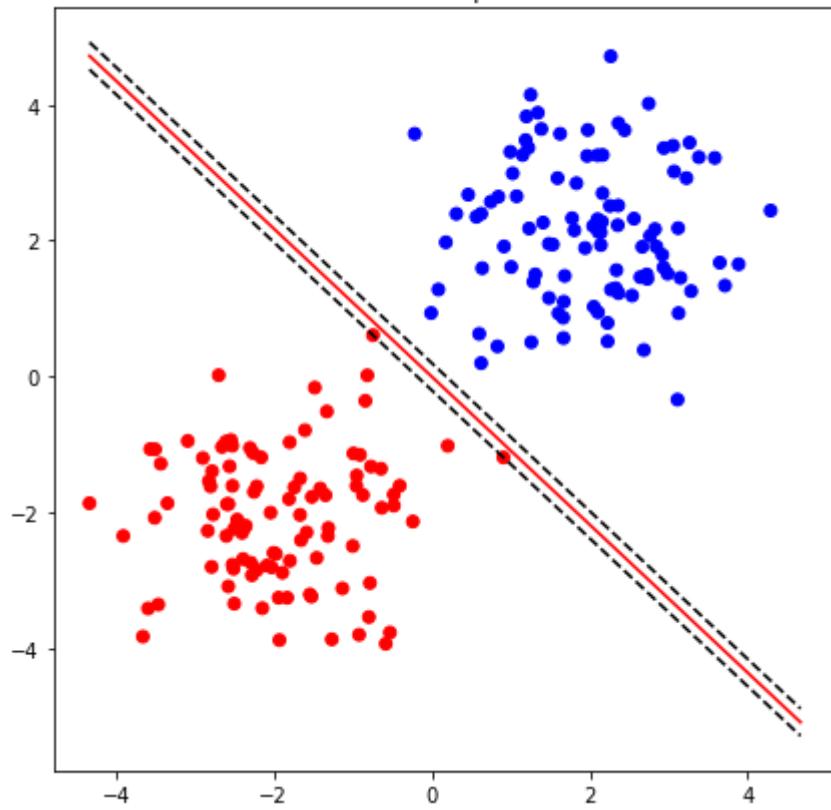
Soft SVM: distance between primal and dual, C = 10

