Support Vector Machines (SVM)

Elsa Ferreira Gomes

2023/2024

- **https://hastie.su.domains/ISLP/ISLP_website.pdf** - Chp9
- This notebook contains an excerpt from the Python Data Science Handbook by Jake VanderPlas

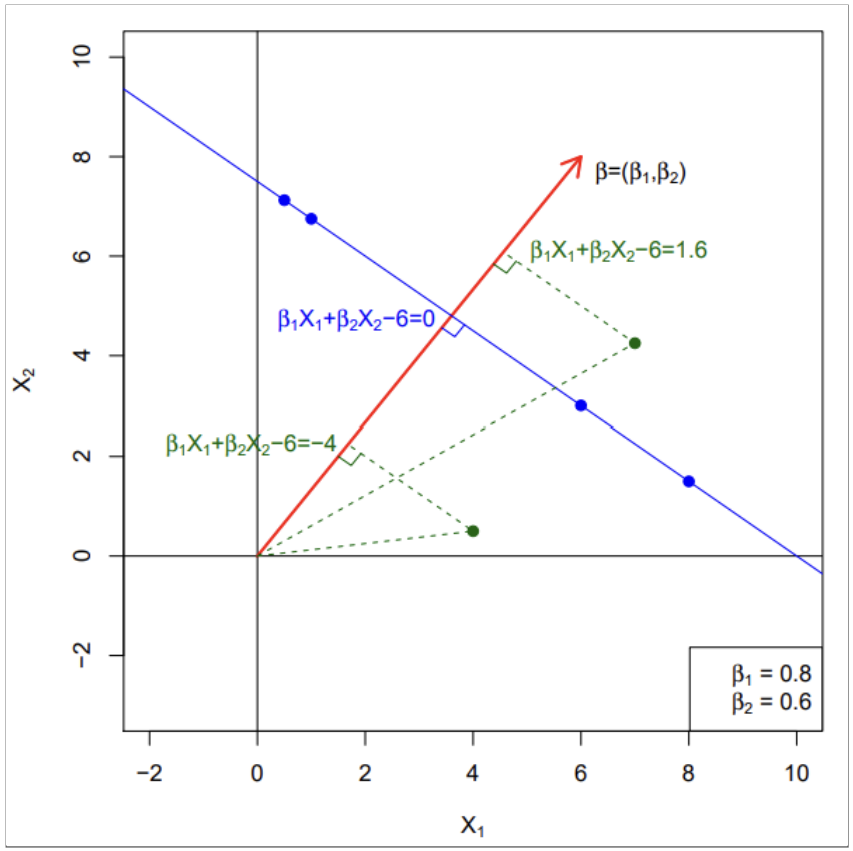
Support Vector Machines for Classification

- Support vector machines (SVM) are a powerful tool of supervised algorithms for both Classification and Regression.
- We will study support vector machines and their use in **Classification problems**.
- Given a set of points of two different classes we want to find a **good boundary**

Hyperplane in 2 dimensions

The mathematical defnition of a hyperplane is quite simple. In two dimensions, a hyperplane is defned by the equation

$\beta_0 + \beta_1 x_1 + \beta_2 x_2 = 0$



Hyperplane

A hyperplane in p dimensions equation:
$h(x) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p = 0$

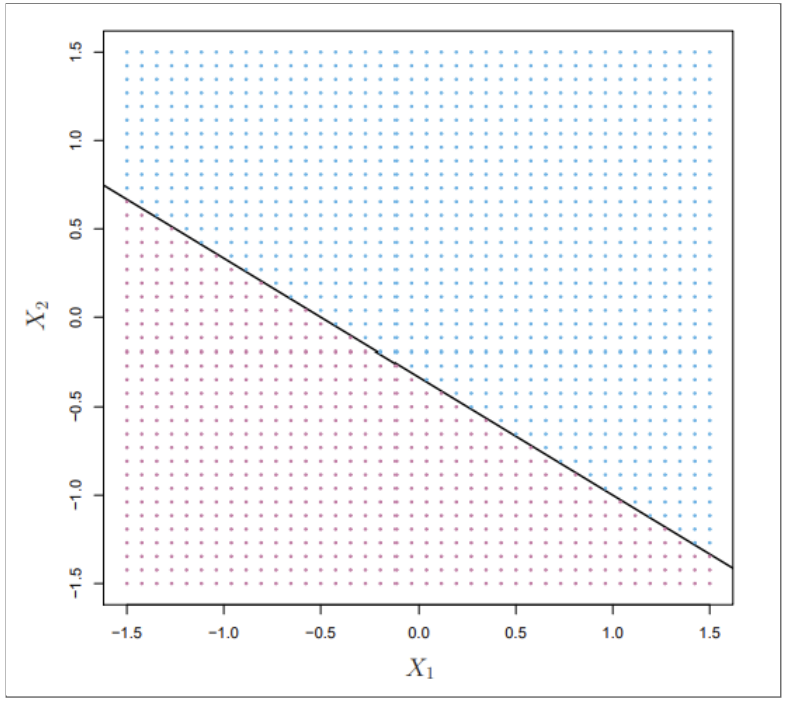
- If $p = 2$ the hyperplane is a line.
- If $\beta_0 = 0$, the hyperplane goes through the origin (otherwise not).
- The vector $\beta = (\beta_1, \ldots, \beta_p)$ is called the normal vector
    - it points in a direction orthogonal to the surface of a hyperplane

