06.3 - Supervised Learning - Classification - SVM

October 26, 2020

1 Programming for Data Science and Artificial Intelligence

1.1 6.3 Supervised Learning - Classification - SVM

1.1.1 Readings:

- [GERON] Ch3, 5
- [VANDER] Ch5
- [HASTIE] Ch12
- Step-by-step tutorials - https://www.svm-tutorial.com

[1]:

```
import numpy as np
import matplotlib.pyplot as plt
```

1.2 Support Vector Machines

SVM is a **binary classification** algorithm, often called the “widest street” algorithm to try to draw the best-fit line that maximize the margin between two classes.

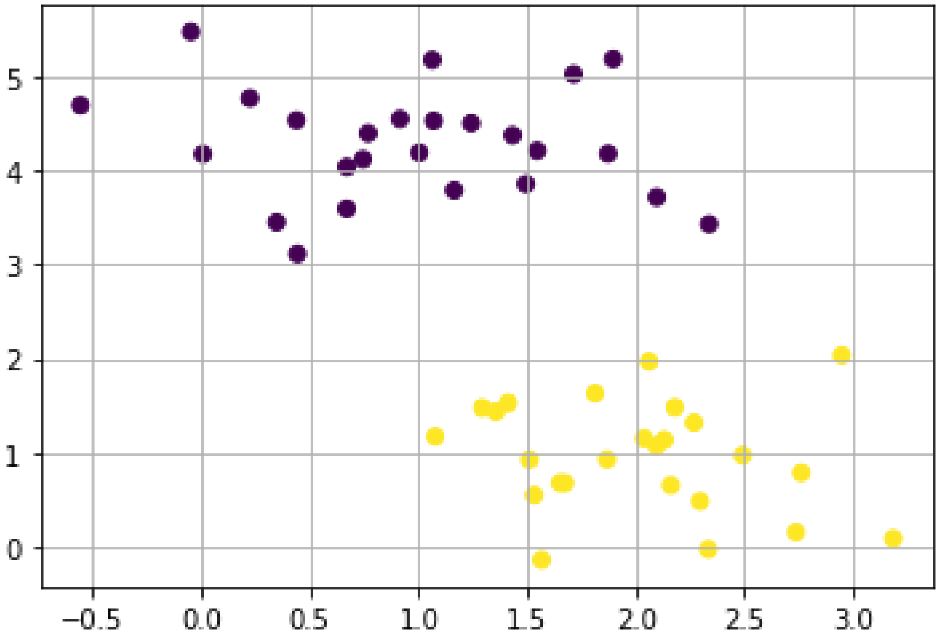
Let’s say we have this following data:

[2]:

```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=50, centers=2,
                  random_state=0, cluster_std=0.60)
plt.grid()
plt.scatter(X[:, 0], X[:, 1], c=y)
```

[2]:

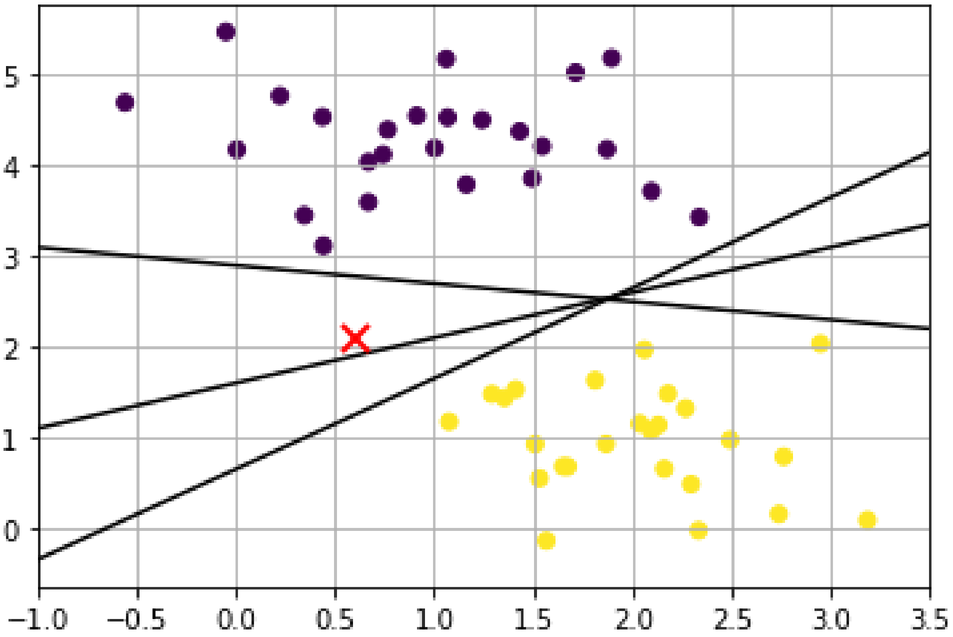
```
<matplotlib.collections.PathCollection at 0x12a473790>
```



