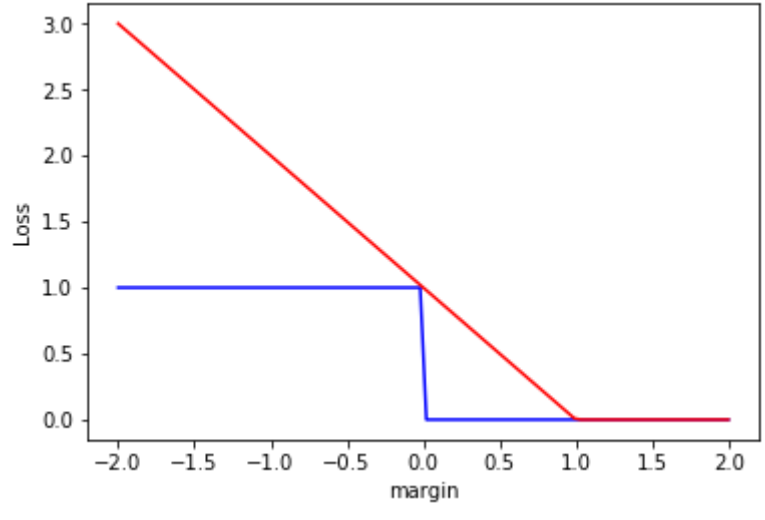
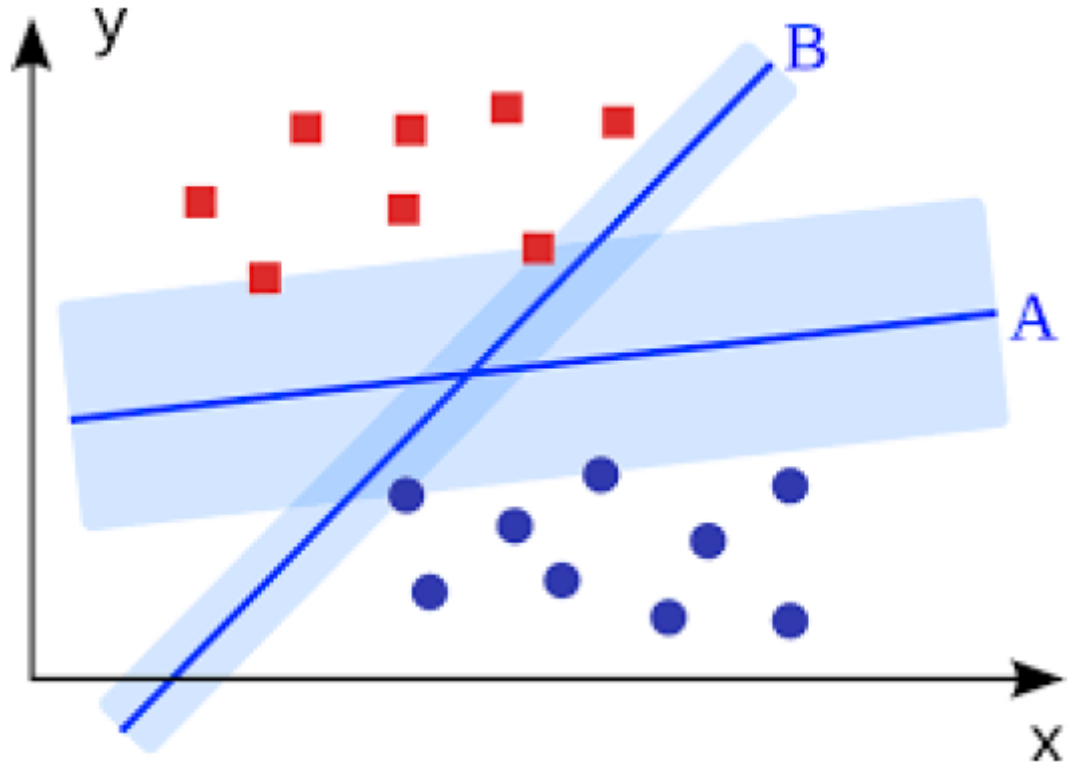
In [1]:
```
import IPython.display
IPython.display.display_latex(IPython.display.Latex(filename="macros.tex"))
```

SVM (Support Vector Machine)

In [2]:

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-20, 20, 100) / 10.0
plt.plot(X, [1 if x < 0 else 0 for x in X], color = [0, 0, 1], label="ER")
plt.plot(X, [0 if ((1 - x) < 0) else (1 - x) for x in X], color = [1, 0, 0], label="SVM"
)

plt.xlabel("margin")
plt.ylabel("Loss")

plt.show()
```




y
B
A
x

Если множество линейно разделимо.

$$\exists\, \theta, \theta_0, \ : \ \mathbb{M}_i(\theta, \theta_0) = y_i * (< \theta, x_i > - \theta_0) > 0, \ \forall i \in [1, \dots N]$$

$$\exists\, \theta, \theta_0, \epsilon \ : \ \mathbb{M}_i(\theta, \theta_0) = y_i * (< \theta, x_i > - \theta_0) \geq \epsilon, \ \forall i \in [1, \dots N]$$

$$\exists\, \theta, \theta_0 \ : \ \mathbb{M}_i(\theta, \theta_0) = y_i * (< \theta, x_i > - \theta_0) \geq 1, \ \forall i \in [1, \dots N]$$

разделяющая полоса:

$$\{x \mid -1 \leq (< \theta, x > -\theta_0) \leq 1\}$$

границы:

$$\begin{cases} < x_-, \theta > -\theta_0 = -1 \\ < x_+, \theta > -\theta_0 = 1 \end{cases}$$

Нормализуем:

$$\begin{cases} \frac{< x_-, \theta >}{\|\theta\|} - \frac{\theta_0 - 1}{\|\theta\|} = 0 \\ \frac{< x_+, \theta >}{\|\theta\|} - \frac{\theta_0 + 1}{\|\theta\|} = 0 \end{cases}$$

ширина полосы:

$$\frac{\theta_0 + 1}{\|\theta\|} - \frac{\theta_0 - 1}{\|\theta\|} = \frac{2}{\|\theta\|}$$

Задача минимизации:

$$\begin{cases} ||\theta|| \to min \\ \mathbb{M}_i(\theta, \theta_0) \geq 1 \end{cases}$$

Мы имеем минимизацию на выпуклом множестве. Решеная проблема линейного программирования (по условиям Куна-Таккера).

Эта задача сводится к минимизации квадратичной функции.

$\theta = \sum_{i=1}^{N} \lambda_i y_i * x_i$

$\theta_0 = \theta * x_i - y_i$

$\lambda \geq 0$

x_i для которых $\lambda_i > 0$ называются опорными векторами.

Опорные вектора - это вектора через которые проходит граница, остальные не используются в формуле($\lambda = 0$).

Если множество линейно не разделимо.

Задача минимизации:

$$\begin{cases} \|\theta\| + C\sum_{i=1}^{N}\xi_i \to min \\ \mathbb{M}_i(\theta, \theta_0) \geq 1 - \xi_i \end{cases}$$

$\xi_i \geq 0$ - error

$\theta = \sum_{i=1}^{N}\lambda_i y_i * x_i$

$\theta_0 = \theta * x_i - y_i$

$\lambda \geq 0$

$$\alpha(x) = sign\big(\sum_{i=1}^{N}\lambda_i y_i * < x_i, x > -\theta_0\big)$$

Non linear svm

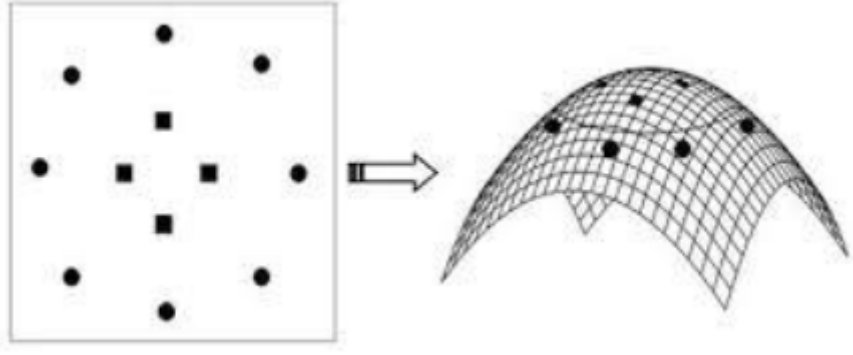
Kernel trick. Мы переводим пространство ($\hat{X}$) в другое пространство (H) где классы (могут быть) разделимы

Для SVM использовать довольно легко.

Скалярное произведение $< x_1, x_2 >$ заменяем на $K(x_1, x_2)$

Считаем что существует $\phi : \hat{X} \to H$ такой что

$$K(x_1, x_2) =< \phi(x_1), \phi(x_2) >_H$$



Наиболее популярные ядра:

- $K(x_1, x_2) =< x_1, x_2 >^d$
- $K(x_1, x_2) = (\gamma < x_1, x_2 > + r)^d$ - polynomial
- $K(x_1, x_2) = exp(-\gamma \|x_1 - x_2\|^2)$ - rbf

SVM:

- Решается задача квадратичного программирования, имеющая единственное решение.
- Уверенная классификация из-за наиболее широкой полосы.
- Метод опорных векторов неустойчив по отношению к шуму в исходных данных(RVM).
- Нет алгоритма подбора ядра (открытая задача).
- Необходимо подобрать параметр C.

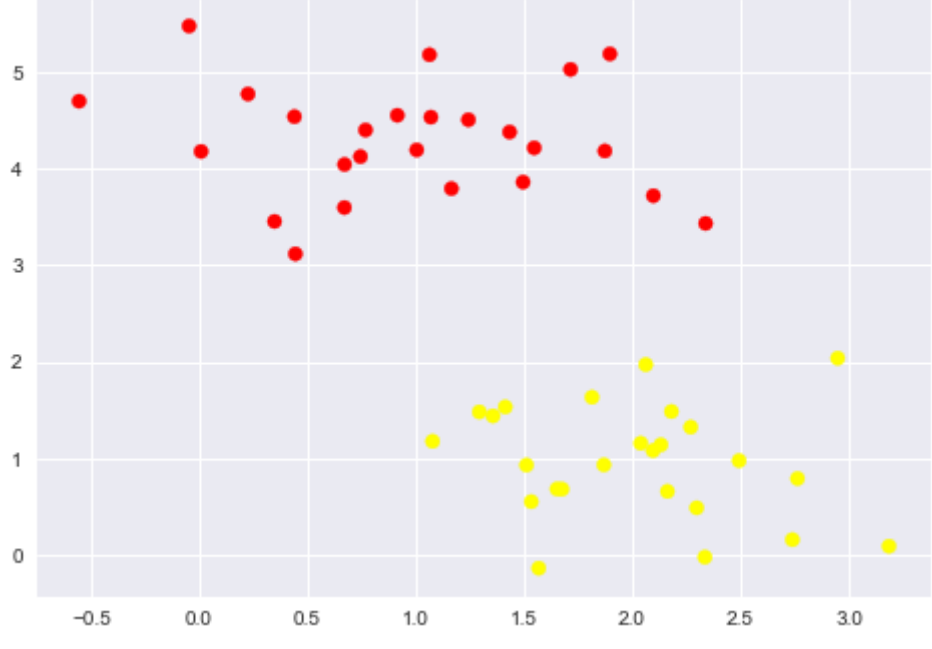
Code examples from https://github.com/jakevdp/PythonDataScienceHandbook (https://github.com/jakevdp/PythonDataScienceHandbook)

In [3]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import seaborn as sns; sns.set
```

