9.3 - SVM Scratch Using CVXOPT-Copy1

September 24, 2020

0.1 Support Vector Machines

Credits: http://cvxopt.org; https://xavierbourretsicotte.github.io

Let's work on both the hard margin and soft margin SVM algorithm in Python using the well known **CVXOPT** library. While the algorithm in its mathematical form is rather straightfoward, its implementation in matrix form using the CVXOPT API can be challenging at first. This notebook will show the steps required to derive the appropriate vectorized notation as well as the inputs needed for the API.

Notations Scalars are denoted with italic lowercases (e.g., y, b), vectors with bold lowercases (e.g., $\mathbf{w}$, $\mathbf{x}$), and matrices with italic uppercases (e.g., W). $\mathbf{w^T}$ is the transpose of $\mathbf{w}$ and $\|\mathbf{w}\| = \mathbf{w}^T\mathbf{w}$

Let:

- $\mathbf{x}$ be a feature vector (i.e., the input of the SVM). $\mathbf{x} \in \mathbb{R}^n$, where is the dimension of the feature vector.
- y be the class (i.e., the output of the SVM). $y \in \{-1, 1\}$, i.e. the classification task is binary.
- $\mathbf{w}$ and b be the parameters of the SVM: we need to learn them using the training set.
- $(\mathbf{x}^{(i)}, y^{(i)})$ be the th sample in the dataset. Let's assume we have m samples in the training set.

With n = 2, one can represent the SVM's decision boundaries as follows:

The class y is determined as follows:

$$y^{(i)} = \begin{cases} -1 & \text{if } \mathbf{w^T x}^{(i)} + b \leq -1 \\ 1 & \text{if } \mathbf{w^T x}^{(i)} + b \geq 1 \end{cases}$$

which can be more concisely written as

$$y^{(i)}(\mathbf{w^T x}^{(i)} + b) \geq 1$$

Goal The SVM aims at satisfying two requirements:

1. The SVM should maximize the distance between the two decision boundaries. Mathematically, this means we want to maximize the distance between the hyperplane defined by $\mathbf{w^T x} + b = -1$ and the hyperplane defined by $\mathbf{w^T x} + b = 1$. This distance is equal to $\frac{2}{\|\mathbf{w}\|}$. This means we want to solve $\max\limits_{\mathbf{w}} \frac{2}{\|\mathbf{w}\|}$. Equivalently we want $\min\limits_{\mathbf{w}} \frac{\|\mathbf{w}\|}{2}$

2. The SVM should also correctly classify all $\mathbf{x}^{(i)}$, which means

$$y^{(i)}(\mathbf{w}^{\mathbf{T}}\mathbf{x}^{(i)} + b) \geq 1, \forall i \in \{1, \ldots, m\}$$

Primal Problem Recall that our primal, optimization problem is of the form:

$$\min_{w,b} f(w,b) = \min_{w,b} \frac{1}{2}||w||^2$$
$$s.t. \;\; g_i(w,b) = -y^{(i)}(w^T x^{(i)} + b) + 1 = 0$$

Lagrange Method The method of Lagrange multipliers allows us to turn a constrained optimization problem into an unconstrained one of the form:

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

Where $\mathcal{L}(w,b,\alpha)$ is called the Lagrangian and α_i are called the Lagrangian multipliers.

Our primal optimization problem with the Lagrangian becomes the following:

$$\min_{w,b}\left(\max_{\alpha} \mathcal{L}(w,b,\alpha)\right)$$

Dual Problem This is the idea of turning primal problem into dual problem by acknowledging that this is roughly the same:

$$\min_{w,b}\left(\max_{\alpha} \mathcal{L}(w,b,\alpha)\right) = \max_{\alpha}\left(\min_{w,b} \mathcal{L}(w,b,\alpha)\right)$$

This allows us to take the partial derivatives of $\mathcal{L}(w,b,\alpha)$ with respect to w and b , equate to zero and then plug the results back into the original equation of the Lagrangian