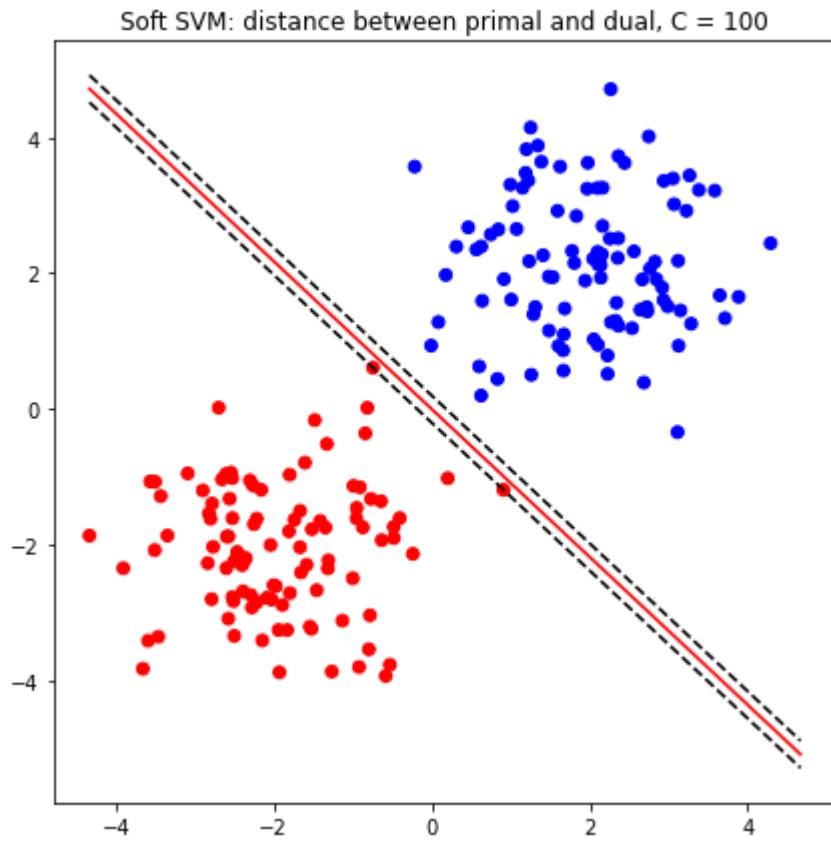


Soft SVM: distance between primal and dual, C = 15



Soft SVM: distance between primal and dual, C = 50





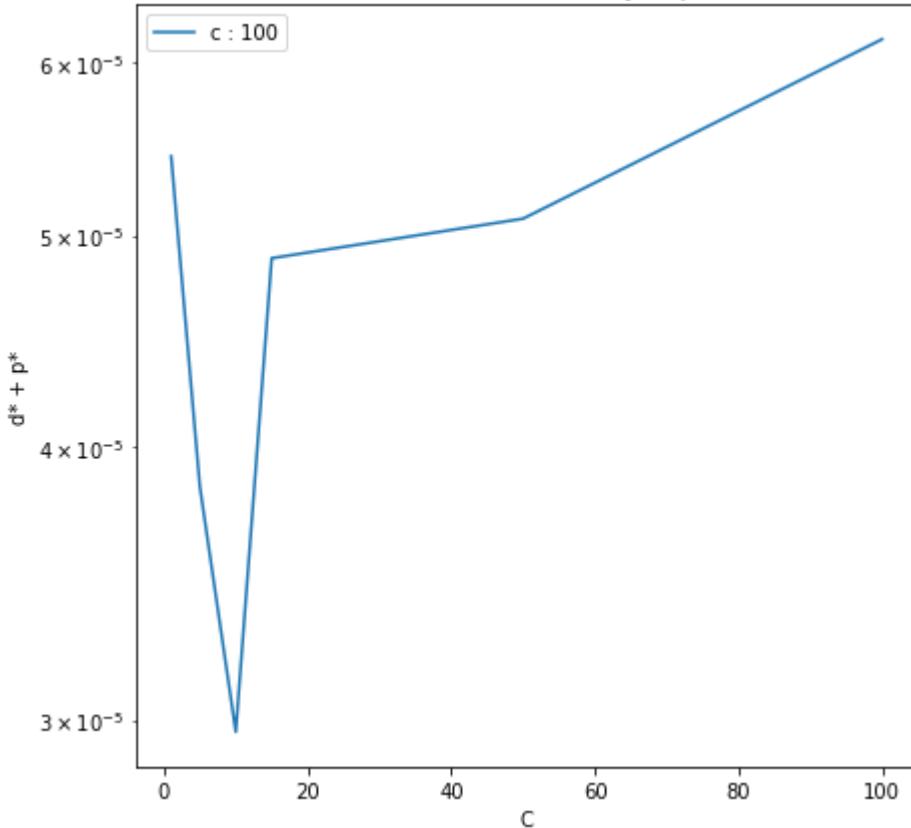
On remarque que plus C est grand, moins on a d'erreurs de classification car la pénalisation est plus importante. Pour C faible on remarque que l'algorithme minimise la distance à l'hyperplan, c'est à dire que les données soient le plus éloignées de l'hyperplan séparateur.

On affiche l'erreur entre la valeur obtenue du dual et du primal en fonction de C :

```
In [13]: fig2, ax2 = plt.subplots(figsize = (7, 7))
ax2.plot(Cs, dual_gap, label = 'c : %s' % c)
ax2.set_xlabel='C', ylabel='d* + p*', title='Barrier Method: Duality Gap')
ax2.set_yscale('log')
ax2.legend()
```

```
Out[13]: <matplotlib.legend.Legend at 0x1179b6160>
```

Barrier Method: Duality Gap



Comme on peut le voir l'écart entre le dual et le primal est très faible et ne semble pas dépendre de C .

