

# Optimisation for Support Vector Machines

## Abstract

This report aims to investigate the performances of different optimization methods for Support Vector Machines(SVM). SVM is an algorithm in the field of supervised learning, used for classification tasks and regression tasks. In this report, a simple synthetic dataset will be generated to establish a nonlinear two-class classification task. Augmented Lagrangian method(AL) and a new optimization method Sequential Minimal Optimisation(SMO) are introduced to solve SVMs.

## 1 Introduction

### Dataset and Task

As shown in Figure 1, a random dataset containing 500 points is generated randomly. The classification task is to separate blue points and green crosses using SVM with suitable optimisation methods.

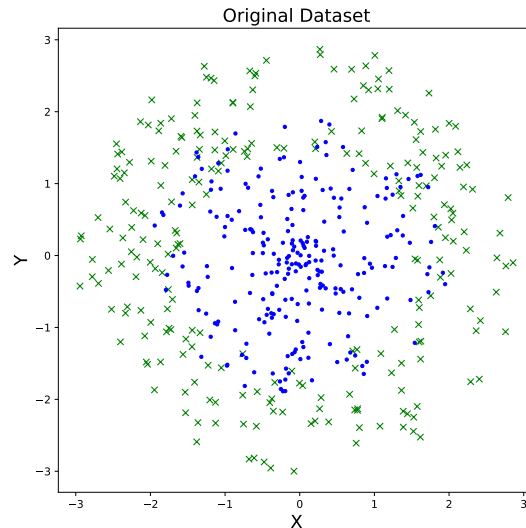


Figure 1: Original Dataset

## Linearly Separable SVM and Lagrange Duality

SVM helps classify data points using a hyperplane in an N-dimensional space. SVM maximizes the margin around the separating hyperplane, which could be considered as a quadratic programming problem.

The training set is

$$S = \{(x_i, y_i)\}_{i=1}^m \in R^n \times \{-1, 1\} \quad (1)$$

First assume  $S$  is linear separable, it can be separated by a line(for 2 dimensions) or hyperplanes(for higher dimensions). The separating hyperplane here is a set  $H_{w,b} = \{x : w^T x + b = 0\}$  (Cortes & Vapnik, 1995).

(Crammer, Singer, Cristianini, Shawe-taylor, & Williamson, 2001) Hence, there exist  $w \in R^n$  and  $b \in R$  such that

$$y_i(w^T x_i + b) > 0, \quad i = 1, \dots, m \quad (2)$$

Inequality (2) is expected to be strictly satisfied.

The distance from a point  $x$  to a hyperplane  $H_{w,b}$  is

$$\rho_x(w, b) := \frac{w^T x + b}{\|w\|} \quad (3)$$

The margin is define as

$$\rho_S(w, b) := \min_{i=1}^m \rho_{x_i}(w, b) \quad (4)$$

The optimal separating hyperplane  $H_{w,b}$  separates the training dataset with a max margin (Herbster, 2021).

The arguments  $w_0$  and  $b_0$  maximize the distance

$$\rho(w_0, b_0) = \min_{x:y=1} \frac{x \cdot w_0}{|w_0|} - \max_{x:y=-1} \frac{x \cdot w_0}{|w_0|} \quad (5)$$

$$= \frac{2}{|w_0|} \quad (6)$$

$$= \frac{2}{\sqrt{w_0^T w_0}} \quad (7)$$

Equation (7) shows that the optimal hyperplane minimizes  $w_0^T w_0$  under the inequality  $y_i(w \cdot x_i + b) \geq 1, i = 1, \dots, m$ . Hence, this can be transferred to a primal quadratic programming problem

$$w^* = \operatorname{argmin}_{w,b} \frac{1}{2} \|w\|^2 \quad (8)$$

subjective to  $y_i(w \cdot x_i + b) \geq 1$  (John, 1998).

Equation (8) minimises a convex function subject to linear inequality constraints. There exists a solution when the necessary and sufficient conditions (KKT) are satisfied.