Out[3]:
```
<function seaborn.rcmod.set>
```

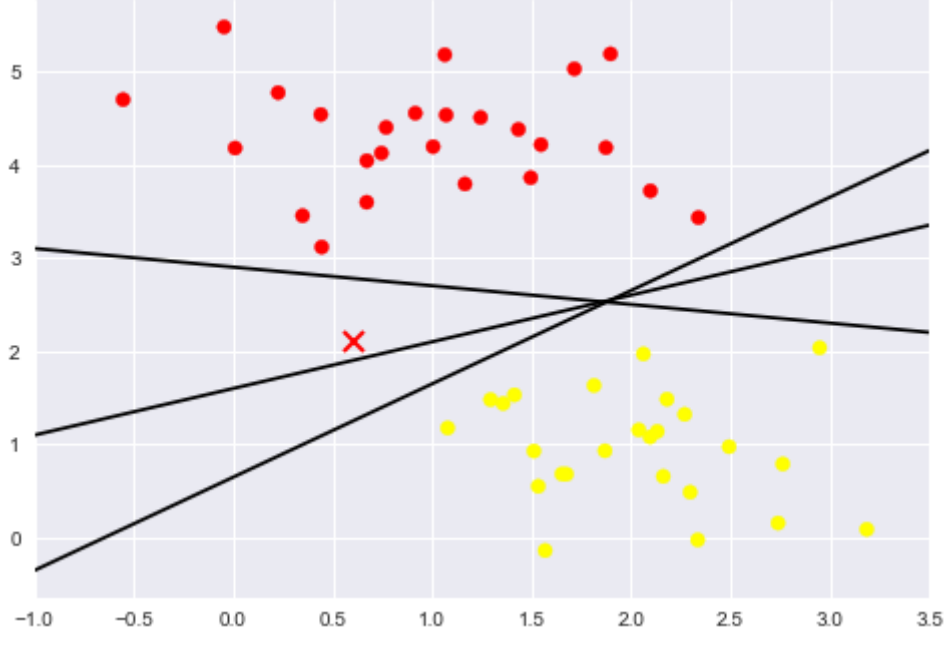
In [4]:
```
from sklearn.datasets.samples_generator import make_blobs

X, y = make_blobs(n_samples=50, centers=2, random_state=0, cluster_std=0.6)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
```