In simple algorithms, we can draw a line separating them like this:

[3]:
```
xfit = np.linspace(-1, 3.5)
plt.grid()
plt.scatter(X[:, 0], X[:, 1], c=y)

for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:
    plt.plot(xfit, m * xfit + b, '-k')

plt.xlim(-1, 3.5)

#where should this value be classified as?
plt.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2, markersize=10)
```

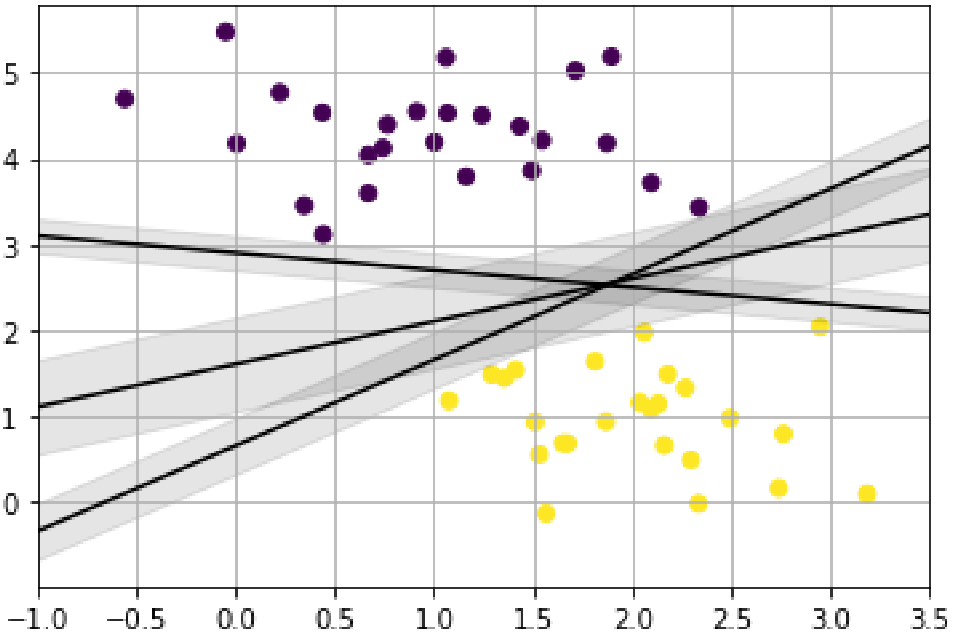
[3]:
```
[<matplotlib.lines.Line2D at 0x12a5c2810>]
```



You can probably already imagine the idea. It is basically draw around each line a margin of some width, up to the nearest point of each class. If this width is maximized for a particular line, that line should probably be the best.

[4]:
```
xfit = np.linspace(-1, 3.5)
plt.grid()
plt.scatter(X[:, 0], X[:, 1], c=y)

for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
    yfit = m * xfit + b
    plt.plot(xfit, yfit, '-k')
    plt.fill_between(xfit, yfit - d, yfit + d,
                     color='black', alpha=0.1)

plt.xlim(-1, 3.5)
```

[4]:
```
(-1.0, 3.5)
```



1.2.1 Mathematical Details

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 n 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:

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

If we multiply y on both equations, the hypothesis function can then be more concisely written as

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

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 $\mathbf{w}^\mathbf{T}\mathbf{x} + b = -1$ and the hyperplane defined by $\mathbf{w}^\mathbf{T}\mathbf{x} + b = 1$. 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:

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

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:

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

Why Dual, not Primal? 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.