Using Lagrangian multiplier, Equation (8) is equivalent to the Lagrangian function

$$L(w, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^m \alpha_i [y^{(i)}(w^T x^{(i)} + b) - 1] \quad (9)$$

$$w^* = \max_{\alpha: \alpha_i \geq 0} \min_w L(w, b, \alpha) \quad (10)$$

Find the partial derivatives for  $w$  and  $b$  respectively

$$\frac{\partial}{\partial w} L(w, b, \alpha) = w - \sum_{i=1}^m \alpha_i y^{(i)} x^{(i)} \quad (11)$$

$$\frac{\partial}{\partial b} L(w, b, \alpha) = - \sum_{i=1}^m \alpha_i y^{(i)} \quad (12)$$

Then equation (11) and (12) give

$$w^* = \sum_{i=1}^m \alpha_i y^{(i)} x^{(i)} \quad (13)$$

$$\sum_{i=1}^m \alpha_i y^{(i)} = 0 \quad (14)$$

Then Lagrangian function is expanded as

$$L(w, b, \alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i,j=1}^l y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)} \quad (15)$$

leading to a dual problem and the complexity of this problem depends on  $m$ .

The inner product of vectors can be written as  $\langle x^{(i)}, x^{(j)} \rangle$ , then the dual form is

$$w^* = \max_{\alpha: \alpha_i \geq 0} \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i,j=1}^m y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \quad (16)$$

with constrain  $\sum_{i=1}^m \alpha_i y^{(i)} = 0$ .

## 2 SVM Used for This Task

Since the dataset generated is not linearly separable in Figure 1, formulations mentioned above should be adjusted. First, kernel functions  $K(x^{(i)}, x^{(j)})$  are introduced to map features to higher dimensions instead of inner product  $\langle x^{(i)}, x^{(j)} \rangle$ . Additionally, assume there is an outlier lying in a wrong class, then the dataset will not be linearly separable. To ensure the separating hyperplane can be found under this situation, some points should be allowed to violate the restriction in the model.

This new restriction is the soft margin

$$\min_{w,b} \|w\|^2 + C \sum_{i=1}^m \xi_i \quad (17)$$

such that  $y^i(w^T x^{(i)} + b) \geq 1 - \xi_i, i = 1, \dots, l$  where  $\xi \geq 0, i = 1, \dots, l$

After changing the constraints, the objective function is also adjusted to penalise the outliers.  $C \sum_{i=1}^l \xi_i$  gives larger value of the objective function with more outliers. Here C is the weight of the outlier, which is a constant. A larger C indicates that the outlier has a greater impact on the objective function.

Since  $l_{0/1}$  is not convex and not continuous, the loss for SVM is instead by surrogate loss. The loss function usually used to maximize margin is the hinge loss function (Yan & Li, 2020)

$$c(x, y, f(x)) = (1 - y \cdot f(x))_+ \quad (18)$$

If the predicted value has the same sign with the actual value, the loss is zero; Otherwise, the loss is calculated by  $1 - y \cdot f(x)$ . To balance the loss and margin maximization, a regularization parameter is added to the loss function. Hence, the hinge loss for SVM is

$$\min_w \lambda \|w\|^2 + \sum_{i=1}^l (1 - y_i \langle x_i, w \rangle)_+ \quad (19)$$

The lagrangian function is changed to be

$$L(w, b, \alpha, \xi, r) = \frac{1}{2} \|w\|^2 + C \sum_{i=1}^l \xi_i - \sum_{i=1}^l \alpha_i [y^{(i)}(w^T x^{(i)} + b) - 1 + \xi_i] - \sum_{i=1}^l r_i \xi_i \quad (20)$$

Find the partial derivatives for  $w, b$  and  $r$  respectively

$$\nabla L(w, b, \alpha, \xi, r) = w - \sum_{i=1}^l \alpha_i y^{(i)} x^{(i)} = 0 \quad (21)$$

$$\frac{\partial}{\partial b} L(w, b, \alpha, \xi, r) = \sum_{i=1}^l \alpha_i y^{(i)} = 0 \quad (22)$$

$$\frac{\partial}{\partial r} L(w, b, \alpha, \xi, r) = C - \alpha_i - \xi_i = 0 \quad (23)$$

Then

$$w = \sum_{i=1}^l \alpha_i y^{(i)} x^{(i)} \quad (24)$$

$$C = \alpha_i + \xi_i \quad (25)$$

The dual form of this question is

$$w^* = \max_{\alpha: 0 \leq \alpha_i \leq C} \sum_{i=1}^l \alpha_i - \frac{1}{2} \sum_{i,j=1}^l y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \quad (26)$$

with constrain  $\sum_{i=1}^l \alpha_i y^{(i)} = 0$

To get better performance of the model, kernel function is introduced to solve linearly inseparability since the features will be linearly separable after mapping to higher dimensions.