Out[4]: <matplotlib.collections.PathCollection at 0x1af5d2fad68>



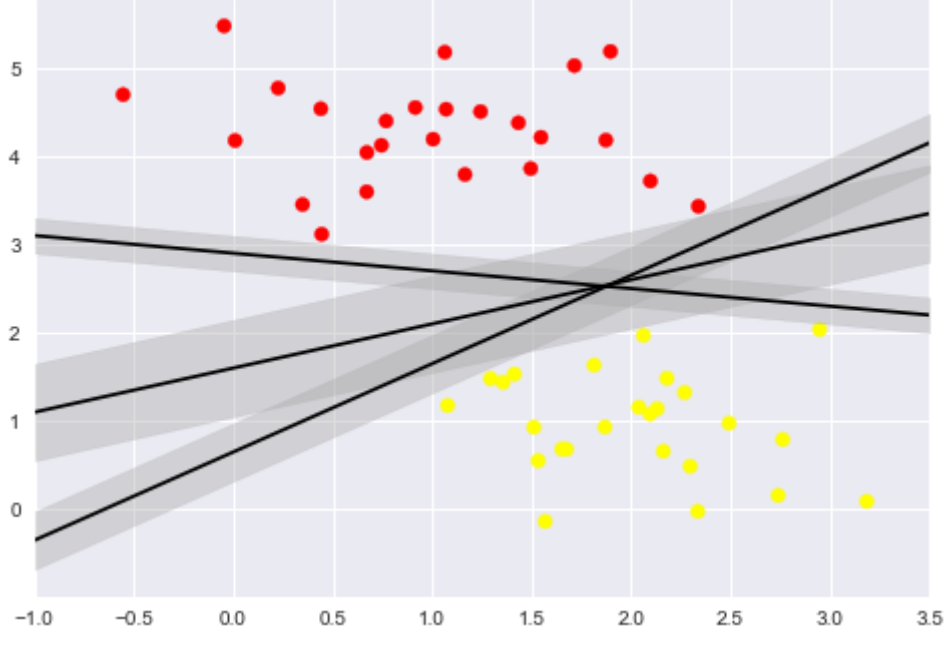
In [5]:
```
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plt.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2, markersize=10)

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)
```

Out[5]:
```
(-1, 3.5)
```



In [6]:

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

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);
```



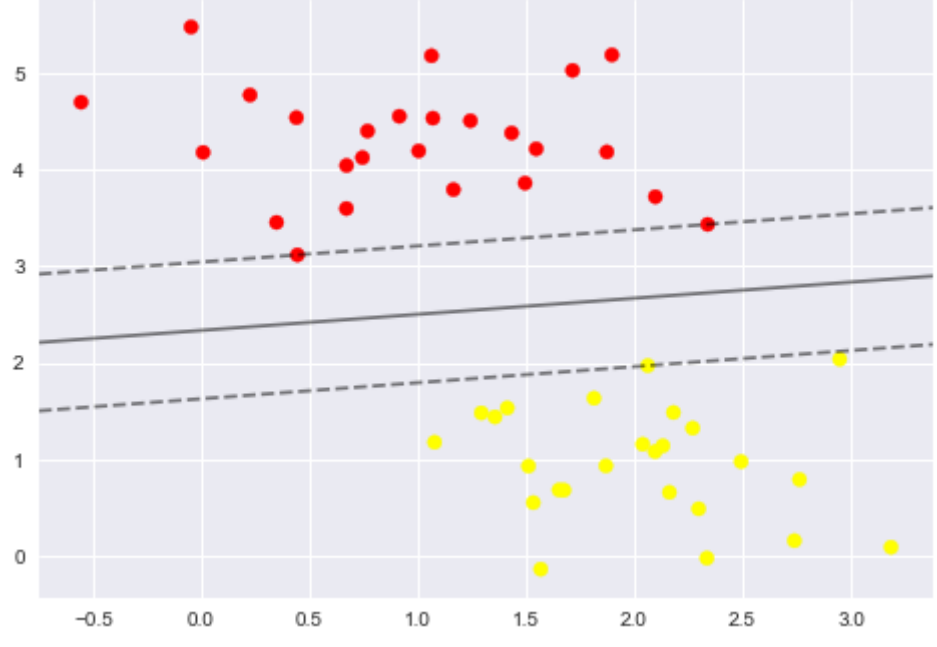
In [7]:
```
from sklearn.svm import SVC # "Support vector classifier"
model = SVC(kernel='linear', C=1E10)
model.fit(X, y)
```

Out[7]:
```
SVC(C=10000000000.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma='auto', kernel='linear',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