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

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w} - \sum \alpha_i \, y_i \, \mathbf{x}_i = 0$$
$$\mathbf{w} = \sum \alpha_i \, y_i \, \mathbf{x}_i \tag{6}$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum \alpha_i y_i = 0,$$

$$\sum \alpha_i \, y_i = 0 \tag{7}$$

Plugging the w back to the original equation, as well as the fact that $-\sum \lambda_i y_i = 0$,, we got that

$$\mathcal{L}(w, b, \alpha) = \sum_i^m \alpha_i - \frac{1}{2} \sum_{i,j}^m y^{(i)} y^{(j)} \alpha_i \alpha_j < x^{(i)} x^{(j)} >$$

hence generating an equivalent dual optimization problem of the form

$$\begin{aligned}
&\max_\alpha \min_{w,b} \mathcal{L}(w, b, \alpha) \\
&\max_\alpha \sum_i^m \alpha_i - \frac{1}{2} \sum_{i,j}^m y^{(i)} y^{(j)} \alpha_i \alpha_j < x^{(i)} x^{(j)} > \\
&s.t.\ \alpha_i \geq 0 \\
&s.t.\ \sum_i^m \alpha_i y^{(i)} = 0
\end{aligned}$$

Why Dual? Short answer: Faster computation + allows to use the kernel trick. Though, there exist some good methods to train SVM in the primal form (See my Assignments 6 - SVM in primal form)

Duality and KTT Karush Kuhn Tucker (KTT) conditions allow us to solve the dual problem instead of the primal one, while ensuring that the optimal solution is the same. In our case the conditions are the following:

- The primal objective and inequality constraint functions must be convex
- The equality constraint function must be affine
- The constraints must be strictly feasible

Then there exists w^*, α^* which are solutions to the primal and dual problems. Moreover, the parameters w^*, α^* satisfy the KTT conditions below:

$$\begin{aligned}
&\frac{\partial}{\partial w_i} \mathcal{L}(w^*, \alpha^*, \beta^*) = 0 && (A) \\
&\frac{\partial}{\partial \beta_i} \mathcal{L}(w^*, \alpha^*, \beta^*) = 0 && (B) \\
&\alpha_i^* g_i(w^*) = 0 && (C) \\
&g_i(w^*) \leq 0 && (D) \\
&\alpha_i^* \geq 0 && (E)
\end{aligned}$$

Moreover, if some w^*, α^* satisfy the KTT solutions then they are also solution to the primal and dual problem.

Equation (C) above is of particular importance and is called the *dual complementarity* condition. It implies that if $\alpha_i^* > 0$ then $g_i(w^*) = 0$ which means that the constraint $g_i(w^*) \leq 0$ is active, i.e., it holds with equality rather than inequality.

Hard margin vs. Soft margin This is the **hard-margin SVM**, as this quadratic optimization problem admits a solution iff the data is linearly separable.

One can relax the constraints by introducing so-called **slack variables** $\xi^{(i)}$. Note that each sample of the training set has its own slack variable. This gives us the **soft-margin SVM** following quadratic optimization problem:

$$\begin{aligned}
\min_{\mathbf{w},b} \quad & \frac{\|\mathbf{w}\|}{2} + C\sum_{i=1}^{m}\xi^{(i)}, \\
s.t. \quad & y^{(i)}(\mathbf{w}^{\mathbf{T}}\mathbf{x}^{(i)} + b) \geq 1 - \xi^{(i)}, \quad \forall i \in \{1,\dots,m\} \\
& \xi^{(i)} \geq 0, \qquad\qquad\qquad\qquad \forall i \in \{1,\dots,m\}
\end{aligned}$$

Thus the Lagrange formulation change as well:

$$\mathcal{L}(w,b,\xi,\alpha,\beta) = \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi^{(i)} - \sum_{i}^{m}\alpha_i[y^{(i)}(w^Tx^{(i)}+b) - 1 - \sum_{i=1}^{m}\beta_i\xi^{(i)}$$

Although this looks tough, terms cancel out nicely.

Following dual form, we need to first minimize w.r.t $\mathbf{w}$, b, and ξ, before maximizing w.r.t each $\alpha_i \geq 0$ and $\beta_i \geq 0$:

$$\frac{\partial\mathcal{L}}{\partial\mathbf{w}} = \mathbf{w} - \sum\alpha_i\, y_i\, \mathbf{x}_i = 0$$

$$\mathbf{w} = \sum\alpha_i\, y_i\, \mathbf{x}_i \tag{6}$$

$$\frac{\partial\mathcal{L}}{\partial b} = -\sum\alpha_i y_i = 0,$$

$$\sum\alpha_i\, y_i = 0 \tag{7}$$

$$\frac{\partial\mathcal{L}}{\partial\xi} = C - \sum\alpha_i - \sum\beta_i = 0$$

Plugging these truths to the Lagrange formula, it happens that we got the same formula, with one addition, i.e., $\alpha_i \leq C$, thus

$$\begin{aligned}
& \max_{\alpha}\min_{w,b,\xi}\mathcal{L}(w,b,\xi,\alpha,\beta) \\
& \max_{\alpha}\sum_{i}^{m}\alpha_i - \frac{1}{2}\sum_{i,j}^{m}y^{(i)}y^{(j)}\alpha_i\alpha_j < x^{(i)}x^{(j)} > \\
& s.t.\ C \geq \alpha_i \geq 0 \\
& s.t.\ \sum_{i}^{m}\alpha_i y^{(i)} = 0
\end{aligned}$$