Replacing  $\langle x^{(i)}, x^{(j)} \rangle$  with  $K(x^{(i)}, x^{(j)})$  and rewriting (24), the equation becomes

$$w^* = \min_{\alpha: 0 \leq \alpha_i \leq C} \frac{1}{2} \sum_{i,j=1}^l y^{(i)} y^{(j)} \alpha_i \alpha_j K(x^{(i)}, x^{(j)}) - \sum_{i=1}^l \alpha_i \quad (27)$$

with linear equality constrain  $\sum_{i=1}^l \alpha_i y^{(i)} = 0$  ([Herbster, 2021](#)).

Table 1: Commonly-used Kernel Functions([wei Hsu et al., 2010](#))

Types	Expressions	Parameters
Linear kernel	$x_i^T x_j$	
Polynomial kernel	$(x_i^T x_j)^d$	$d \geq 1$ is the degree of the polynomial
RBF kernel	$\exp(-\frac{\ x_i - x_j\ ^2}{2\sigma^2})$	$\sigma > 0$
Laplacian kernel	$\exp(-\frac{\ x_i - x_j\ }{\sigma})$	$\sigma > 0$
Sigmoid Kernel	$\tanh(\gamma x_i^T x_j + r)$	$\gamma > 0, r < 0$

### 3 Optimisation Method

The constraint optimisation problem (27) is a quadratic program problem, which could be solved by interior point method, augmented Lagrangian, quadratic penalty and sequential minimal optimization. Augmented Lagrangian (from lecture) and sequential minimal optimization (not on the syllabus) will be discussed below.

#### Augmented Lagrangian(AL)(Betcke, 2022)

$$\begin{aligned}
L_A(x, \nu; \mu) &:= f(x) + \sum_{i=1}^p v_i h_i(x) + \frac{\mu}{2} \sum_{i=1}^p h_i^2(x) \\
&+ \sum_{i=1}^p v_i \max\{f_{1,i}, 0\} + \frac{\mu}{2} \sum_{i=1}^p v_i \max\{f_{1,i}^2, 0\} \\
&+ \sum_{i=1}^p v_i \max\{f_{2,i}, 0\} + \frac{\mu}{2} \sum_{i=1}^p v_i \max\{f_{2,i}^2, 0\} \\
\nabla_x L_A(x_k, \nu^k; \mu_k) &= \nabla f(x_k) + \sum_{i=1}^p [v_i^k + \mu_k h_i(x_k)] \nabla h_i(x_k) \\
\nabla_x L_A(x_k, \nu^k) &= \nabla f(x_k) + \sum_{i=1}^p v_i^* \nabla h_i(x_k), \quad v_i^* \approx v_i^k + \mu_k h_i(x_k) \quad (28) \\
\nabla_{xx}^2 L_A(x_k, \nu^k) &= \nabla^2 f(x_k) + \sum_{i=1}^p [v_i^* \nabla^2 h_i(x_k)]
\end{aligned}$$

Let  $x^*$  be a local minimal where the constraint gradients are linearly independent and satisfies  $2^{nd}$  order sufficient conditions with Lagrange multipliers  $\nu^*$ . Then for all  $\mu \geq \bar{\mu} > 0$ ,  $x^*$  is a strict local minimal of  $L_A(x, \nu^*; \mu)$ .

To guarantee the global convergence of AL, a stopping criteria needs to be used (Yan & Li, 2020).

#### Sequential Minimal Optimisation(SMO)(John, 1998)

SMO can solve SVM quadratic programming problem without extra matrix storage. SMO follows the idea: if KKT conditions are satisfied then the quadratic programming problem is solved; else, solve for two Lagrange multipliers.

SMO avoids the inner iterations and solve quadratic programming problem quickly.