In [8]:

```
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, facecolors='none');
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
```

In [9]:
```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(model);
```

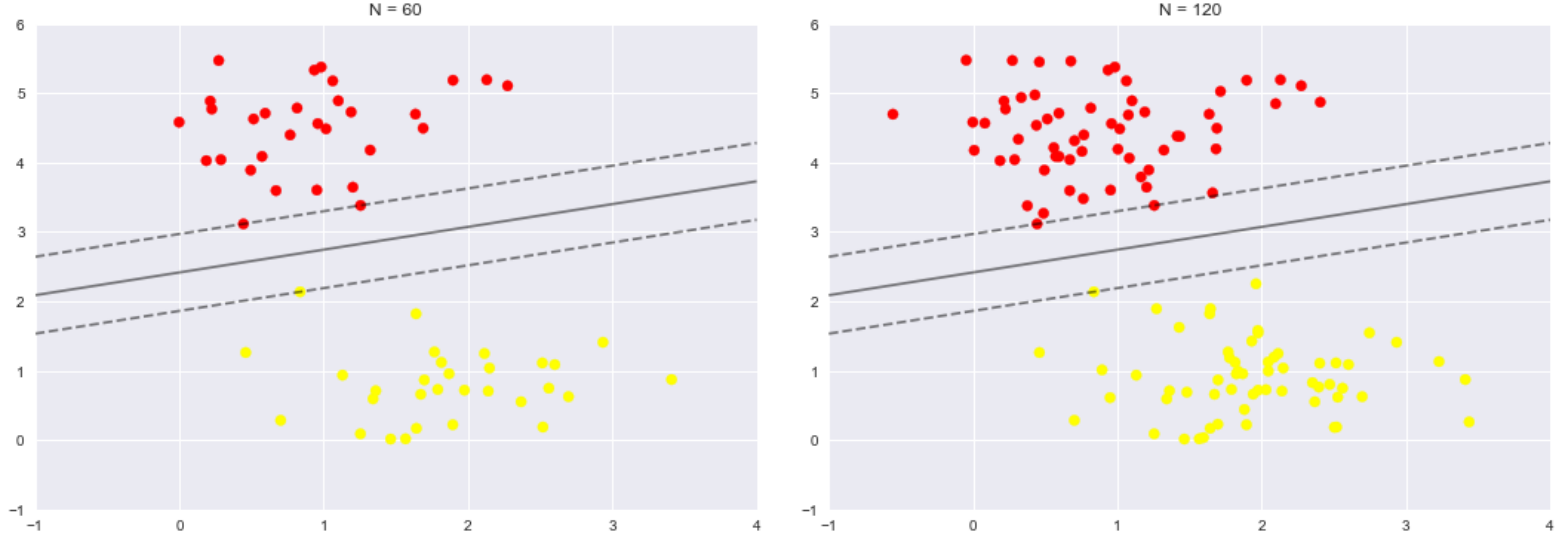


In [10]:
```
model.support_vectors_
```

Out[10]:
```
array([[ 0.44359863,  3.11530945],
       [ 2.33812285,  3.43116792],
       [ 2.06156753,  1.96918596]])
```