Back to the Dual Problem Now that you are convinced dual problem is worth solving more than primal problem. To solve dual problem, we take the partial derivatives of $\mathcal{L}(w, b, \alpha)$ with respect to w and b, equate to zero (according to KKT conditions) 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

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

What's next To max α according to the equation, we cannot use gradient since the equation is neither concave nor convex. But notice the quadratic form of the equation. To optimize such equation, we can use **quadratic programming**. Later in the course, we shall discuss using **CVXOPT** python library. Solving the equation gives us back α, which using the α, we can get w, and using w, we can then get b. Hooray, we got our hypothesis function $\mathbf{w^T}\mathbf{x}^{(i)}+b$

We shall also discuss other minor improvements to the SVM. Namely two - 1) **Soft margin**, 2) **Kernels**.

Hard margin vs. Soft margin What we have done so far 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:

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

Following the inequality constraints on the slack variables, we add a beta in front of it (following the Lagrange method). Thus the Lagrange formulation change to:

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

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 $\beta_i\geq 0$:

$$\frac{\partial\mathcal{L}}{\partial\mathbf{w}}=\mathbf{w}-\sum\alpha_i\,y^{(i)}\,\mathbf{x}_i=0$$

$$\mathbf{w}=\sum\alpha_i\,y^{(i)}\,\mathbf{x}^{(i)}$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum \alpha_i y^{(i)} = 0,$$

$$\sum \alpha_i \, y^{(i)} = 0$$

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

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 Recall this equation:

$$\mathbf{w} = \sum \alpha_i \; y^{(i)} \; \mathbf{x}^{(i)}$$

Attempting to plug this back to our hypothesis function, we get

$$h = y^{(i)} (\sum \alpha_i \; y^{(i)} \; \mathbf{x}^{(i)} \mathbf{x'}^{(i)} + b) \geq 1$$

Here we put $'$ to another x to denote that these two x are different. Make an observation that to predict anything, we simply need to find the dot product of a pair of x

Based on this observation, 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 $\varphi : \mathbb{R}^n \rightarrow \mathbb{R}^d$, where d is of higher dimensions. Then the dot product of $\mathbf{x}$ and $\mathbf{x}'$ in this space is $\varphi(\mathbf{x})^T \varphi(\mathbf{x}')$. A kernel is a function k that corresponds to this dot product without having to performing φ, i.e.,

$$k(\mathbf{x}, \mathbf{x}') = \varphi(\mathbf{x})^T \varphi(\mathbf{x}')$$

Why is this useful? or How is this different from previous transformations such as polynomial feature engineering we did on linear regression? That is, kernels give a way to **compute dot products in some feature space without even knowing what this space is and what is** φ. Since we don't need to actually transform features, this saves a lot of computation time.

For example, consider a simple polynomial kernel of degree 2, $k(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x}^T\mathbf{x}')^2$ (Note that in general polynomial kernel, 2 can be p) with $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^2$. 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 $\mathbf{x}' = (x'_1, x'_2)$, let's expand this expression:

$$\begin{aligned} k(\mathbf{x}, \mathbf{x}') &= (1 + \mathbf{x}^T\mathbf{x}')^2 = (1 + x_1\, x'_1 + x_2\, x'_2)^2 = \\ &= 1 + x_1^2 x_1'^2 + x_2^2 x_2'^2 + 2x_1x'_1 + 2x_2x'_2 + 2x_1x_2x'_1x'_2 \end{aligned}$$

Note that this is nothing else but a dot product between two vectors of $\varphi(\mathbf{x}) = (1, x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2)$ and $\varphi(\mathbf{x}') = (1, x_1'^2, x_2'^2, \sqrt{2}x'_1, \sqrt{2}x'_2, \sqrt{2}x'_1x'_2)$. So the kernel $k(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x}^T\mathbf{x}')^2 = \varphi(\mathbf{x})^T\varphi(\mathbf{x}')$ computes a dot product in 6-dimensional space without explicitly visiting this space.