The amount of memory for SMO is linear in the size of training dataset and SMO solves linear SVMs and sparse data sets fastest. The time complexity for one SMO step is  $\Theta(pWn + (1-p)ln)$ , where  $p$  is the probability of the second Lagrange multiplier in the working set,  $W$  is the size of the working set, and  $n$  is the input dimension (Flake & Lawrence, 2002). SMO decomposes the entire large quadratic programming problem into quadratic programming sub-problems. Osuna's theorem says when the quadratic programming sub-problem solved by each iteration contains at least one example violating KKT conditions, the objective function will decrease and the algorithm will eventually lead to global convergence (Osuna, Freund, & Girosi, 1997). The convergence rate is said to be linear when the kernel matrix is positive definite (López & Dorronsoro, 2013).

SMO first chooses two Lagrange multiplier  $\alpha_i$  and  $\alpha_j$ , then fixes other  $\alpha_k$  and optimizes the objective function. Assume  $\alpha_1$  and  $\alpha_2$  are the two chosen multipliers, they lie on a diagonal line due to the linear equality constraints (Stanford, 2009). If target  $y_1 \neq y_2$ ,  $\alpha_2$  has the bounds

$$L = \max(0, \alpha_2^{old} - \alpha_1^{old}), \quad H = \min(C, C + \alpha_2^{old} - \alpha_1^{old}) \quad (29)$$

If target  $y_1 = y_2$ ,  $\alpha_2$  has the bounds

$$L = \max(0, \alpha_2^{old} + \alpha_1^{old} - C), \quad H = \min(C, \alpha_2^{old} + \alpha_1^{old}) \quad (30)$$

After calculating, the update formula for  $\alpha_2$  is

$$\alpha_2^{new} := \alpha_2^{old} - \frac{y_2(E_1 - E_2)}{\eta} \quad (31)$$

where  $E_k = f(x^{(k)}) - y^{(k)}$  is the error for  $k$ th example and  $\eta = 2 < x_1, x_2 > - < x_1, x_1 > - < x_2, x_2 >$ .

Then

$$\alpha_2^{new,clipped} = \begin{cases} H, & \alpha_2^{new} \geq H \\ \alpha_2^{new}, & L < \alpha_2^{new} < H \\ L, & \alpha_2^{new} \leq L \end{cases} \quad (32)$$

$$\alpha_1^{new} = \alpha_1^{old} + y_1 y_2 (\alpha_2^{old} - \alpha_2^{new,clipped}) \quad (33)$$

Finally, the update formula for  $b$  is

$$b := \begin{cases} b_1, & 0 < \alpha_1 < C \\ b_2, & 0 < \alpha_2 < C \\ \frac{b_1+b_2}{2}, & otherwise \end{cases} \quad (34)$$

$$b_1 = E_1 + y_1(\alpha_1^{new} - \alpha_1^{old})k(x_1, x_1) + y_2(\alpha_2^{new,clipped} - \alpha_2^{old})k(x_1, x_2) + b^{old} \quad (35)$$

$$b_2 = E_2 + y_1(\alpha_1^{new} - \alpha_1^{old})k(x_1, x_2) + y_2(\alpha_2^{new,clipped} - \alpha_2^{old})k(x_2, x_2) + b^{old} \quad (36)$$

The error is given by

$$E_k^{new} = E_k^{old} + y_1(\alpha_1^{new} - \alpha_1^{old})k(x_1, x_k) + y_2(\alpha_2^{new,clipped} - \alpha_2^{old})k(x_1, x_2) + b^{old} - b^{new} \quad (37)$$

## 4 Results and Discussion

Figure 2 and 3 below show classification results after applying SMO with kernel function RBF and Laplacian. The tolerance used is  $10^{-6}$ ,  $\sigma$  for kernel is 0.1 and  $C$  is 1.