Datasets non linéairement séparable

Lorsque le jeu de données n'est pas séparable, il peut être plus dur de séparer les données linéairement. Dans cette partie, j'utilise le kernel trick pour utiliser des séparations non linéaires. Cela revient à remplacer la matrice Q par la matrice $Q_{i,j} = \frac{1}{2}y_i y_j K(x_i, x_j)$. Le cas linéaire correspond à $K(x_i, x_j) = x_i^T \cdot x_j$. Je commence par un kernel linéaire (identique à ce qui est fait plus haut).

```
In [14]: from sklearn.datasets import make_moons, make_circles
X_non, y_non = make_moons(n_samples=200, noise=0.05, random_state=0)
y_non = y_non[:, None]
X_non = X_non - X_non.mean()
A = np.concatenate((np.eye(m), -np.eye(m)), axis = 0)

p = -np.ones((m, 1))
## parameters for backtracking line search :
_alpha = 0.1
_beta = 1/2
## parameters for barr_method :
t_barr = 1
sigma = 0.13

def K_RBF(x_i, x_j):
    #return (x_i.T.dot(x_j))
    return np.exp(- np.linalg.norm(x_i - x_j, ord = 2)**2 / (2* sigma**2))

def K_lin(x_i, x_j):
    return (x_i.T.dot(x_j))

def compute_primal_solution_given_dual_kernel(lambda_, c = 10, eps_ = 1e-8):
```

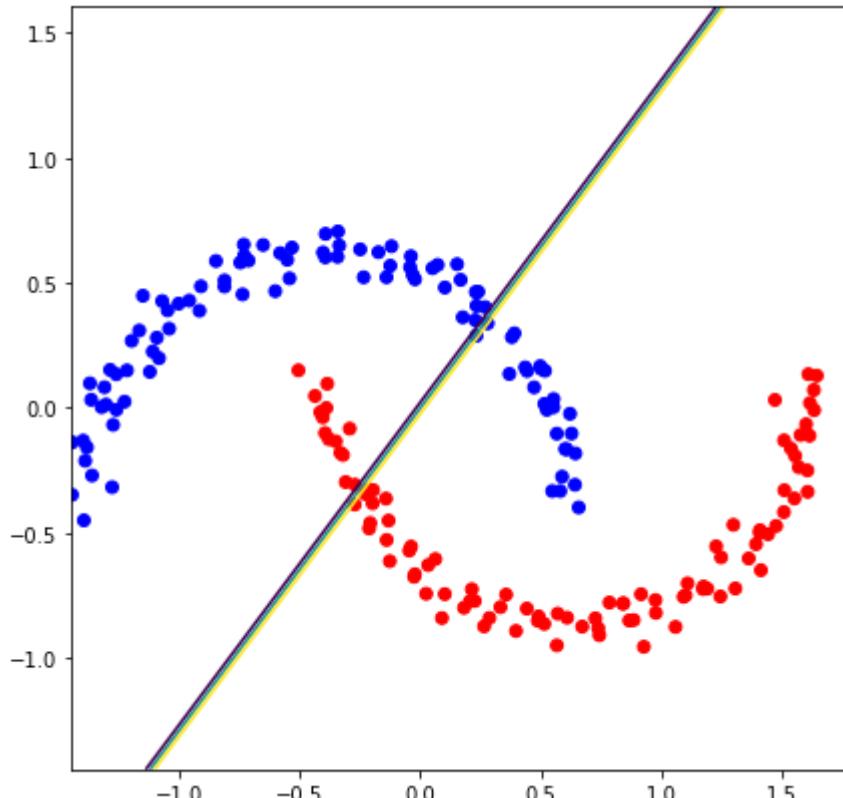
```
w_ = (lambda_*y_non*X_non).sum(axis = 0)[:, None]
return w_