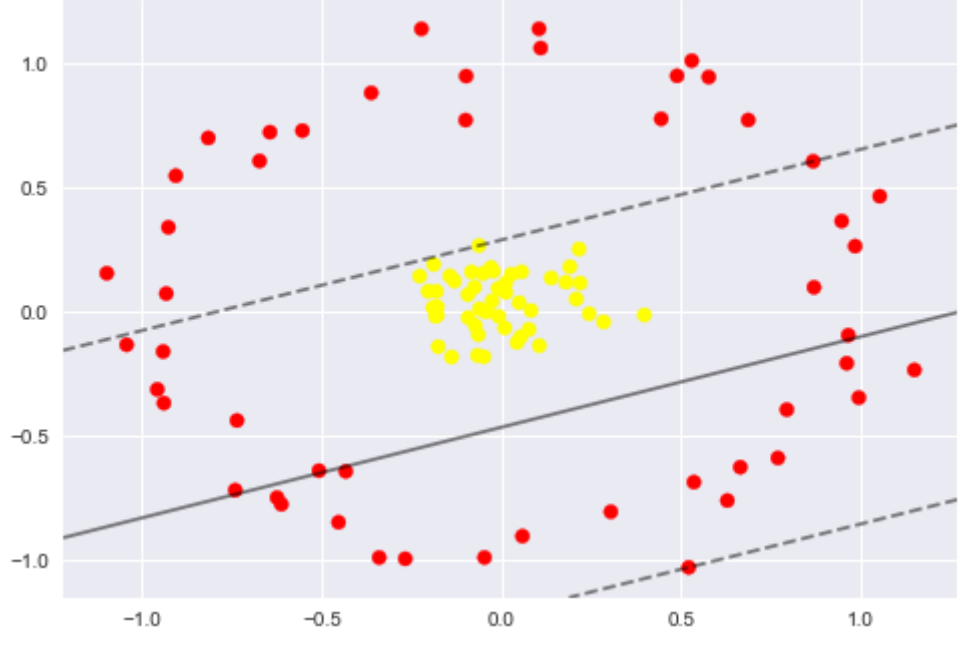
In [11]:

```
def plot_svm(N=10, ax=None):
    X, y = make_blobs(n_samples=200, centers=2,
                      random_state=0, cluster_std=0.60)
    X = X[:N]
    y = y[:N]
    model = SVC(kernel='linear', C=1E10)
    model.fit(X, y)

    ax = ax or plt.gca()
    ax.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
    ax.set_xlim(-1, 4)
    ax.set_ylim(-1, 6)
    plot_svc_decision_function(model, ax)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
for axi, N in zip(ax, [60, 120]):
    plot_svm(N, axi)
    axi.set_title('N = {0}'.format(N))
```



In [12]:

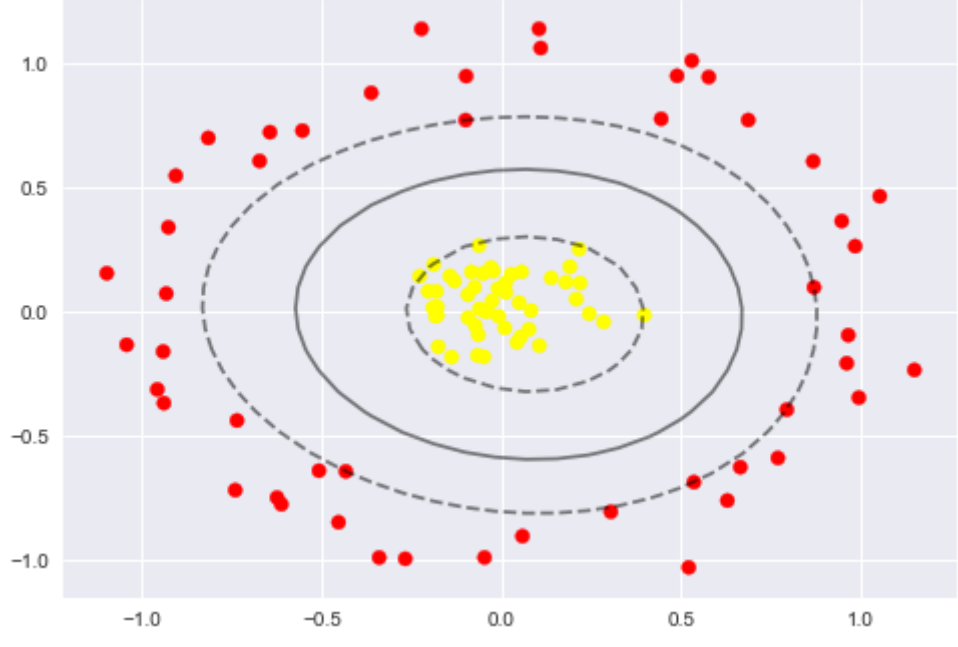
```
from sklearn.datasets.samples_generator import make_circles
X, y = make_circles(100, factor=.1, noise=.1)

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);
```