Kernels One can add even more flexibility by introducing a function ϕ that maps the original feature space to a higher dimensional feature space. This allows non-linear decision boundaries.

Suppose we have a mapping $\varphi : \mathbb{R}^n \to \mathbb{R}^m$, Then the dot product of $\mathbf{x}$ and $\mathbf{y}$ in this space is $\varphi(\mathbf{x})^T\varphi(\mathbf{y})$. A kernel is a function k that corresponds to this dot product, i.e.,

$$k(\mathbf{x},\mathbf{y}) = \varphi(\mathbf{x})^T\varphi(\mathbf{y})$$

Why is this useful?. Kernels give a way to compute dot products in some feature space without even knowing what this space is and what is φ

For example, consider a simple polynomial kernel of degree 2, $k(\mathbf{x},\mathbf{y}) = (1+\mathbf{x}^T\mathbf{y})^2$ (Note that in general polynomial kernel, 2 can be p) with $\mathbf{x},\mathbf{y} \in \mathbb{R}^2$. This doesn't seem to correspond to any mapping function φ. It's just a function that returns a real number. Assuming that $\mathbf{x} = (x_1, x_2)$ and $\mathbf{y} = (y_1, y_2)$, let's expand this expression:

$$\begin{aligned} k(\mathbf{x},\mathbf{y}) &= (1+\mathbf{x}^T\mathbf{y})^2 = (1 + x_1\,y_1 + x_2\,y_2)^2 = \\ &= 1 + x_1^2y_1^2 + x_2^2y_2^2 + 2x_1y_1 + 2x_2y_2 + 2x_1x_2y_1y_2 \end{aligned}$$

Note that this is nothing else but a dot product between two vectors of $(1, x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2)$ and $\varphi(\mathbf{x}) = \varphi(x_1, x_2) = (1, x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2)$. So the kernel $k(\mathbf{x},\mathbf{y}) = (1+\mathbf{x}^T\mathbf{y})^2 = \varphi(\mathbf{x})^T\varphi(\mathbf{y})$ computes a dot product in 6-dimensional space without explicitly visiting this space.

Another example is Gaussian kernel:

$$k(\mathbf{x},\mathbf{y}) = \exp\big(-\gamma\,\|\mathbf{x}-\mathbf{y}\|^2\big)$$

Here is a short proof, ignoring γ, we can use the taylor series expansion, then we get:

$$= \exp\big(-\ \|\mathbf{x}-\mathbf{y}\|^2\big)$$

$$= \exp\big(-\mathbf{x}^2\big)\exp\big(-\mathbf{y}^2\big)\Sigma_{k=0}^{\infty}\frac{2^k(\mathbf{x})^k(\mathbf{y})^k}{k!}$$

The quadratic optimization problem becomes:

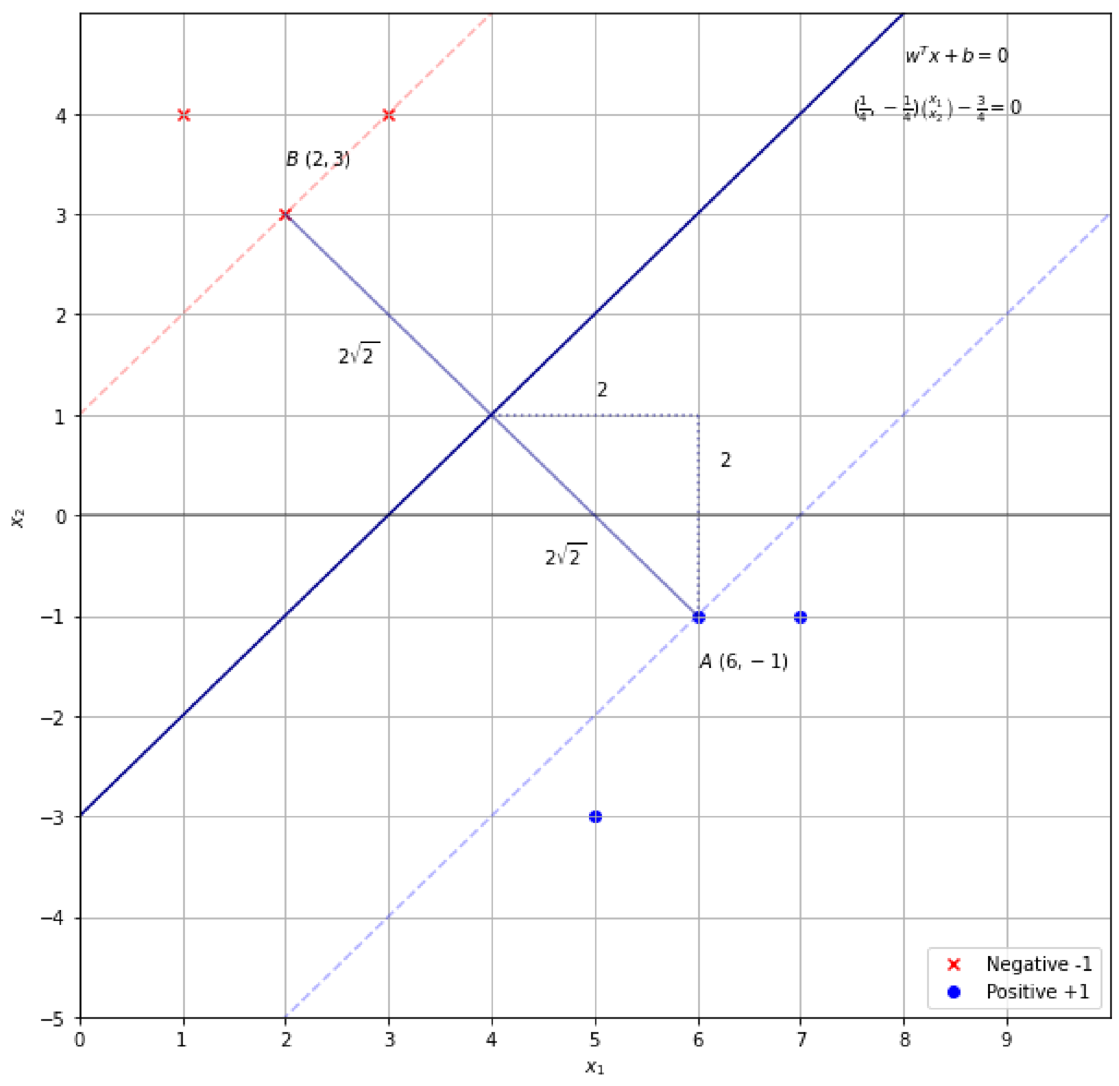
$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{\|\mathbf{w}\|}{2} + C\sum_{i=1}^{m}\xi^{(i)}, \\ s.t. \quad & y^{(i)}(\mathbf{w^T}\phi\left(\mathbf{x}^{(i)}\right) + b) \geq 1 - \xi^{(i)}, \quad \forall i \in \{1,\ldots,m\} \\ & \xi^{(i)} \geq 0, \qquad\qquad\qquad\qquad\quad \forall i \in \{1,\ldots,m\} \end{aligned}$$

Making a prediction Once the $\alpha^{(i)}$ are learned, one can predict the class of a new sample with the feature vector X_test as follows:

$$\begin{aligned} y^{\text{test}} &= \operatorname{sign}\left(\mathbf{w}^{\mathbf{T}}\phi\left(\mathbf{x}^{\text{test}}\right)+b\right) \\ &= \operatorname{sign}\left(\sum_{i=1}^{m}\alpha^{(i)}y^{(i)}\phi\left(x^{(i)}\right)^{T}\phi\left(\mathbf{x}^{\text{test}}\right)+b\right) \end{aligned}$$

0.1.1 Let's start with a simple dataset that is linearly separable

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

#Data set
x_neg = np.array([[3,4],[1,4],[2,3]])
y_neg = np.array([-1,-1,-1])
x_pos = np.array([[6,-1],[7,-1],[5,-3]])
y_pos = np.array([1,1,1])
x1 = np.linspace(-10,10)
x = np.vstack((np.linspace(-10,10),np.linspace(-10,10)))

#Data for the next section
X = np.vstack((x_pos, x_neg))
y = np.concatenate((y_pos,y_neg))