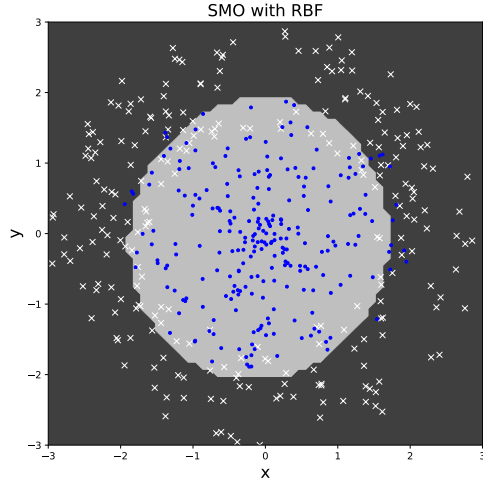


Figure 2: Sequential Minimal Optimisation with RBF kernel

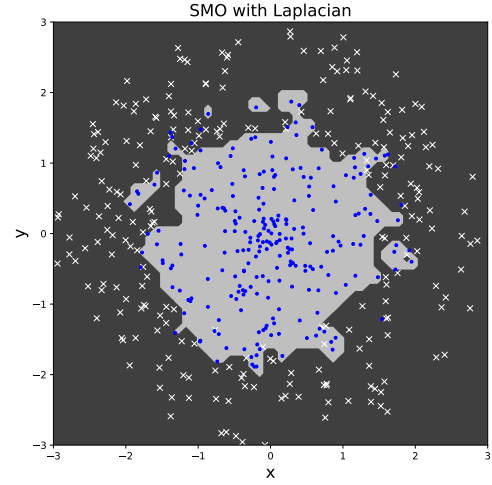


Figure 3: Sequential Minimal Optimisation with Laplacian kernel

Figure 2 and 3 give the resonable results of classfication. RBF gives a circle separating two classes, which could not distinguish the points on the edge of the circle. Laplacian gives a more accurate identification and successfully classified crosses around single blue point.



Plotting the rate of convergence against iterations using the formula  $\frac{\|x_k - x^*\|_2}{\|x_{k-1} - x^*\|_2}$ , the results are shown below.

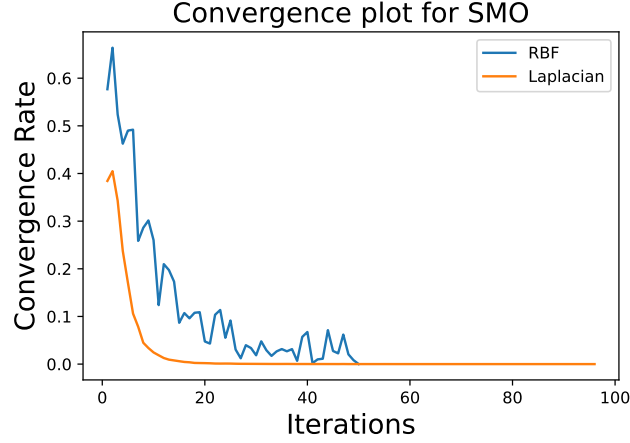


Figure 4: Rate of Convergence for SMO

In Figure 4, the error for SMO for both kernels are decreasing quickly within 20 iterations and eventually converge. The error for SMO using RBF kernel decreases smoothly while the error for SMO using Laplacian kernel decrease zigzag. But both of them show linear trends in first few iterations.

I used *memory\_profiler.memory\_usage()* to calculate the memory difference before and after applying the classification. Also, I used *time* to record the total time each method used. The results are showed in Table 2.

Table 2: **Memory and Time each method used**

Method	Iterations	Total Time(s)	Memory Used
SMO with RBF	50	181.49	2.7
SMO with Laplacian	97	456.76	1.1

Laplacian kernel took more time and iterations to converge than RBF kernel. The classification accuracy using Laplacian is also higher than using RBF.

## References

- Betcke, M. M. (2022). Numerical optimisation slides. *UCL*.
- Cortes, C., & Vapnik, V. (1995). Support-vector networks. In *Machine learning* (pp. 273–297).
- Crammer, K., Singer, Y., Cristianini, N., Shawe-taylor, J., & Williamson, B. (2001). On the algorithmic implementation of multi-class kernel-based vector machines. *Journal of Machine Learning Research*, 265–292.
- Flake, G. W., & Lawrence, S. (2002). Efficient svm regression training with smo. *Machine Learning*, 46(1), 271–290.
- Herbster, M. (2021). Supervised learning slides. *UCL*.
- John, P. (1998). Sequential minimal optimization: A fast algorithm for training support vector machines.
- López, J., & Dorronsoro, J. R. (2013). The convergence rate of linearly separable smo. In *The 2013 international joint conference on neural networks (ijcnn)* (p. 1-7). DOI: 10.1109/IJCNN.2013.6707034
- Osuna, E., Freund, R., & Girosi, F. (1997). An improved training algorithm for support vector machines. In *Neural networks for signal processing vii. proceedings of the 1997 ieee signal processing society workshop* (pp. 276–285).
- Stanford. (2009). Cs229: The simplified smo algorithm slides. <http://cs229.stanford.edu/materials/smo.pdf>.
- wei Hsu, C., chung Chang, C., & jen Lin, C. (2010). *A practical guide to support vector classification*.
- Yan, Y., & Li, Q. (2020). An efficient augmented lagrangian method for support vector machine. *Optimization Methods and Software*, 35(4), 855–883.