Finding a decision boundary

- Our training data are pairs (x_i, y_i)
- $x_i \in \mathbb{R}^p$
- $y_i \in \{1, -1\}$
- In this case the classes are **separable**
- One **possible criterium** is
    - Find the boundary that is **most distant** to the **nearest points**

Example of a separating Hyperplane

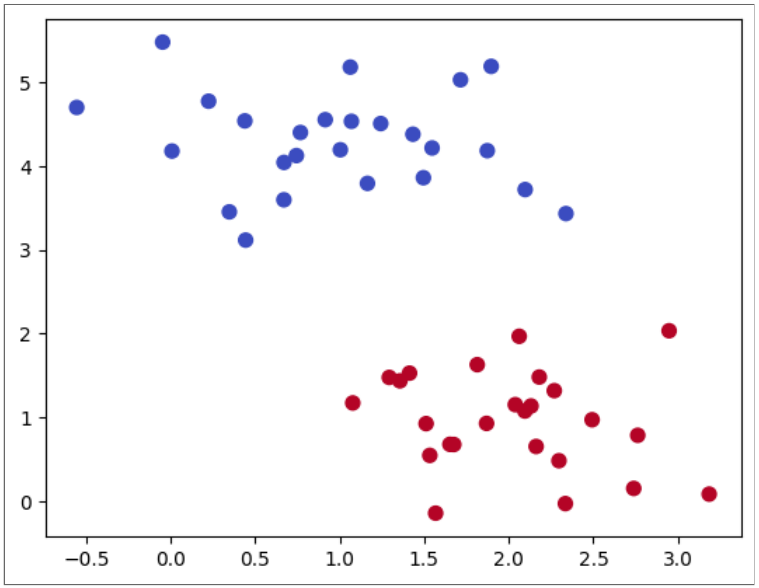
Example:$1 + 2X_1 + 3X_2 = 0$



Where:

- Blue region is the set of points for which $1 + 2X_1 + 3X_2 > 0$,
- Purple region is the set of points for which $1 + 2X_1 + 3X_2 < 0$

In [2]:
```
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=50, centers=2,
                  random_state=0, cluster_std=0.60)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap=cm.coolwarm);
```



Support Vector Classifier

As an example of this, consider the simple case of a classification task, in which the two classes of points are well separated:
A linear discriminative classifier would attempt to draw a straight line separating the two sets of data, and then create a model for classification.

- for two dimensional, this is a task we could do by hand.
- there is more than one possible dividing line that can perfectly discriminate between the two classes

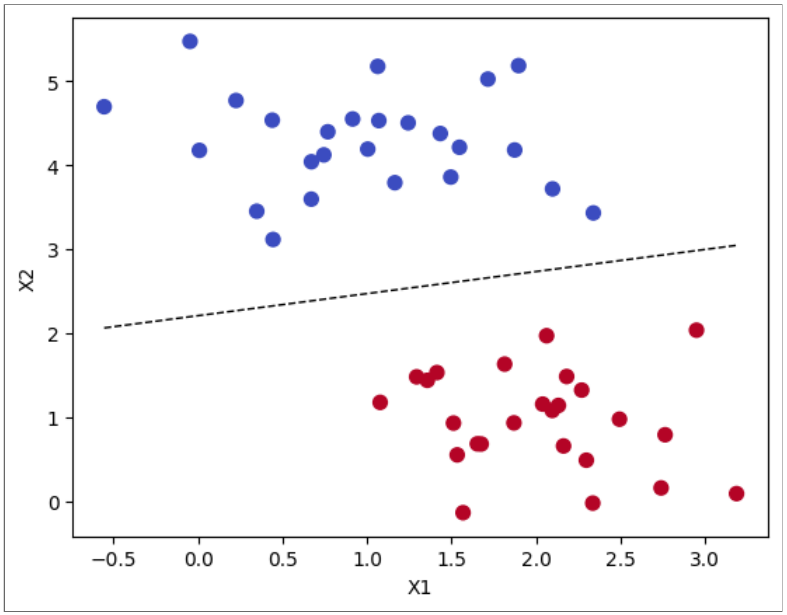
We can draw them as follows:

Finding a decision boundary

- Many good **linear** boundaries can separate the two classes
- Logistic regression gives **one answer**
    - **Assumes** that $\Pr(class)$ under the parameters follows a logistic law
    - Uses **maximum likelihood** to infer the parameters
- LDA gives a slightly different answer, in general
    - Different assumptions

In [3]:
```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression(random_state=0).fit(X, y)

# Retrieve the model parameters.
b = clf.intercept_[0]
w1, w2 = clf.coef_.T

# Calculate the intercept and gradient of the decision boundary.
c = -b/w2
m = -w1/w2

# prepare the points for drawing the linear boundary
xmin, xmax = min(X[:,0]), max(X[:,0])
xd = np.array([xmin, xmax])
yd = m*xd + c

# replot, but now with the boundary
##xfit = np.linspace(-1, 3.5)
```

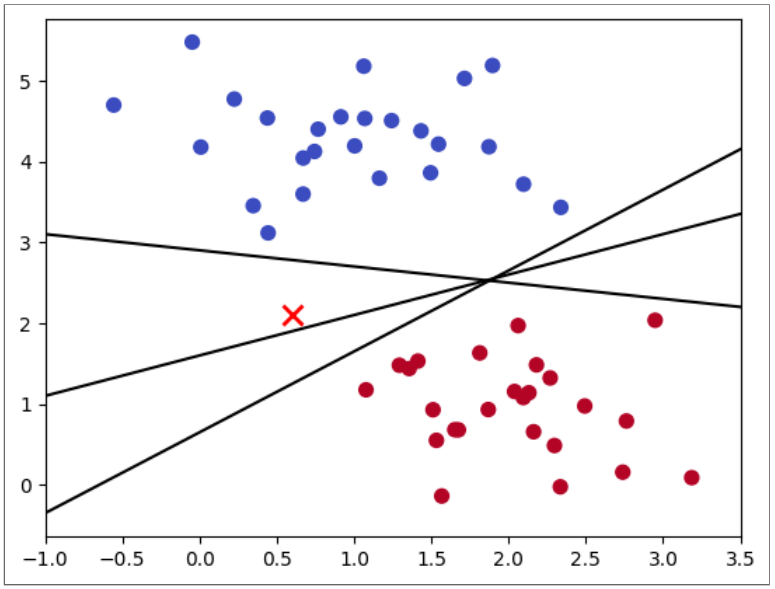
In [4]:
```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap=cm.coolwarm)

plt.xlabel('X1')
plt.ylabel('X2')
plt.plot(xd, yd, 'k', lw=1, ls='--')
plt.show()

w1, w2, b
```



Out[4]:

```
(array([0.57306256]), array([-2.18013484]), 4.813932021145475)
```

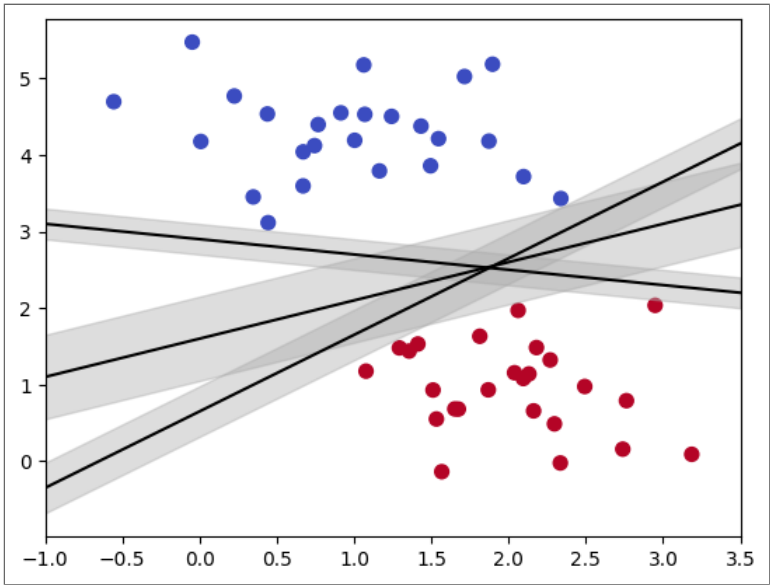
In [5]:
```
xfit = np.linspace(-1, 3.5)
       plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap=cm.coolwarm);
plt.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2, markersize=10)

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);
```



Maximizing the Margin

- We have three very different separators discriminate between these samples.
- Depending on the choose a new data point ("X" in this plot) will be assigned a different label.
- Support vector machines offer one way to improve on this. The intuition is this:
    - rather than simply drawing a zero-width line between the classes, we can draw around each line a margin of some width, up to the nearest point.
        - this margin is the one we will choose as the optimal model. Support vector machines are an example of such a **maximum margin** estimator.