#Parameters guessed by inspection
w = np.array([1,-1]).reshape(-1,1)
b = -3

#Plot
fig = plt.figure(figsize = (10,10))
plt.scatter(x_neg[:,0], x_neg[:,1], marker = 'x', color = 'r', label =␣
 ↪'Negative -1')
plt.scatter(x_pos[:,0], x_pos[:,1], marker = 'o', color = 'b',label = 'Positive␣
 ↪+1')
plt.plot(x1, x1  - 3, color = 'darkblue')
plt.plot(x1, x1  - 7, linestyle = '--', alpha = .3, color = 'b')
plt.plot(x1, x1  + 1, linestyle = '--', alpha = .3, color = 'r')
plt.xlim(0,10)
plt.ylim(-5,5)
plt.xticks(np.arange(0, 10, step=1))
plt.yticks(np.arange(-5, 5, step=1))

#Lines
plt.axvline(0, color = 'black', alpha = .5)
plt.axhline(0,color = 'black', alpha = .5)
plt.plot([2,6],[3,-1], linestyle = '-', color = 'darkblue', alpha = .5 )
plt.plot([4,6],[1,1],[6,6],[1,-1], linestyle = ':', color = 'darkblue', alpha =␣
↪.5 )

#Annotations
plt.annotate(s = '$A \ (6,-1)$', xy = (5,-1), xytext = (6,-1.5))
plt.annotate(s = '$B \ (2,3)$', xy = (2,3), xytext = (2,3.5))#, arrowprops =␣
↪{'width':.2, 'headwidth':8})
plt.annotate(s = '$2$', xy = (5,1.2), xytext = (5,1.2) )
plt.annotate(s = '$2$', xy = (6.2,.5), xytext = (6.2,.5))
plt.annotate(s = '$2\sqrt{2}$', xy = (4.5,-.5), xytext = (4.5,-.5))
plt.annotate(s = '$2\sqrt{2}$', xy = (2.5,1.5), xytext = (2.5,1.5))
plt.annotate(s = '$w^Tx + b = 0$', xy = (8,4.5), xytext = (8,4.5))
plt.annotate(s = '$(\\frac{1}{4},-\\frac{1}{4}) \\binom{x_1}{x_2}- \\frac{3}{4}␣
↪= 0$', xy = (7.5,4), xytext = (7.5,4))

#Labels and show
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.legend(loc = 'lower right')
plt.grid()
```



0.1.2 1. Implementing the SVM algorithm (Hard Margin)

Since we will solve this optimization problem using the **CVXOPT** library in python, we will need to match the solver's API which, according to the documentation is of the form:

$$\begin{aligned} &\min \frac{1}{2} x^T P x + q^T x \\ s.t.\ \ &Gx \leq h \\ &Ax = b \end{aligned}$$

Recall that the dual problem is expressed as:

$$\max_{\alpha} \sum_{i}^{m} \alpha_i - \frac{1}{2} \sum_{i,j}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j < x^{(i)} x^{(j)} >$$

Let $\mathbf{H}$ be a matrix such that

$$H_{ij} = y^{(i)} y^{(j)} < x^{(i)} x^{(j)} >$$

Then the optimization becomes:

$$\begin{aligned} &\max_{\alpha} \sum_{i}^{m} \alpha_i - \frac{1}{2} \alpha^T \mathbf{H} \alpha \\ s.t.\ &\alpha_i \geq 0 \\ &\sum_{i}^{m} \alpha_i y^{(i)} = 0 \end{aligned}$$

We convert the sums into vector form and multiply both the objective and the constraint by -1 which turns this into a minimization problem and reverses the inequality

$$\begin{aligned} &\min_{\alpha} \frac{1}{2} \alpha^T \mathbf{H} \alpha - 1^T \alpha \\ &s.t.\ -\alpha_i \leq 0 \\ &s.t.\ y^T \alpha = 0 \end{aligned}$$

We have that strange 1.T just for the sake of matching with CVXOPT solver API.

We are now ready to convert our numpy arrays into the cvxopt format, using the same notations as in the documentation.

- $\mathbf{P} = \mathbf{H}$ a matrix of size (m, m)
- $\mathbf{q} = -\mathbf{1}$ a matrix of size $(m, 1)$
- $\mathbf{G} = -\mathbf{diag}[\mathbf{1}]$ a diagonal matrix of $-1s$ size (m, m)
- $\mathbf{h} = \mathbf{0}$ a vector of size $(m, 1)$
- $\mathbf{A} = \mathbf{y}$ a vector of size $(m, 1)$
- $\mathbf{b} = \mathbf{0}$ a scalar$

Note that in the simple example of $m = 2$ the matrix $\mathbf{G}$ and vector $\mathbf{h}$ which define the constraint are

$$G = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$h = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

0.1.3 Computing the matrix H in vectorized form

Consider the simple example with 2 input samples $\{x^{(1)}, x^{(2)}\} \in \mathbb{R}^2$ which are two dimensional vectors

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \end{bmatrix} \quad y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \end{bmatrix}$$

We now proceed to create a new matrix X' where each input sample x is multiplied by the corresponding output label y. This can be done easily in Numpy using vectorization and padding.

$$X' = \begin{bmatrix} x_1^{(1)}y^{(1)} & x_2^{(1)}y^{(1)} \\ x_1^{(2)}y^{(2)} & x_2^{(2)}y^{(2)} \end{bmatrix}$$

Finally we take the **matrix multiplication** of X' and its transpose giving $H = X'X'.T$

$$H = X'@X'^T = \begin{bmatrix} x_1^{(1)}y^{(1)} & x_2^{(1)}y^{(1)} \\ x_1^{(2)}y^{(2)} & x_2^{(2)}y^{(2)} \end{bmatrix} \begin{bmatrix} x_1^{(1)}y^{(1)} & x_1^{(2)}y^{(2)} \\ x_2^{(1)}y^{(1)} & x_2^{(2)}y^{(2)} \end{bmatrix}$$

[2]:
```python
#Importing with custom names to avoid issues with numpy / sympy matrix
from cvxopt import matrix as cvxopt_matrix
from cvxopt import solvers as cvxopt_solvers