# Appendix

Word Count = 1790

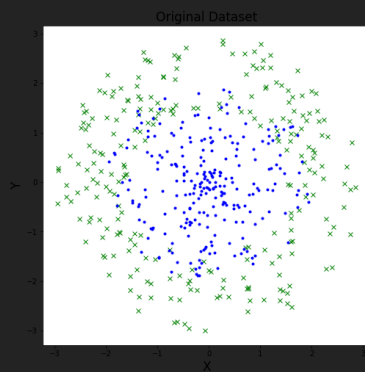
Generate dataset and define kernel function

```
%matplotlib inline
import random
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import memory_profiler as mem
import time

# generate dataset
a1 = np.random.uniform(0,2*np.pi,250)
b1 = np.random.uniform(1.5,3,250)
a2 = np.random.uniform(0,2*np.pi,250)
b2 = np.random.uniform(0,2,250)

x1, y1 = b1*np.cos(a1), b1*np.sin(a1)
x2, y2 = b2*np.cos(a2), b2*np.sin(a2)

plt.figure(figsize=(8,8))
plt.plot(x1, y1, 'gx')
plt.plot(x2, y2, 'b.')
plt.title('Original Dataset', fontsize=17)
plt.xlabel('X', fontsize=17)
plt.ylabel('Y', fontsize=17)
plt.savefig('scatter1.pdf', bbox_inches = 'tight')
plt.show()
```



```
# Kernel functions: RBF and Laplacian
def RBF(x, y, sigma):
    K = np.exp(- sigma * (np.linalg.norm(x - y) ** 2))
    return K

def Laplacian(x, y, sigma):
    K = np.exp(- np.linalg.norm(x - y) / sigma)
    return K
```

