Support Vector Machines (SVMs)

SVMs are very powerful binary classifiers, based on the Statistical Learning Theory (SLT) framework. SVMs can be used to solve hard classification problems, where they look for an optimal hyperplane able to maximize the classifier margin.

Practical example - classifier margin

First of all we do all necessary imports.

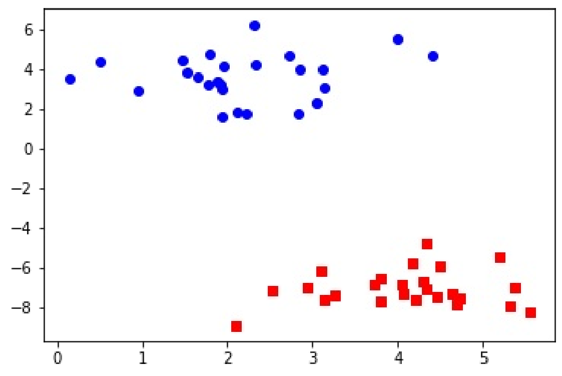
In [1]:

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

from mpl_toolkits.mplot3d import Axes3D

from sklearn.datasets import make_blobs, make_circles
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC

# Setting random seed.
seed = 10
```

Then, we generate a very simple linear separable dataset and plot it.

In [2]:

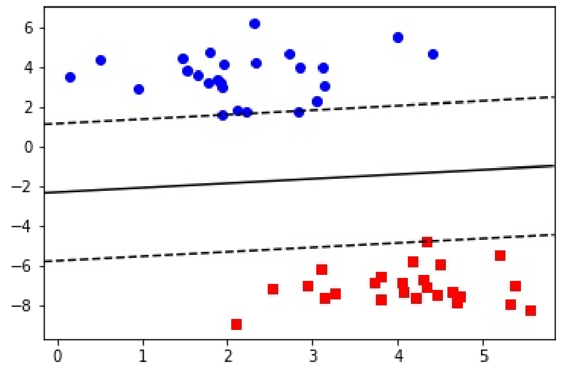
```
# Generating a linear separable dataset with 50 samples and 2 classes.
X, y = make_blobs(n_samples=50, centers=2, center_box=[-7.5, 7.5], random_state=seed)

# Method to plot the linear separable dataset.
def plot_data(X, y):
    class0 = np.where(y == 0)[0]
    plt.scatter(X[class0, 0], X[class0, 1], c='red', marker='s')

    class1 = np.where(y == 1)[0]
    plt.scatter(X[class1, 0], X[class1, 1], c='blue', marker='o')

plot_data(X, y)
plt.show()
```



Next, we train a SVM classifier with linear kernel and plot the optimal hyperplane as well as the classifier margins.

In [3]:

```
svm = SVC(C=100, kernel='linear', random_state=seed)
svm.fit(X, y)

plot_data(X, y)

# Method to plot SVMs' hyperplane and margins.
# This code is based on http://scikit-learn.org/stable/auto_examples/svm/plot_separating_hyperplane.html
def plot_margins(svm, X, y):
    xmin, xmax = plt.xlim()
    ymin, ymax = plt.ylim()

    # create grid to evaluate model
    xx = np.linspace(xmin, xmax, 30)
    yy = np.linspace(ymin, ymax, 30)
    XX, YY = np.meshgrid(xx, yy)
    xy = np.vstack([XX.ravel(), YY.ravel()]).T
    Z = svm.decision_function(xy).reshape(XX.shape)

    # plot decision boundary and margins
    plt.contour(XX, YY, Z, colors='black', levels=[-1, 0, 1], alpha=1.0, linestyles=['--', '-', '--'])