In [6]:
```
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap=cm.coolwarm)

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, edgecolor='none',
                     color='#AAAAAA', alpha=0.4)

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



Finding a decision boundary

- A linear boundary in $\mathbb{R}^p$ is an **hyperplane** defined by $h(x) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p = 0$
- The **signed distance** between a point x and the hyperplane is equal to $h(x)$, if $\|\beta\| = 1$
- Given a boundary, we have a **classifier**
    - $G(x) = sign[x^T \beta + \beta_0]$
- We need to know which is the **best boundary** (**best classifier**)

Finding a decision boundary

- If the classes are **linearly separable** as in the example
    - We can find a boundary such that $y_i h(x_i) > 0 \ \ \forall i$
    - I.e., all points are on the right side of the boundary
        - If $y_i = 1$, then x_i should have a **positive distance** to the boundary h
        - If $y_i = -1$, then x_i should have a **negative distance** to the boundary h

To better visualize and uderstand, we can use a function that will plot SVM decision boundaries.

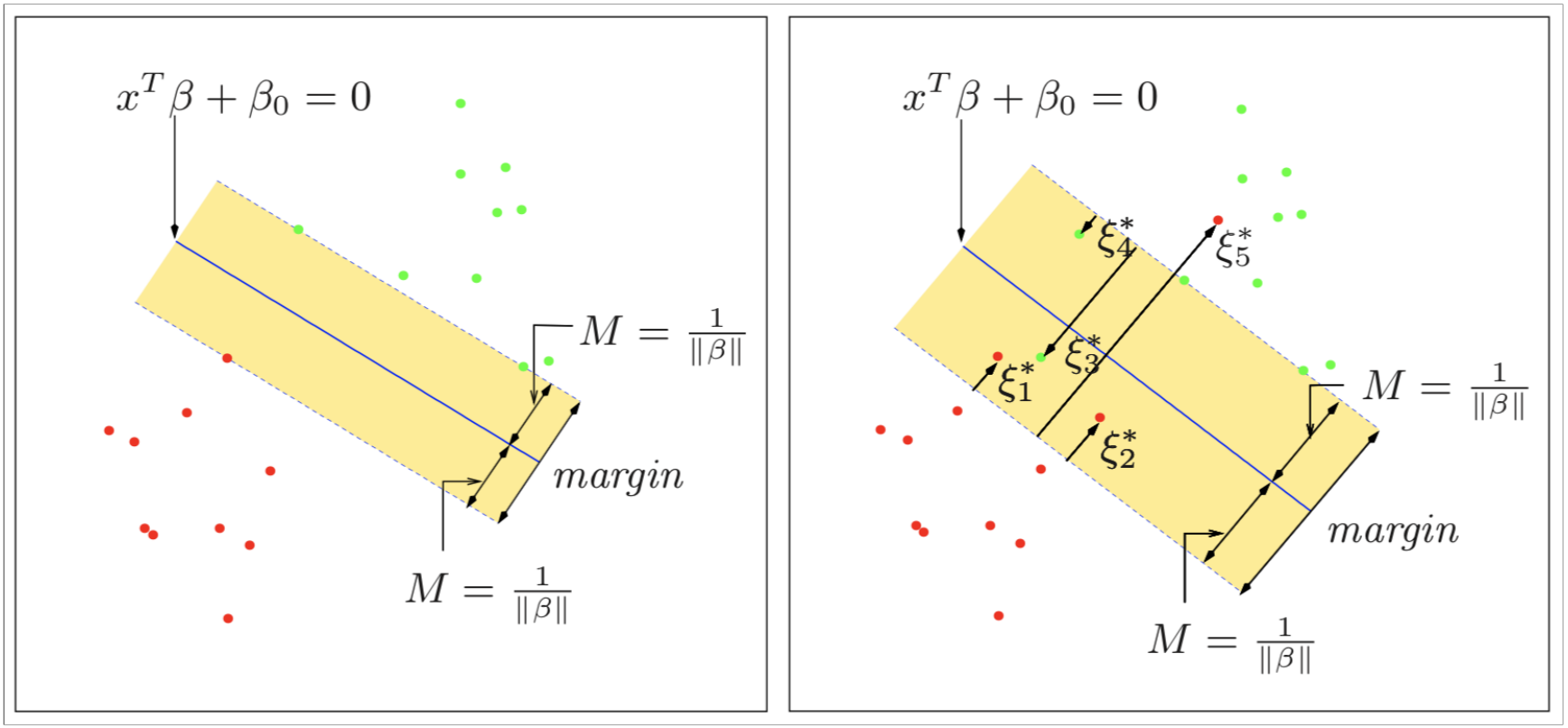
- We can now look for the boundary that maximizes **distance to the nearest point** of any class
- The maximal value M such that $y_i h(x_i) \geq M \ \ \forall i$ is the **margin**
    - we want the boundary that **maximizes** M
    - this gives us the largest **safety margin** with respect to the data

$$\max_{\beta, \beta_0, \|\beta\|=1} M$$

$$\text{subject to} \ \ \ y_i(x_i^T \beta + \beta_0) \geq M$$

In [7]:
```
from IPython.display import Image
Image("10n-SVM-margin-fig12.1-ESL.png")
```

Out[7]:



Fitting a Support Vector Machine

Let see the result of an actual fit to this data: we will use Scikit-Learn's support vector classifier to train an SVM model on this data. We will use a linear kernel and set a value to the **C** parameter.

In [8]:
```
from sklearn.svm import SVC # "Support vector classifier"