## Define SVM and SMO

```
def get_B(alpha):
    s = 0
    w = 0
    b = 0
    for i in range(np.size(alpha)):
        w += alpha[i] * y[i] * x[i]
    for i in range(np.size(alpha)):
        if alpha[i] != 0:
            b += (y[i] * w.dot(x[i].transpose()) - 1) / y[i]
            s += 1
    b += 1/s
    return b

def F(x, y, i, alpha, b, sigma, Kernel):
    fx = 0
    N = np.size(y)
    for j in range(N):
        fx += alpha[j] * y[j] * Kernel(x[j], x[i], sigma)
    return fx + b

def Pred(x, y, p, alpha, b, sigma, Kernel):
    predict = np.zeros((np.size(y), 1))
    for i in range(np.size(y)):
        predict[i] = F(x, y, p[i], alpha, b, sigma, Kernel)
    predict = np.sign(predict)
    predict[predict == 0] = 1
    return predict

#a adapted from https://github.com/hejue/svm_smo and
# https://www.codeproject.com/Articles/1267445/An-Introduction-to-Support-Vector-Machine-SVM-and
def SMO(C, tol, max_iter, x, y, sigma, Kernel):

    def B(Kernel):
        b1 = b - E_i - y[i] * (
            alpha[i] - alpha_prev_i) * Kernel(x[i], x[i], sigma) - y[j] * (
            alpha[j] - alpha_prev_j) * Kernel(x[j], x[j], sigma)
        b2 = b - E_j - y[i] * (
            alpha[i] - alpha_prev_i) * Kernel(x[i], x[i], sigma) - y[j] * (
            alpha[j] - alpha_prev_j) * Kernel(x[j], x[j], sigma)
        return b1, b2

    def Eta(Kernel):
        eta = 2 * Kernel(x[i], x[j], sigma) - Kernel(
            x[i], x[i], sigma) - Kernel(x[j], x[j], sigma)
        return eta

    iteration = 0
    N = np.size(y)
    alpha = np.ones(N)
    alpha_prev = np.ones(N)
    b = 0
    N = np.size(y)
    total_t = 0
    k = 0
    Err = []

    while (iteration < max_iter and np.linalg.norm(alpha - alpha_prev) / np.linalg.norm(alpha_prev) > tol):
        Err.append(np.linalg.norm(alpha - alpha_prev) / np.linalg.norm(alpha_prev))
        alpha_prev = np.array(list(alpha))
        alphas_changed = 0
```

```

for i in range(N):
    E_i = F(x, y, i, alpha, b, sigma, Kernel) - y[i]
    if (y[i] * E_i < -tol and alpha[i] < C) or (y[i] * E_i > tol and alpha[i] > 0):
        j = random.randint(0, N - 1)
        while j == i:
            j = random.randint(0, N - 1)

    E_j = F(x, y, j, alpha, b, sigma, Kernel) - y[j]
    alpha_prev_i = alpha[i]
    alpha_prev_j = alpha[j]

    if y[i] == y[j]:
        L = max([0, alpha[i] + alpha[j] - C])
        H = min([C, alpha[i] + alpha[j]])
    elif y[i] != y[j]:
        L = max([0, alpha[j] - alpha[i]])
        H = min([C, C + alpha[j] - alpha[i]])

    if L != H:
        eta = Eta(Kernel)
        if eta <= 0:
            alpha[j] = alpha[j] - y[j] * (E_i - E_j) / eta
            if alpha[j] > H:
                alpha[j] = H
            elif alpha[j] < L:
                alpha[j] = L

        if abs(alpha[j] - alpha_prev_j) >= tol:
            alpha[i] = alpha[i] + y[i] * y[j] * (alpha_prev_j - alpha[j])
            b1, b2 = B(Kernel)
            b = (b1 + b2) / 2
            if 0 < alpha[i] < C:
                b = b1
            elif 0 < alpha[j] < C:
                b = b2
            alphas_changed += 1

    iteration += 1

print("Total iterations: ", iteration)
return alpha, b, Err

x = np.zeros((500, 2))
x[0:250, 0], x[0:250, 1] = x1, y1
x[250:, 0], x[250:, 1] = x2, y2

y = np.zeros(500)
y[0:250], y[250:] = 1, -1

sigma = 0.1
tol = 1e-6
max_iter = 1000
C = 1

```

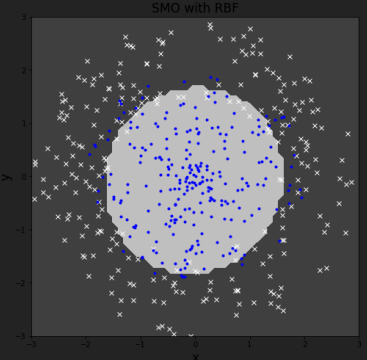
## Plot and record time and memory used

```
time_start = time.time()
print(f'memory before:{mem.memory_usage()}')
alpha_rbf_smo,b_rbf_smo,err_rbf_smo = SMO(C, tol, max_iter, x, y, sigma, RBF)
print(f'memory after:{mem.memory_usage()}')
time_end = time.time()
print('time cost:', time_end - time_start, 's')

memory before:[60.92578125]
Total iterations: 50
memory after:[58.1875]
time cost: 181.4914300441742 s

X = Y = np.linspace(-3, 3, 60)
Z = np.array([Pred(x, y, np.array([i, j]), alpha_rbf_smo, b_rbf_smo, sigma, RBF) for j in Y for i in X])
Z = Z.reshape(60,60)

plt.figure(figsize=(8,8))
plt.plot(x1, y1, 'wx')
plt.plot(x2, y2, 'b.')
plt.contourf(X, Y, Z, 1, cmap= 'gray_r')
plt.xlabel('x', fontsize=17)
plt.ylabel('y', fontsize=17)
plt.title('SMO with RBF', fontsize=17)
plt.savefig('SMO_RBF.pdf', bbox_inches = 'tight')
plt.show()



x = np.zeros((500, 2))
x[0:250, 0], x[0:250, 1] = x1, y1
x[250:, 0], x[250:, 1] = x2, y2

y = np.zeros(500)
y[0:250] = 1
y[250:] = -1

sigma = 0.1
tol = 1e-6
max_iter = 1000
C = 1
```

```

time_start = time.time()
print(f'memory before:{mem.memory_usage()}')
alpha_la_smo, b_la_smo, err_la_smo = SMO(C, tol, max_iter, x, y, sigma, Laplacian)
print(f'memory after:{mem.memory_usage()}')
time_end = time.time()
print('time cost:', time_end - time_start, 's')