plot_margins(svm, X, y)
plt.show()
```



In the above plot, the filled line represents the optimal hyperplane found while the dashed lines represent the hyperplanes defined by the support vectors. The margin of the classifier is the distance between the optimal hyperplane and any of the support vector hyperplanes.

Practical example - Non-linear decision boundary

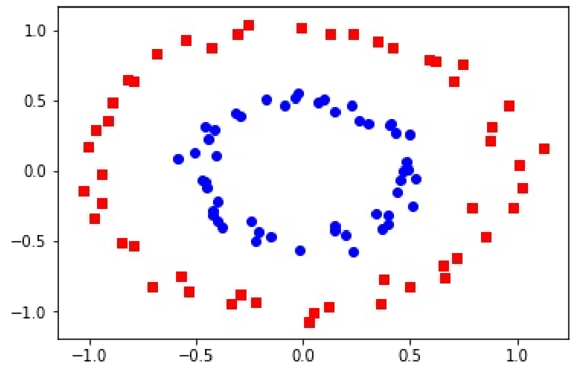
SVMs are linear classifiers. Since most of the real world problems are not linearly separable, how can we deal with them?

Next, we will show a very simple application of the Kernel Trick, which ables us to learn non-linear decision boundaries.

First, we generate a very simple and not linearly separable dataset.

In [14]:

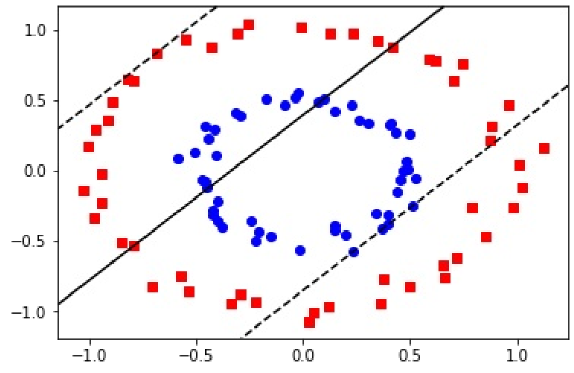
```
X, y = make_circles(n_samples=100, noise=0.05, factor=0.5, random_state=seed)
plot_data(X, y)
plt.show()
```



Then, we try to fit a custom SVM with linear kernel. Clearly, this classifier will not achieve good results.

In [5]:

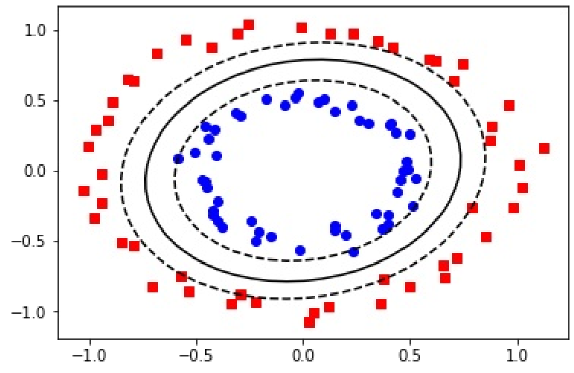
```
svm = SVC(C=100, kernel='linear', random_state=seed)
svm.fit(X, y)

plot_data(X, y)
plot_margins(svm, X, y)
plt.show()
```



However, if we apply a polynomial kernel of degree 2, we are able to learn the optimal decision boundary for this dataset.

In [6]:

```
svm = SVC(C=100, kernel='poly', degree=2, random_state=seed)
svm.fit(X, y)

plot_data(X, y)
plot_margins(svm, X, y)
plt.show()
```



The Kernel Trick consists of implicitly mapping a lower dimensional dataset, which is not linearly separable, to a higher dimensional space where the data becomes linearly separable.

In the above example, the standard linear kernel calculates the standard dot product as the similarity between two vectors $\mathbf{u}$ and $\mathbf{v}$. That is: $k(\mathbf{u}, \mathbf{v}) = \mathbf{u} \cdot \mathbf{v}$.

When we apply a polynomial kernel of degree 2, we are calculating the similarity between two vectors u and v as:
$k(\mathbf{u}, \mathbf{v}) = (\mathbf{u} \cdot \mathbf{v})^2$
$k(\mathbf{u}, \mathbf{v}) = (u_1v_1 + u_2v_2)^2$
$k(\mathbf{u}, \mathbf{v}) = u_1^2v_1^2 + 2u_1v_1u_2v_2 + u_2^2v_2^2$

The above calculation can be rewritten as:

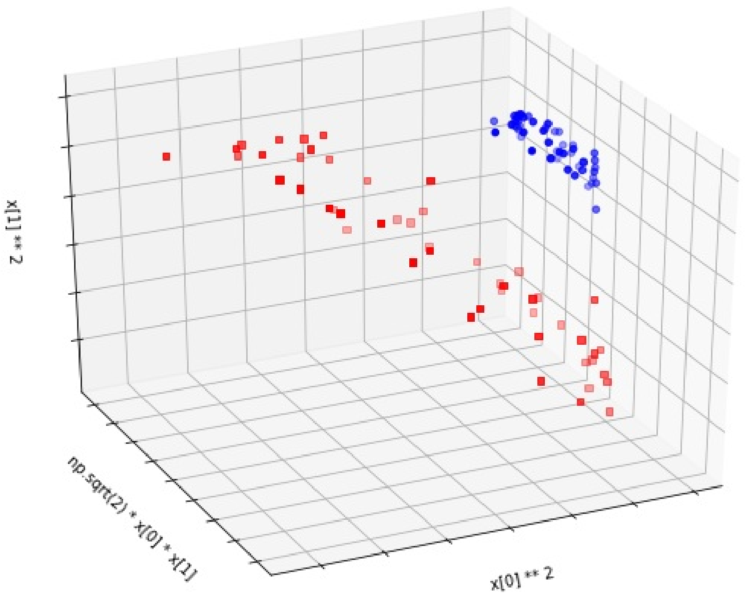
$$k(\mathbf{u}, \mathbf{v}) = \begin{bmatrix} u_1^2 \\ \sqrt{2}u_1u_2 \\ u_2^2 \end{bmatrix} \cdot \begin{bmatrix} v_1^2 \\ \sqrt{2}v_1v_2 \\ v_2^2 \end{bmatrix},$$

which is a dot product of two three dimensional vectors.

Finally, we will plot the original dataset in this new three dimensional space.

In [24]:

```
to3d = lambda x : [x[0] ** 2, np.sqrt(2) * x[0] * x[1], x[1] ** 2]
X_3D = np.array(list(map(to3d, X)))

fig = plt.figure(1, figsize=(8, 6))
ax = Axes3D(fig, elev=-150, azim=115)

class0 = np.where(y == 0)[0]
ax.scatter(X_3D[class0, 0], X_3D[class0, 1], X_3D[class0, 2], c='red', marker='s')

class1 = np.where(y == 1)[0]
ax.scatter(X_3D[class1, 0], X_3D[class1, 1], X_3D[class1, 2], c='blue', marker='o')

ax.set_xlabel('x[0] ** 2')
ax.set_ylabel('np.sqrt(2) * x[0] * x[1]')
ax.set_zlabel("x[1] ** 2")

ax.set_xticklabels([])
ax.set_yticklabels([])
ax.set_zticklabels([])

plt.show()
```



As it can be seen, the original dataset is linear separable in this new three dimensional space.

Practical example - Breast Cancer

Finally, as a last example, we will apply SVMs on the Breast Cancer dataset.

In [8]:

```
# Loading Breast Cancer dataset.
data = pd.read_csv('data/breast_cancer.csv')

# Creating a LabelEncoder and transforming the dataset labels.
le = LabelEncoder()
y = le.fit_transform(data['diagnosis'].values)

# Extracting the instances data.
X = data.drop('diagnosis', axis=1).values

# Splitting into train and test sets.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.34, stratify=y, random_state=seed)
```

Since this dataset has high dimensionality and probably is not linearly separable, we will apply a SVM with Radial Basis Function (RBF) kernel.

In [9]:

```
svm = SVC(kernel='rbf')
svm.fit(X_train, y_train)

y_pred = svm.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("SVM's accuracy score: {}".format(accuracy))
```

```
SVM's accuracy score: 0.6288659793814433
```

Unfortunately, SVMs are sensitive to data scale. Thus, we will standartize the dataset and train the SVM again.

In [10]:

```
scaler = StandardScaler()
scaler.fit(X_train)

# Normalizing train and test data.
X_train_scaled, X_test_scaled = scaler.transform(X_train), scaler.transform(X_test)

# Training SVM with normalized data.
svm.fit(X_train_scaled, y_train)

# Testing SVM with normalized data.
y_pred = svm.predict(X_test_scaled)
accuracy = accuracy_score(y_test, y_pred)
print("SVM's accuracy score: {}".format(accuracy))
```

```
SVM's accuracy score: 0.9845360824742269
```

Multiclass problems

SVMs were designed to deal with binary classification problems. Several approaches are available to deal with multiclass problems. Some of them are:

- One-vs-one classifiers: suppose the classification problem is composed by k classes. Thus, $k(k-1)/2$ SVMs are fitted, each one for a different pair of classes. For prediction, the class that received most of the votes is returned as output.
- One-vs-all classifiers: suppose the classification problem is composed by k classes. Then, k different classifiers are fitted, one for each class.

The sklearn.svm.SVC class implements the one-vs-one scheme.