# When using a smaller cost parameter (C=0.1) the margin is wider, resulting in more support vectors.
svm_linear = SVC(kernel='linear', C=1E10)
svm_linear.fit(X, y)
```

Out[8]:

SVC

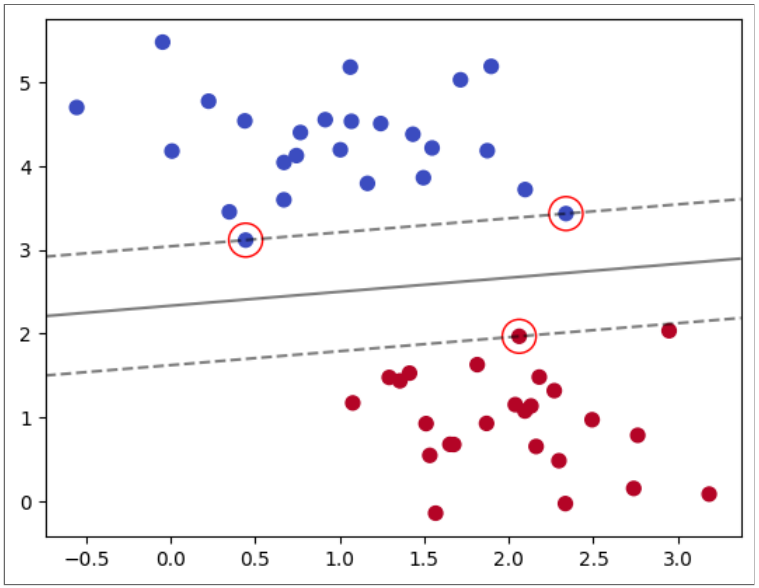
```
SVC(C=10000000000.0, kernel='linear')
```

In [9]:
```
def plot_svc_decision_function(model, ax=None, plot_support=True):
        """Plot the decision function for a 2D SVC"""
    if ax is None:
        ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # create grid to evaluate model
    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    Y, X = np.meshgrid(y, x)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P = model.decision_function(xy).reshape(X.shape)

    # plot decision boundary and margins
    ax.contour(X, Y, P, colors='k',
               levels=[-1, 0, 1], alpha=0.5,
               linestyles=['--', '-', '--'])

    # plot support vectors
    if plot_support:
        ax.scatter(model.support_vectors_[:, 0],
                   model.support_vectors_[:, 1],
                   s=300, linewidth=1, edgecolors='red', facecolors='none');
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
```

In [10]:
```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap=cm.coolwarm)
        plot_svc_decision_function(svm_linear);
svm_linear.support_vectors_
```

Out[10]:

```
array([[0.44359863, 3.11530945],
       [2.33812285, 3.43116792],
       [2.06156753, 1.96918596]])
```



Tuning the SVM

But what if your data has some amount of overlap?

- The SVM implementation has a bit of a fudge-factor which "softens" the margin: that is, it allows some of the points to creep into the margin if that allows a better fit.
- The hardness of the margin is controlled by a tuning parameter, most often known as C.
    - For very large C: the margin is hard, and points cannot lie in it.
    - For smaller C: the margin is softer, and can grow to encompass some points.

Fitting a Support Vector Machine

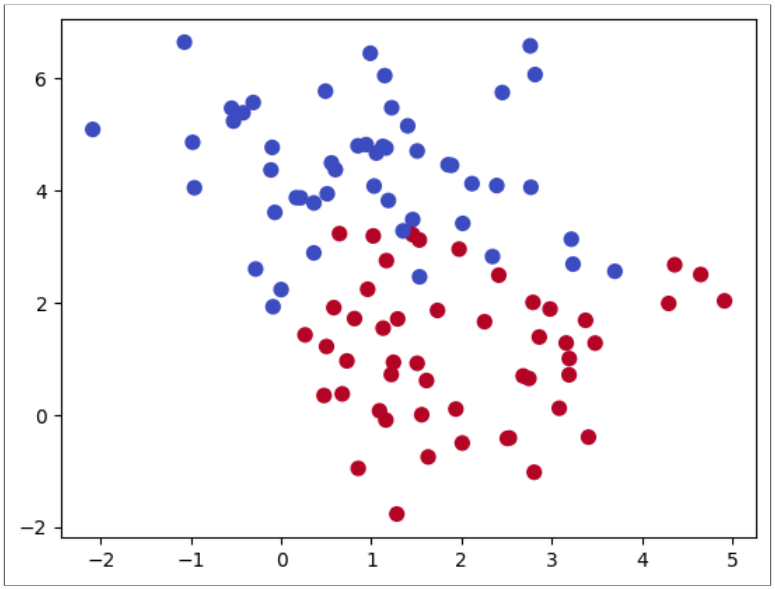
- This line that maximizes the margin between the two sets of points.
- Notice that a few of the training points just touch the margin (in the circles in this figure):
    - These points are the pivotal elements of this fit, and are known as the support vectors, and give the algorithm its name.

In Scikit-Learn, the identity of these points are stored in the **support*vectors* attribute** of the classifier

- Only the position of the support vectors matter;
    - any points further from the margin which are on the correct side do not modify the fit

Technically, this is because these points do not contribute to the loss function used to fit the model, so their position and number do not matter so long as they do not cross the margin.
We can see this, for example, if we plot the model learned from the first 60 points and first 120 points of this dataset:

In [11]:
```
X, y = make_blobs(n_samples=100, centers=2,
                  random_state=0, cluster_std=1.2)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap=cm.coolwarm);
```

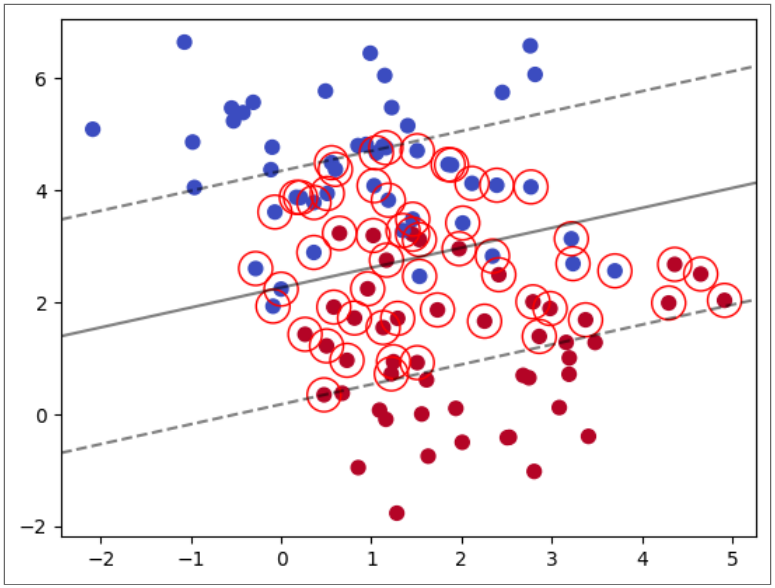


In [12]:
```
# Select the optimal C parameter by cross-validation
kfold = skm.KFold(5,
                  random_state=0,
                  shuffle=True)
grid = skm.GridSearchCV(svm_linear,
                        {'C':[0.001,0.01,0.1,1,5,10,100]},
                        refit=True,
                        cv=kfold,
                        scoring='accuracy')
grid.fit(X, y)
grid.best_params_
```

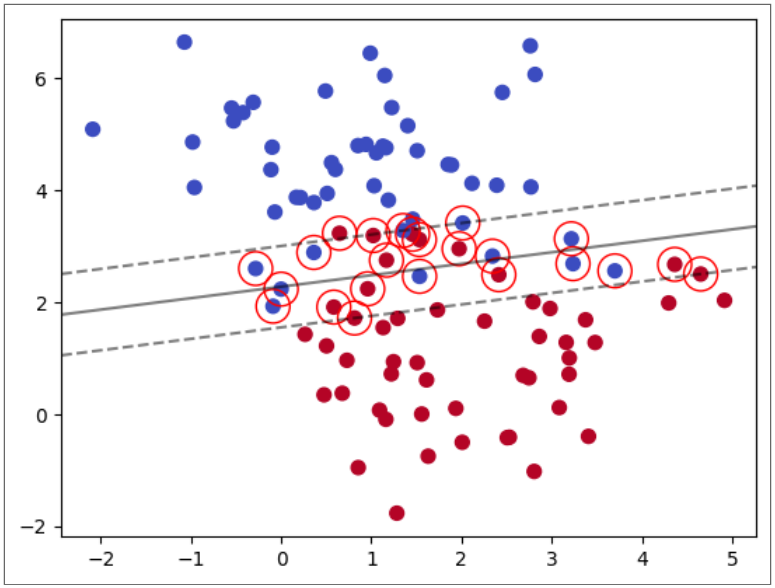
Out[12]:

```
{'C': 0.01}
```

In [13]:
```
svm_linear_small = SVC(C=0.01, kernel='linear')
svm_linear_small.fit(X, y)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap=cm.coolwarm)
plot_svc_decision_function(svm_linear_small);
```



In [14]:
```
svm_linear_small = SVC(C=20, kernel='linear')
svm_linear_small.fit(X, y)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap=cm.coolwarm)
plot_svc_decision_function(svm_linear_small);
```



In this figure we can see the dividing line that maximizes the margin between the two sets of points.

- Only a few of the training points just touch the margin (marked with the black circles).
- These points are the pivotal elements of this fit, and are known as the support vectors. In Scikit-Learn, the identity of these points are stored in the support*vectors* attribute of the classifier:

In [15]:
```
svm_linear.coef_
```

Out[15]:

```
array([[ 0.23525694, -1.41250783]])
```

Softening the Decision Boundary

- The learning problem can be **softened** as

$$\max_{\beta,\beta_0,\|\beta\|=1} M$$

$$\text{subject to} \quad y_i(x_i^T\beta + \beta_0) \geq M(1-\xi_i)$$

$$\xi_i \geq 0\,, \quad \sum_i \xi_i \leq Cost_{max}$$

- This leads to the standard **Support Vector Classifier**
  - The ξ_i are **exceptions** to the margin
  - The constant $Cost_{max}$ is like a budget for the misclassification cost

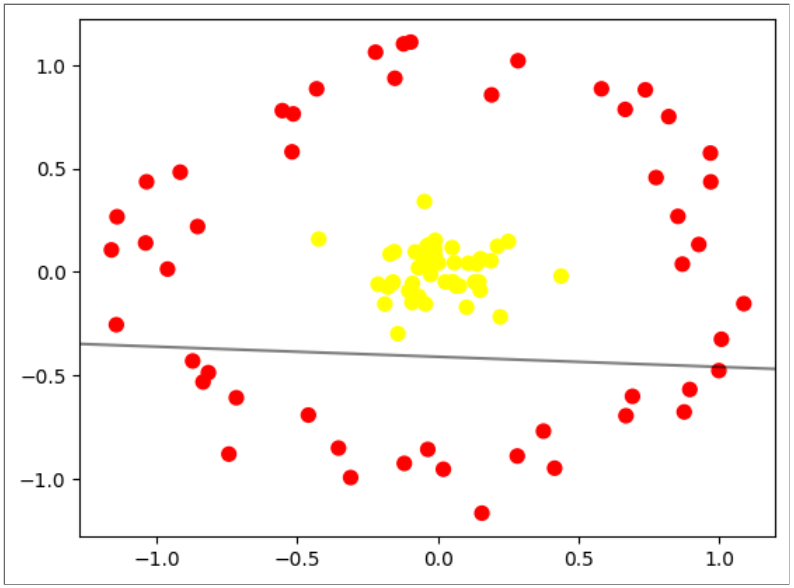
Support Vector Classifier > Example

- An artificial example with non-separable classes
- Large C has **fewer** support vectors
    - If C is large, minimization focuses on ξ_i
    - β_i tend to be larger
    - The **margin is smaller**

Support Vector Machine with Non Linear Kernel

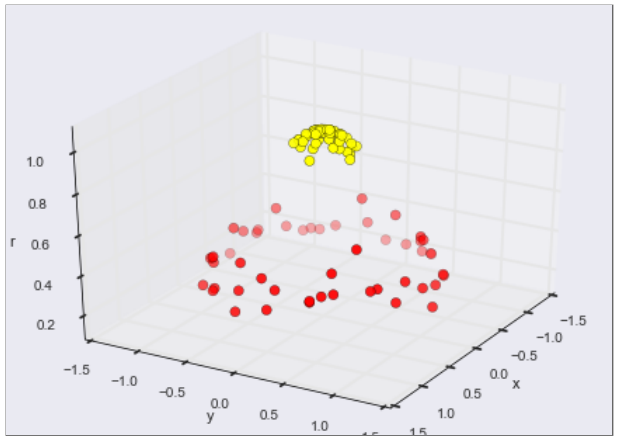
SVM is extremely powerful when it is combined with *kernels*.
To motivate the need for kernels, let's look at some data that is not linearly separable

In [18]:
```
from sklearn.datasets import make_circles
X, y = make_circles(100, factor=.1, noise=.1)