#Initializing values and computing H. Note the 1. to force to float type
m,n = X.shape
y = y.reshape(-1,1) * 1.
X_dash = y * X
H = np.dot(X_dash , X_dash.T) * 1.

#Converting into cvxopt format
P = cvxopt_matrix(H)
q = cvxopt_matrix(-np.ones((m, 1)))
G = cvxopt_matrix(-np.eye(m))
h = cvxopt_matrix(np.zeros(m))
A = cvxopt_matrix(y.reshape(1, -1))
b = cvxopt_matrix(np.zeros(1))

#Setting solver parameters (change default to decrease tolerance)
cvxopt_solvers.options['show_progress'] = False
cvxopt_solvers.options['abstol'] = 1e-10
cvxopt_solvers.options['reltol'] = 1e-10
cvxopt_solvers.options['feastol'] = 1e-10

#Run solver
sol = cvxopt_solvers.qp(P, q, G, h, A, b)
alphas = np.array(sol['x'])
```

Compute w and b parameters

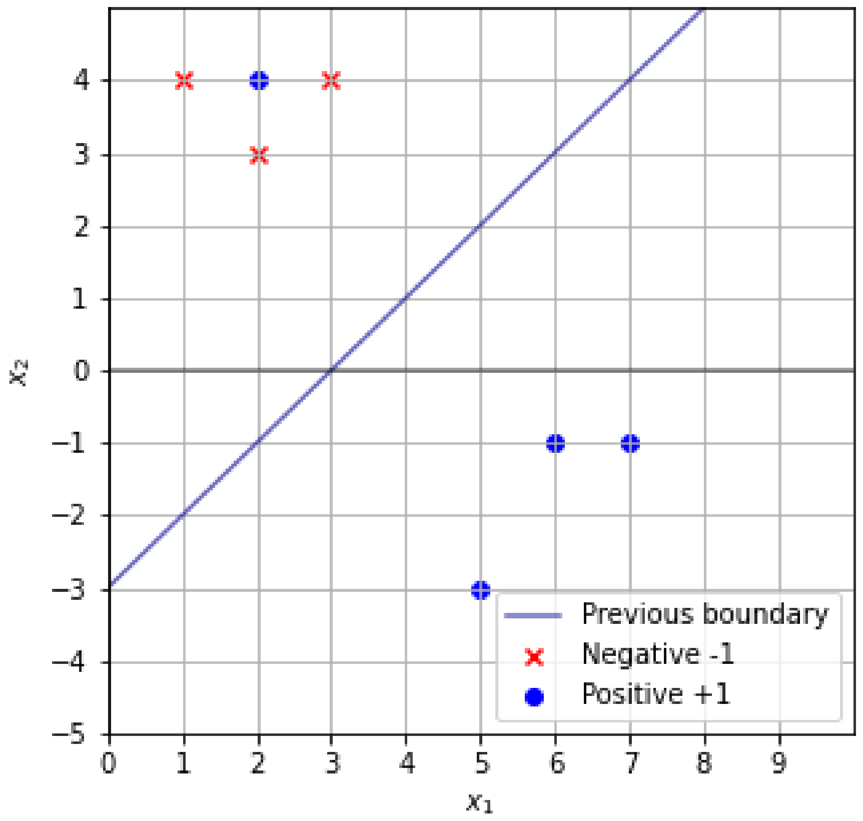
[3]:
```
#w parameter in vectorized form
w = ((y * alphas).T @ X).reshape(-1,1)