def hyperplane(x1, x2, w, X, y, v, kernel):
    vec_x = np.array([x1, x2])
    sum = 0
    for i in range(len(v)):
        sum += v[i]*y[i]*kernel(vec_x, X[i])
    return sum

def plot_data(kernel, C):
    ## on definit Q
    b = np.concatenate((C * np.ones((m, 1)), np.zeros((m, 1))), axis = 0)
    Q = np.zeros((200, 200))
    for i in range(Q.shape[0]):
        for j in range(Q.shape[0]):
            Q[i, j] = y[i]*y[j]*kernel(X_non[i], X_non[j])
    v_ini = np.random.rand(m, 1)
    v_seq, newton_crit, barr_method_crit = barr_method(v0 = v_ini, eps_barr =
    f_seq = [compute_dual(v_, Q = Q, p = p, A = A, b = b) for v_ in v_seq]
    index_max = np.argmin(f_seq)
    v_star = v_seq[index_max]
    w = compute_primal_solution_given_dual_kernel(v_star, C = C, eps_ = 1e-3)

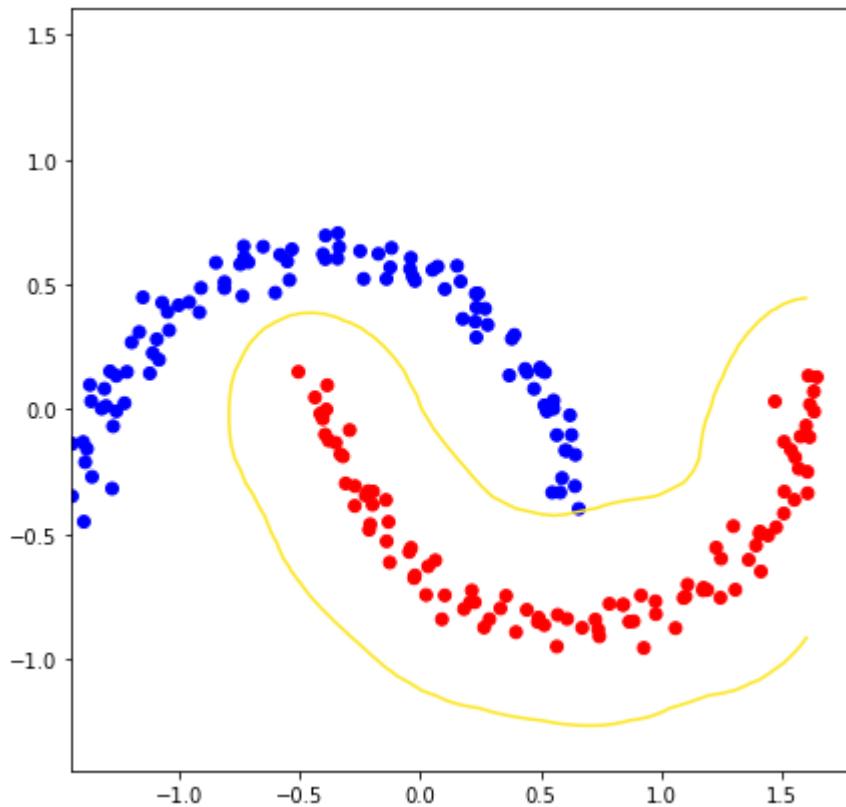
    fig, ax = plt.subplots(figsize = (7, 7))
    ax.scatter(X_non[:, 0], X_non[:, 1], c = y_non, cmap = ListedColormap(['blue', 'red']))
    mini_, maxi_ = X_non.min(), X_non.max()
    x1_ = np.arange(mini_, maxi_, 0.05)
    x2_ = np.arange(mini_, maxi_, 0.05)
    X1, X2 = np.meshgrid(x1_, x2_)
    Z = np.vectorize(lambda x1, x2 : hyperplane(x1, x2, w, X_non, y_non, v =
    ax.contour(X1, X2, Z, levels = [-1, 0, 1]))
```