Another example is Gaussian kernel:

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

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

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

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

Using kernel, our optimization problem can be revised to:

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

And our lagrange formula changed only slightly to

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

Making a prediction Once the α_i are learned, one can predict the class of a new sample with the feature vector X_test as follows:

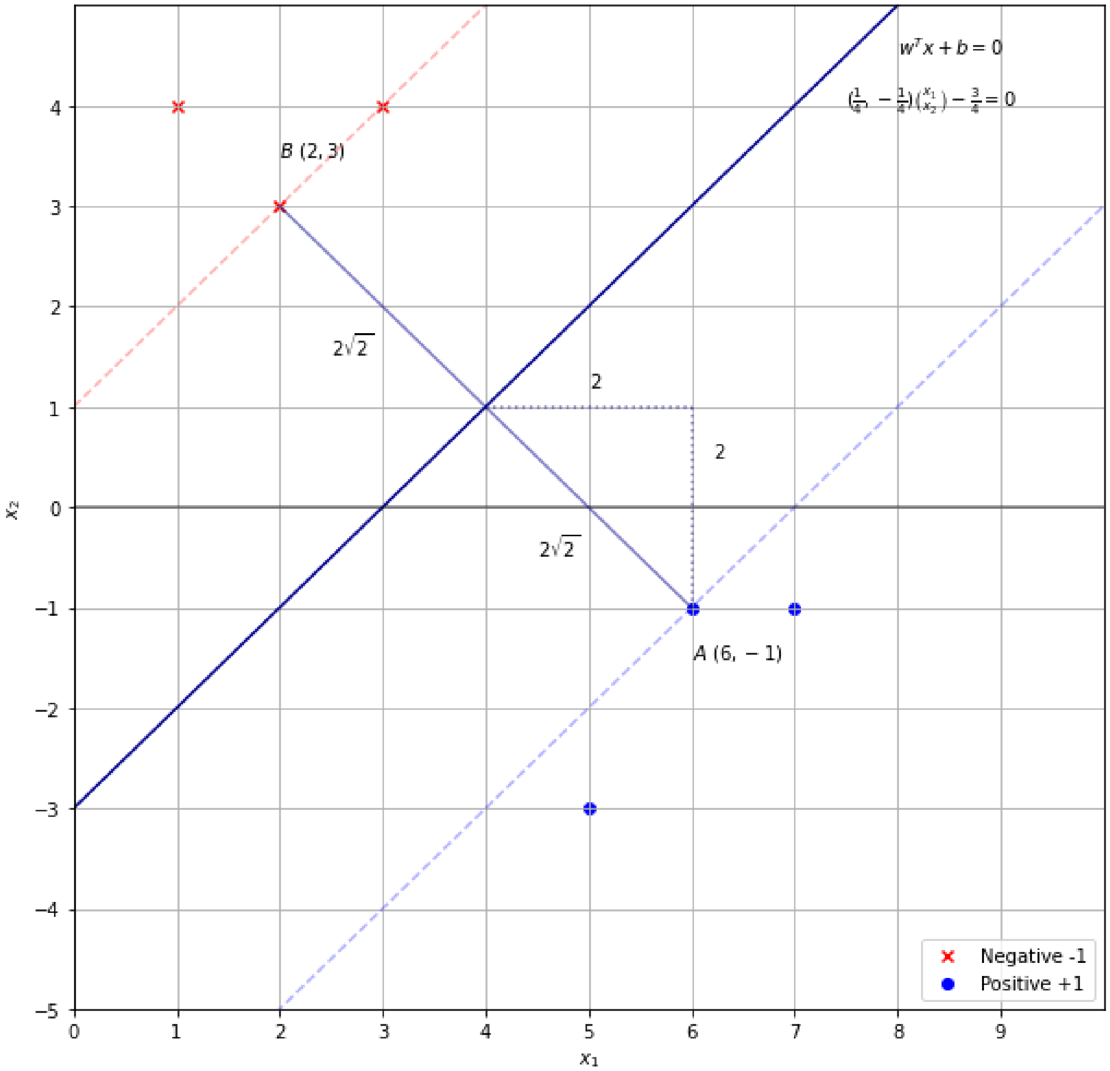
$$\begin{aligned}\hat{y} &= \operatorname{sign}\left(\mathbf{w}^{\mathbf{T}}\phi\left(\mathbf{x}^{\text{test}}\right)+b\right)\\ &= \operatorname{sign}\left(\sum_{i=1}^{m}\alpha_i y^{(i)}\phi\left(x^{(i)}\right)^{T}\phi\left(\mathbf{x}^{\text{test}}\right)+b\right)\end{aligned}$$