#clf = SVC(kernel='linear', C=1).fit(X, y)

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

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');

plot_svc_decision_function(clf, plot_support=False);
```



No linear discrimination will ever be able to separate this data!
But how we might project the data into a higher dimension such that a linear separator would be sufficient.

- For example, one simple projection we could use would be to compute a radial basis function centered on the middle clump: r=np.exp(-(X ** 2).sum(1))
- We can visualize this extra data dimension using a three-dimensional plot:



The kernel trick

A potential problem with this strategy—projecting N points into N dimensions—is that it might become very computationally intensive as N grows large. However, because of a neat little procedure known as the **kernel trick**, a fit on kernel-transformed data can be done implicitly
This kernel trick is built into the SVM, and is one of the reasons the method is so powerful.
In Scikit-Learn, we can apply kernelized SVM simply by changing our linear kernel to an RBF **Radial Basis Function** kernel, using the kernel model hyperparameter.
This kernel transformation strategy is used often in machine learning to turn fast linear methods into fast nonlinear methods, especially for models in which the kernel trick can be used.

- We can project the **original space**
    - into an **expanded space**
    - where the data is **linearly separable** (see the next example, eg. *r*=0.7.)
- This is done using the **Kernel trick** (**https://en.wikipedia.org/wiki/Kernel_trick**)
- In the example below, the points are **projected** on the surface of a sphere
    - SVM does this by generalizing the algorithm.
- The polynomial kernel can produce non-linear boundaries (try `rbf` as well).

SVM: kernels

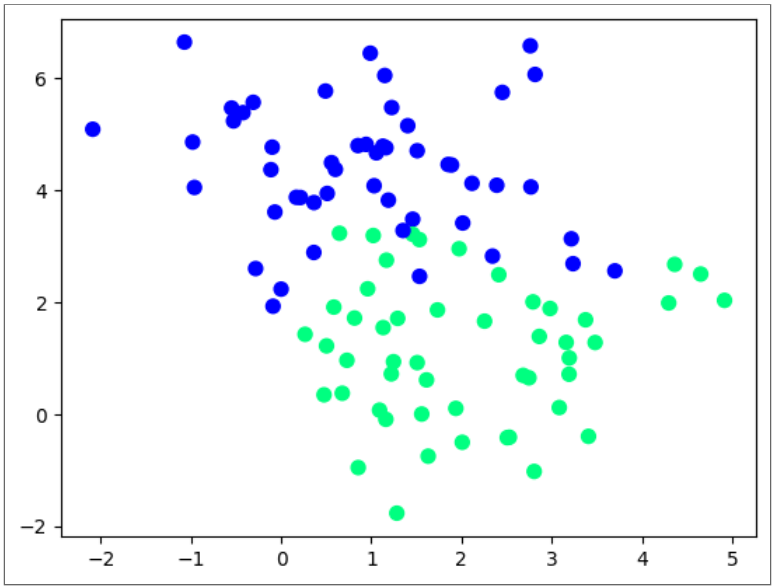
- **How** do we expand the original space in a **feasible way**?
- It can be shown that the vector β can be **defined** using the support vectors

$$\beta = \sum_{i=1}^{N} \alpha_i y_i x_i$$

  - α_i are non-negative and are **non zero** only for the support vectors
- So, $f(x)$, the **classifier**, is

$$G(x) = sign[\sum_{i=1}^{N} \alpha_i y_i x_i x + \beta_0]$$

In [19]:
```
# from van der Plas, an example without a clear separation
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=100, centers=2,
                  random_state=0, cluster_std=1.2)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='winter');
```



In [20]:
```
# we can also try a different kernel on this data

clf = SVC(kernel='rbf',gamma=0.5,C=100).fit(X, y)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='winter')
plot_svc_decision_function(clf, plot_support=True)
print("#SV =",len(clf.support_vectors_))
```

```
#SV = 23
```