In [15]: `plot_data(kernel = K_lin, C = 1)`



Comme on peut le voir le kernel linéaire ne parvient pas à séparer le jeux de données.

In [16]: `plot_data(kernel = K_RBF, C = 20)`

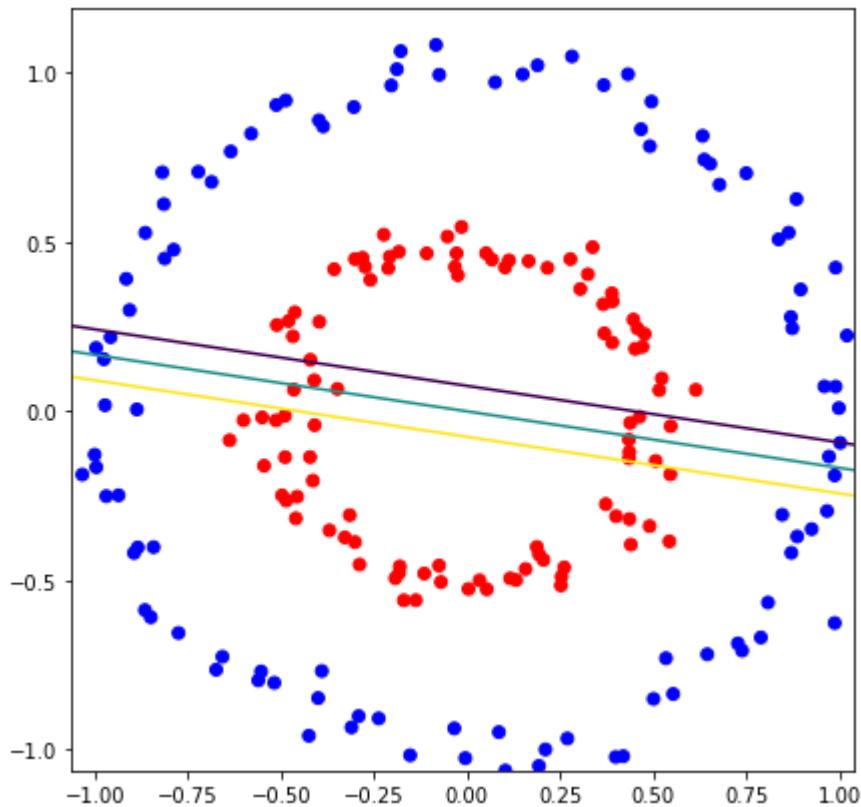


A l'inverse le kernel RBF (radial basis function) définit comme $x, y \mapsto \exp(-\|x - y\|^2/\sigma^2)$ permet de séparer efficacement le jeux de données. La valeur de σ permet de contraindre plus ou moins la frontière.

```
In [17]: x_non, y_non = make_circles(n_samples=200, factor=.5,
                                    noise=.05)
y_non = y_non[:, None]
X_non = X_non - X_non.mean()
A = np.concatenate((np.eye(m), -np.eye(m)), axis = 0)

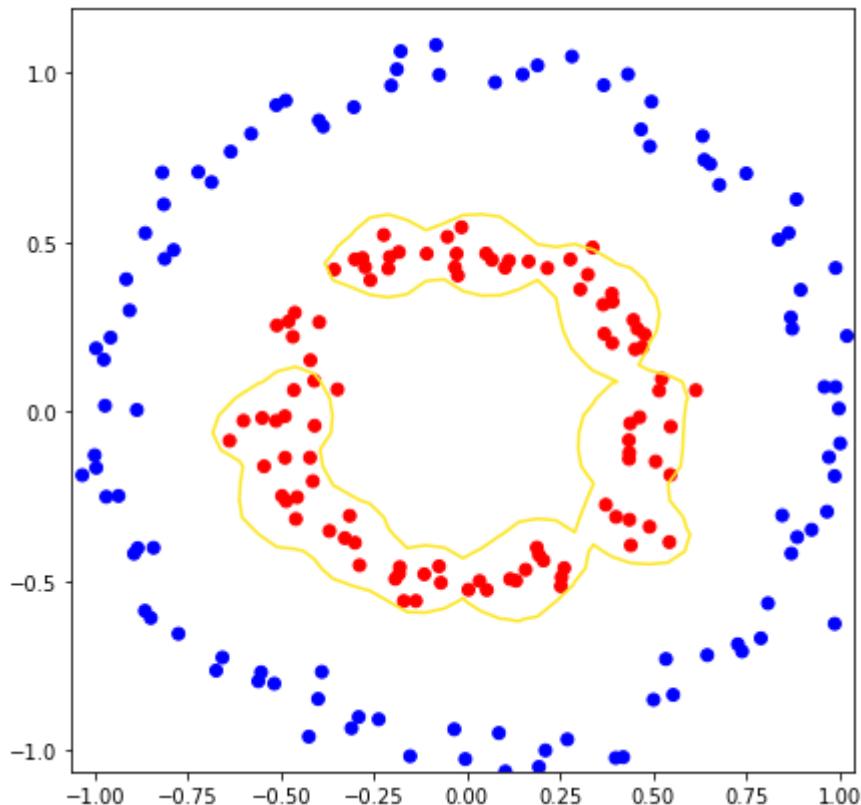
p = -np.ones((m, 1))
## parameters for backtracking line search :
_alpha = 0.1
_beta = 1/2
## parameters for barr_method :
t_barr = 1
sigma = 0.05

plot_data(kernel = K_lin, C = 20)
```



Pour un autre jeu de données, on peut voir l'efficacité du noyau RBF. Cependant, la valeur de sigma change significativement les résultats.

```
In [18]: sigma = 0.05  
plot_data(kernel = K_RBF, C = 10)
```



```
In [19]: sigma = 0.08  
plot_data(kernel = K_RBF, C = 10)
```