In [13]:
```
clf = SVC(kernel='rbf', C=1E6)
clf.fit(X, y)
```

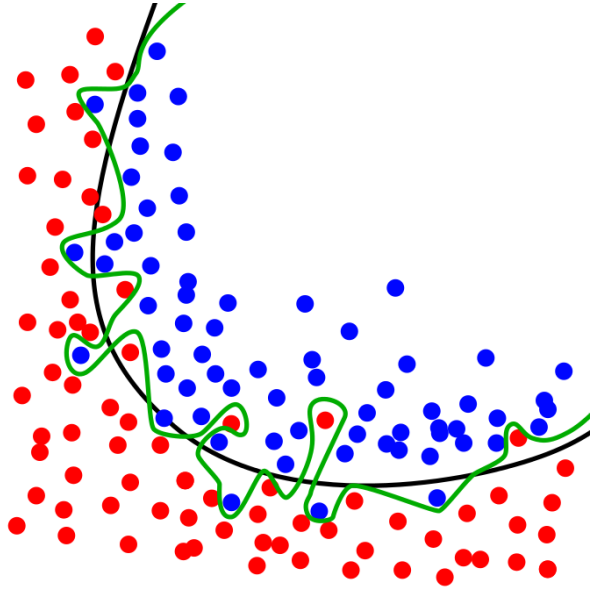
Out[13]:
```
SVC(C=1000000.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

In [14]:
```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(clf)
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
            s=300, lw=1, facecolors='none');
```



regularization

overfitting in algorithm:
very good at train set and a lot of errors in real (population)



regularization - процесс введения дополнительной информации для предотвращения переобучения
Регуляризация в линейных классификаторах:

$$Q + \lambda * ||\theta||^p \rightarrow min$$

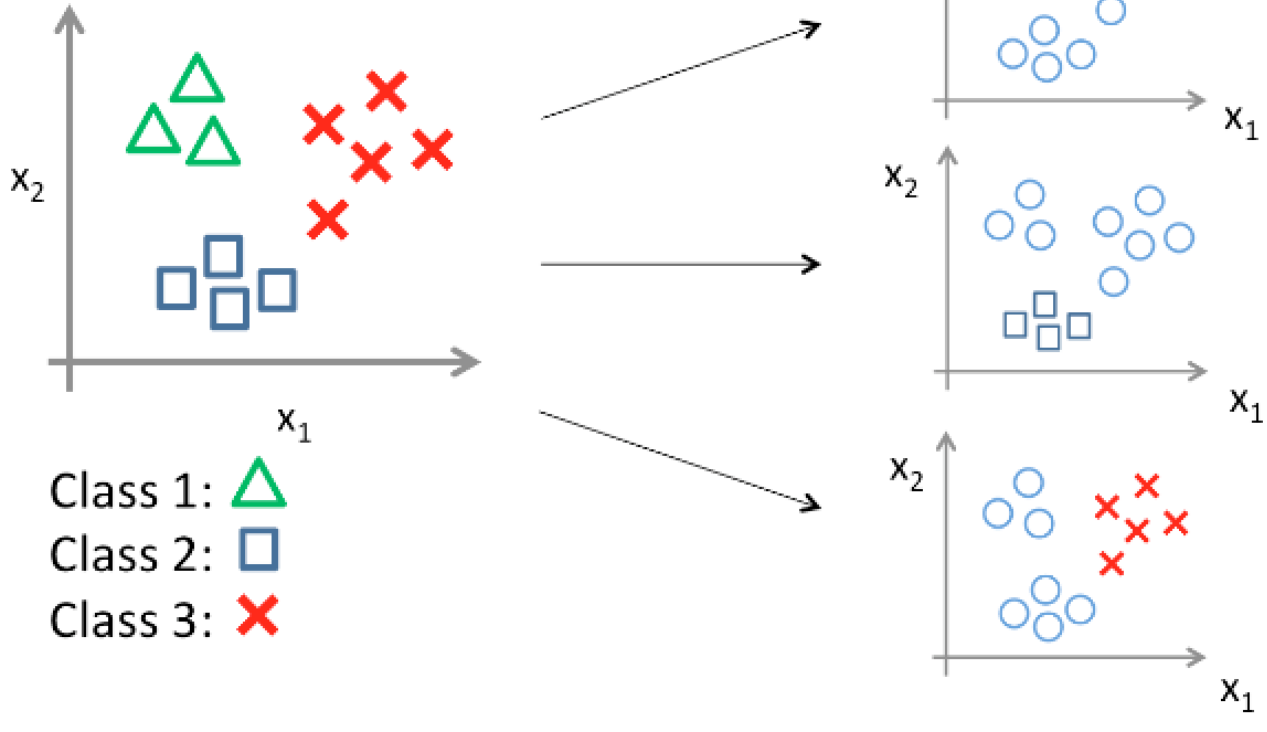
Добавление штрафного слагаемого $\lambda * ||\theta||^p$ снижает риск переобучения и повышает устойчивость вектора весов по отношению к малым изменениям обучающей выборки.