```

```

memory before:[44.875]
Total iterations: 97
memory after:[43.7734375]
time cost: 456.7596559524536 s

```

```

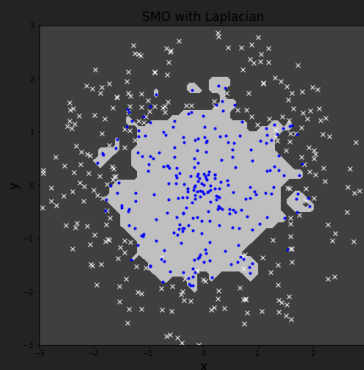
X = Y = np.linspace(-3, 3, 60)
Z = np.array([Pred(x, y, np.array([i, j]), alpha_la_smo, b_la_smo, sigma, Laplacian) for j in Y for i in X])
Z = Z.reshape(60,60)

```

```

plt.figure(figsize=(8,8))
plt.plot(x1, y1, 'wx')
plt.plot(x2, y2, 'b.')
plt.contourf(X, Y, Z, 1, cmap= 'gray_r')
plt.xlabel('x', fontsize=17)
plt.ylabel('y', fontsize=17)
plt.title('SMO with Laplacian', fontsize=17)
plt.savefig('SMO_LA.pdf', bbox_inches = 'tight')
plt.show()

```



```

plt.plot(np.arange(1,len(err_rbf_smo)),err_rbf_smo[1:], label = "RBF")
plt.plot(np.arange(1,len(err_la_smo)),err_la_smo[1:], label = "Laplacian")
plt.legend()
plt.title("Convergence plot for SMO", fontsize=17)
plt.xlabel("Iterations", fontsize=17)
plt.ylabel("Convergence Rate", fontsize=17)
plt.savefig('SMO_error.pdf', bbox_inches = 'tight')
plt.show()

```



Rewrite some matlab code into python. Failed to complete Augmented Lagrangian method in python.

```
# rewrite matlab code into python
def backtracking(f, df, x, p, alpha0, rho, c1=1e-4):
    alpha = alpha0
    alphas = [alpha]
    while f(x + alpha * p) > f(x) + c1 * alpha * np.dot(df(x), p):
        alpha = rho * alpha
        alphas.append(alpha)
    return np.array(alphas)

def steepest_descent(f, df, d2f, x0, tol=1e-6, max_i=1000):
    x = x0
    xs = [x]
    i = 0
    while i < max_i:
        i += 1
        p = -1.0 * df(x)
        alphas = backtracking(f, df, x, p)
        x_1 = x
        x = x_1 + alphas[-1] * p
        xs.append(x)
        if np.linalg.norm(x - x_1) / np.linalg.norm(x_1) < tol:
            break
    return (i, np.array(xs))

def AL(X, y, Kernel, descent_function, mu, lamb, C=1.0):
    L = len(X)
    H = np.zeros((L, L))
    for i in range(L):
        for j in range(L):
            H[i, j] = y[i] * y[j] + Kernel(X[i], X[j])

    f = lambda x: 0.5 * x @ H @ x - np.sum(x) + 0.5 * mu * (y @ x)**2 - lamb * np.sum(x - C)
    df = lambda x: H @ x - np.ones(len(X)) + mu * y * (y @ x) - lamb * np.ones(len(x))
    d2f = lambda x: H + mu * y * y

    iters, xs = descent_function(f, df, d2f, x0=np.full(len(X), 1.0))

    return (iters, xs)

class SVM(object):
    def __init__(self):
        return

    def train(self, X, y, Kernel, AL, descent_method):
        assert len(X) == len(y),
        self.i, self.xs = AL(X, y, Kernel, descent_method)
        self.a = self.xs[-1]
        self.s = X[np.where(self.a > 0.0)]
        self.b = 0.0
        return
```