#Selecting the set of indices S corresponding to non zero parameters
S = (alphas > 1e-4).flatten()

#Computing b
b = y[S] - np.dot(X[S], w)

#Display results
print('Alphas = ',alphas[alphas > 1e-4])
print('w = ', w.flatten())
print('b = ', b[0])
```

```
Alphas =  [0.0625     0.06249356]
w =  [ 0.24999356 -0.25000644]
b =  [-0.74996781]
```

0.1.4 2. Implementing the SVM algorithm (Soft Margin)

To make the data no longer linearly separable, we shall add a positive point in the middle of the negative cluster:

[4]:
```
x_neg = np.array([[3,4],[1,4],[2,3]])
y_neg = np.array([-1,-1,-1])
x_pos = np.array([[6,-1],[7,-1],[5,-3],[2,4]])
y_pos = np.array([1,1,1,1])
x1 = np.linspace(-10,10)
x = np.vstack((np.linspace(-10,10),np.linspace(-10,10)))



fig = plt.figure(figsize = (5,5))
plt.scatter(x_neg[:,0], x_neg[:,1], marker = 'x', color = 'r', label =␣
 ↪'Negative -1')
plt.scatter(x_pos[:,0], x_pos[:,1], marker = 'o', color = 'b',label = 'Positive␣
 ↪+1')
plt.plot(x1, x1  - 3, color = 'darkblue', alpha = .6, label = 'Previous␣
 ↪boundary')
plt.xlim(0,10)
plt.ylim(-5,5)
plt.xticks(np.arange(0, 10, step=1))
plt.yticks(np.arange(-5, 5, step=1))

#Lines
plt.axvline(0, color = 'black', alpha = .5)
plt.axhline(0,color = 'black', alpha = .5)
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

plt.legend(loc = 'lower right')
plt.grid()

#New dataset (for later)
X = np.array([[3,4],[1,4],[2,3],[6,-1],[7,-1],[5,-3],[2,4]] )
y = np.array([-1,-1, -1, 1, 1 , 1, 1 ])
```



For the softmax margin SVM, the optimization problem can be slightly revised and be expressed as

$$\max_{\alpha} \sum_{i}^{m} \alpha_i - \frac{1}{2}\alpha^T \mathbf{H}\alpha$$

$$s.t. \quad 0 \leq \alpha_i \leq C$$

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

which can be written in standard form as

$$\min_{\alpha} \frac{1}{2}\alpha^T \mathbf{H}\alpha - 1^T \alpha$$

$$s.t. \; -\alpha_i \leq 0$$

$$\alpha_i \leq C$$

$$y^T \alpha = 0$$

We translate this new constraint into standard form by concatenating below matrix **G** a diagonal matrix of 1s of size (m, m). Similarly for the vector **h** to which the value of **C** is added m times.

Note that in the simple example of $m = 2$ the matrix **G** and vector h which define the constraint are:

$$G = \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$h = \begin{bmatrix} 0 \\ 0 \\ C \\ C \end{bmatrix}$$

[5]:
```
#Initializing values and computing H. Note the 1. to force to float type
C = 10
m,n = X.shape
y = y.reshape(-1,1) * 1.
X_dash = y * X
H = np.dot(X_dash , X_dash.T) * 1.