Степень регуляризатора p определяет класс методов оптимизации.

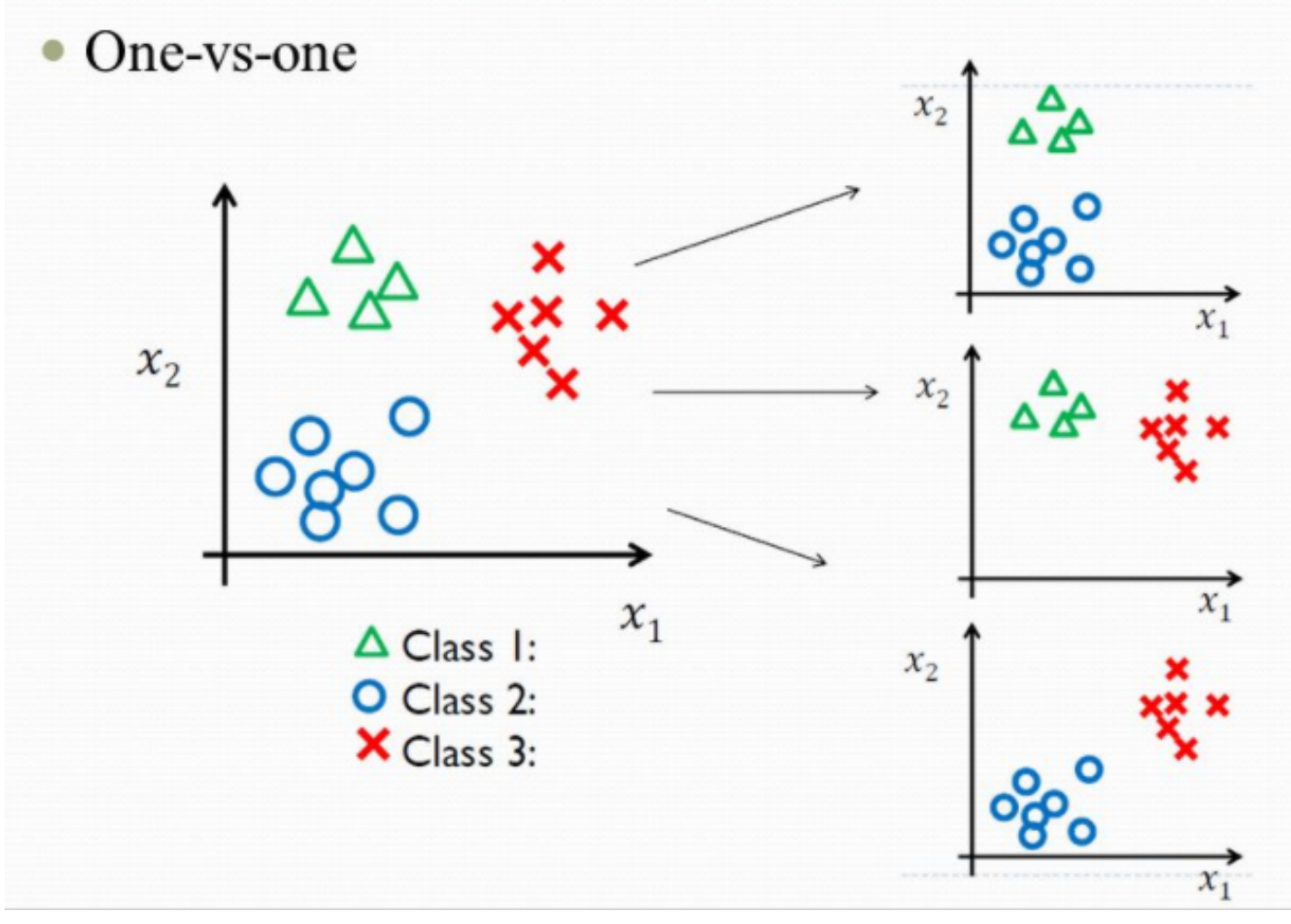
- $p = 2$ и гладком (по параметру θ) функционале Q можно применять стандартные градиентные методы минимизации.
- $p = 1$ и выпуклом функционале Q возникает задача выпуклого программирования с ограничениями типа неравенств. В результате её решения часть коэффициентов θ обнуляются, что фактически означает отсев неинформативных признаков.

from binary classification to several class

One-vs-all (one-vs-rest):



- One-vs-one



one-vs-all:

$$\alpha(x) = argmax_{c \in C}(b_c(x))$$

one-vs-one(all-vs-all):

$$\alpha(x) = argmax_{c \in C} \sum_{c` \in C, c' \neq c} b_{cc`}(x)$$

one-vs-all:

- $|C|$ классификаторов
- несбалансированно

one-vs-all:

- $|C| * (|C| - 1)$ классификаторов
- сбалансированно (если было сбалансированно)