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

[5]:

```
#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 )
```

```python
#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()
```



1.3 1. Scratch: SVM with 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

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

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

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 $H = X'X'.T$

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

[6]:

```
#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

[7]:

```
#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])
```

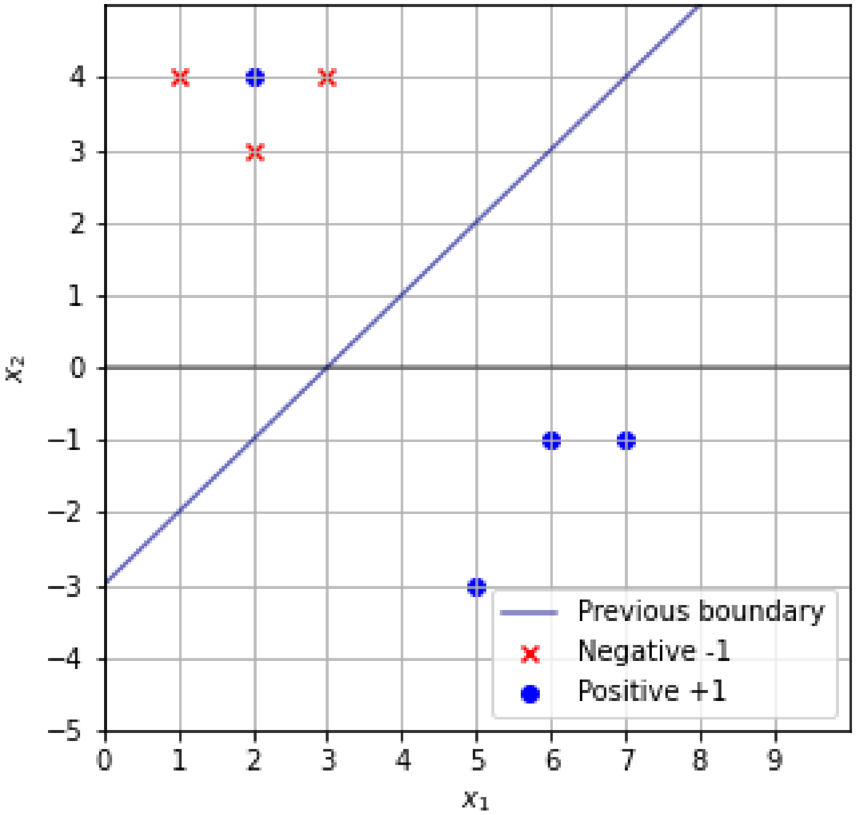
```
Alphas =  [0.0625     0.06249356]
w =  [ 0.24999356 -0.25000644]
b =  [-0.74996781]
```

1.4 2. Scratch: SVM with Soft Margin

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

[8]:

```
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

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

which can be written in standard form as

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

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:

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

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

[9]:

```
#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

```
[10]: 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_))
```

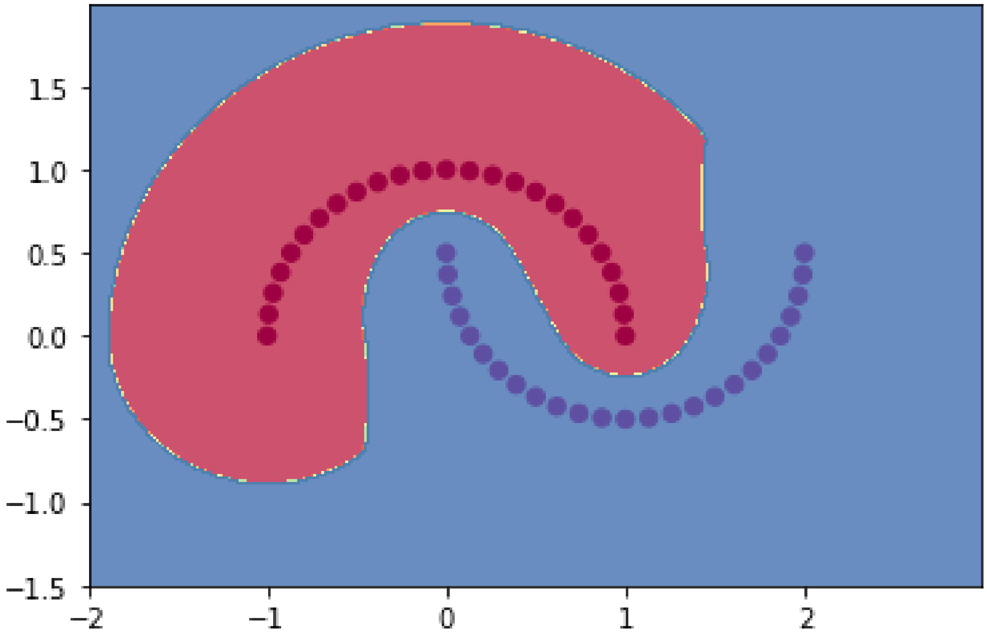
```
w =  [[ 0.25 -0.25]]
b =  [-0.75]
Indices of support vectors =  [0 2 3 6]
Support vectors =  [[ 3.  4.]
 [ 2.  3.]
 [ 6. -1.]
 [ 2.  4.]]
Number of support vectors for each class =  [2 2]
Coefficients of the support vector in the decision function =  [[ 5.      6.3125
1.3125 10.    ]]
```

1.5 3. Scratch: Implementing Kernels

[11]:

```
import cvxopt
from sklearn import datasets

#here I use z instead of xprime since I don't know how to write prime in code....
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):  #<----this is X_test
        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(self.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]}")
```

```
Accuracy: 1.0
```



1.6 4. Sklearn

[12]:
```
from sklearn.svm import SVC # "Support vector classifier"
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import train_test_split

#this code will run very long time!!!

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
assert X_train.shape[0] == y_train.shape[0]

#select the type of hyperplane used to separate the data
kernel = ['linear', 'rbf']

#regularized penalty
#when C is small, classifier is ok with misclassified data
#when C is large, classifier is heavily penalized for misclassified data
C_range = [1, 2]
```

```
#parameter for RBF kernel, increasing gamma
#allows for more complex decision boundary but can lead
#to overfitting
gamma_range = [0.0001, 0.0005]

param_grid = dict(gamma=gamma_range, C=C_range, kernel=kernel)
cv = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=cv)
grid.fit(X_train, y_train)

print("=========Best parameters=======")
print(f"The best parameters are {grid.best_params_} with" +
          f"a score of {grid.best_score_:.2f}")

model = grid.best_estimator_

model.fit(X_train, y_train)
yhat = model.predict(X_test) #svm use this instead

#plot precision recall curve
from sklearn.metrics import average_precision_score, classification_report,␣
 ↪confusion_matrix
print("=========Average precision score=======")
print(average_precision_score(y_test, yhat))
print("=========Classification report=======")
print("Report: ", classification_report(y_test, yhat))

"""
If you want better score, try larger range of params...I tried only a few so␣
 ↪not to slow
down my computer.
"""
```

```
=========Best parameters=======
The best parameters are {'C': 1, 'gamma': 0.0001, 'kernel': 'linear'} witha
score of 0.80
=========Average precision score=======
0.875
=========Classification report=======
Report:                 precision    recall  f1-score   support

          -1       1.00      0.88      0.93         8
           1       0.88      1.00      0.93         7

    accuracy                           0.93        15
   macro avg       0.94      0.94      0.93        15
```

```
weighted avg       0.94      0.93      0.93        15
```

[12]:

```
'\nIf you want better score, try larger range of params…I tried only a few so
not to slow\ndown my computer.\n'
```

1.6.1 When to Use SVM

SVM work fantastic, given the following advantages: - Kernel trick allows non-linearity and works well with high-dimensional data - Supports regularization out of the box with the slack variables - Use only few support vectors as decision boundary thus the prediction phase is very fast.

However, SVMs have several disadvantages as well: - For large number of training amples, the computational cost can be scary! Around $\mathcal{O}[N^3]$ at worst - Require fine-tuning C which often takes a lot of time! - Does not support probability - only output the class

We usually use SVM when our Baye or Logistic fellows fail us, or we believe our data is non-linear to some extent.

[ ]:

```

```