#Converting into cvxopt format - as previously
P = cvxopt_matrix(H)
q = cvxopt_matrix(-np.ones((m, 1)))
G = cvxopt_matrix(np.vstack((np.eye(m)*-1,np.eye(m))))
h = cvxopt_matrix(np.hstack((np.zeros(m), np.ones(m) * C)))
A = cvxopt_matrix(y.reshape(1, -1))
b = cvxopt_matrix(np.zeros(1))
```

```
#Run solver
sol = cvxopt_solvers.qp(P, q, G, h, A, b)
alphas = np.array(sol['x'])

#==================Computing and printing␣
↪parameters===============================#
w = ((y * alphas).T @ X).reshape(-1,1)
S = (alphas > 1e-4).flatten()
b = y[S] - np.dot(X[S], w)

#Display results
print('Alphas = ',alphas[alphas > 1e-4])
print('w = ', w.flatten())
print('b = ', b[0])
```

```
Alphas =  [ 5.      6.3125  1.3125 10.    ]
w =  [ 0.25 -0.25]
b =  [-0.75]
```

Comparing to Sklearn results

[6]:
```
from sklearn.svm import SVC

clf = SVC(C = 10, kernel = 'linear')
clf.fit(X, y.ravel())

print('w = ',clf.coef_)
print('b = ',clf.intercept_)
print('Indices of support vectors = ', clf.support_)
print('Support vectors = ', clf.support_vectors_)
print('Number of support vectors for each class = ', clf.n_support_)
print('Coefficients of the support vector in the decision function = ', np.
↪abs(clf.dual_coef_))
```

```
w =  [[ 0.25 -0.25]]
b =  [-0.75]
Indices of support vectors =  [0 2 3 6]
Support vectors =  [[ 3.  4.]
 [ 2.  3.]
 [ 6. -1.]
 [ 2.  4.]]
Number of support vectors for each class =  [2 2]
Coefficients of the support vector in the decision function =  [[ 5.      6.3125
1.3125 10.    ]]
```

0.1.5 3. Implementing Kernels

[ ]:
```python
import numpy as np
import cvxopt
from sklearn import datasets

def linear(x, z):
    return np.dot(x, z.T)

def polynomial(x, z, p=5):
    return (1 + np.dot(x, z.T)) ** p

def gaussian(x, z, sigma=0.1):
    return np.exp(-np.linalg.norm(x - z, axis=1) ** 2 / (2 * (sigma ** 2)))

def plot_contour(X, y, svm):
    # plot the resulting classifier
    h = 0.01
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    points = np.c_[xx.ravel(), yy.ravel()]

    Z = svm.predict(points)
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.8)

    # plt the points
    plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)


class SVM:
    def __init__(self, kernel=gaussian, C=1):
        self.kernel = kernel
        self.C = C

    def fit(self, X, y):
        self.y = y
        self.X = X
        m, n = X.shape

        # Calculate Kernel
        self.K = np.zeros((m, m))
        for i in range(m):
            self.K[i, :] = self.kernel(X[i, np.newaxis], self.X)
        # Solve with cvxopt final QP needs to be reformulated
        # to match the input form for cvxopt.solvers.qp
        P = cvxopt.matrix(np.outer(y, y) * self.K)
        q = cvxopt.matrix(-np.ones((m, 1)))
        G = cvxopt.matrix(np.vstack((np.eye(m) * -1, np.eye(m))))
        h = cvxopt.matrix(np.hstack((np.zeros(m), np.ones(m) * self.C)))
        A = cvxopt.matrix(y, (1, m), "d")
        b = cvxopt.matrix(np.zeros(1))
        cvxopt.solvers.options["show_progress"] = False
        sol = cvxopt.solvers.qp(P, q, G, h, A, b)
        self.alphas = np.array(sol["x"])

    def predict(self, X):
        y_predict = np.zeros((X.shape[0]))
        sv = self.get_parameters(self.alphas)

        for i in range(X.shape[0]):
            y_predict[i] = np.sum(
                self.alphas[sv]
                * self.y[sv, np.newaxis]
                * self.kernel(X[i], self.X[sv])[:, np.newaxis]
            )

        return np.sign(y_predict + self.b)

    def get_parameters(self, alphas):
        threshold = 1e-5

        sv = ((alphas > threshold) * (alphas < self.C)).flatten()
        self.w = np.dot(X[sv].T, alphas[sv] * self.y[sv, np.newaxis])
        self.b = np.mean(
            self.y[sv, np.newaxis]
            - self.alphas[sv] * self.y[sv, np.newaxis] * self.K[sv, sv][:, np.
↪newaxis]
        )
        return sv

if __name__ == "__main__":
    X, y = datasets.make_moons(50)

    #transform our y to be -1 and 1 to meet svm purpose
    y[y==0] = -1

    svm = SVM(kernel=gaussian)
    svm.fit(X, y)
    y_pred = svm.predict(X)
```

```
plot_contour(X, y, svm)

print(f"Accuracy: {sum(y==y_pred)/y.shape[0]}")
```

[ ]:

```

```