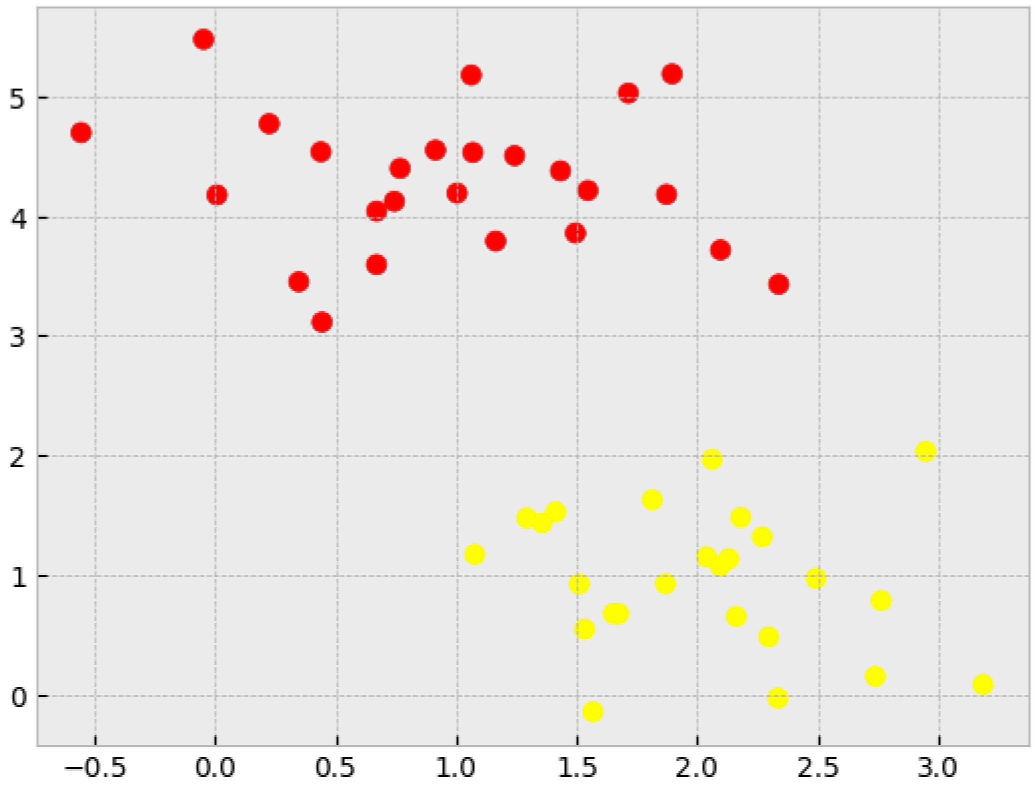
Lecture 15 - Support Vector Machines (SVMs)

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('bmh')
```

In [14]:
```
from sklearn.datasets import make_blobs

X, t = make_blobs(n_samples=50, centers=2,
                  random_state=0, cluster_std=0.6)

plt.scatter(X[:, 0], X[:, 1], c=t, s=50, cmap='autumn');
```

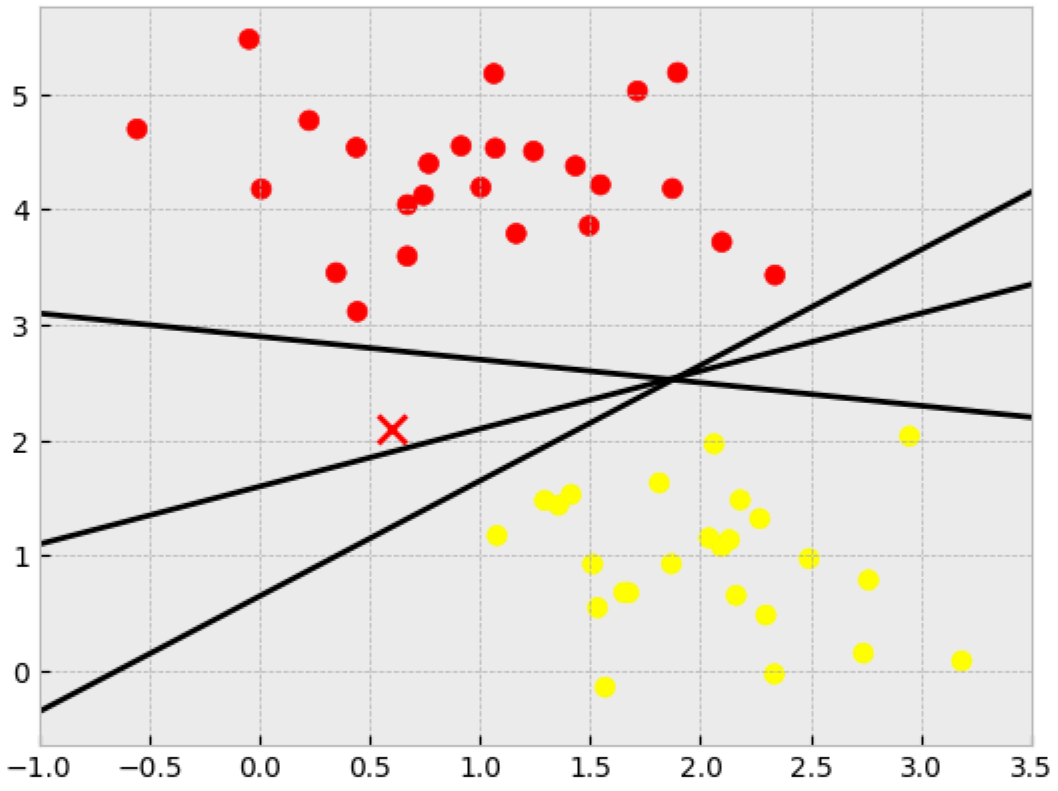


A linear discriminative classifier would attempt to draw a straight line separating the two sets of data, and thereby create a model for classification. For two dimensional data like that shown here, this is a task we could do by hand. But immediately we see a problem: there is more than one possible dividing line that can perfectly discriminate between the two classes!

We can draw them as follows:

In [18]:
```
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=t, s=50, cmap='autumn')
plt.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2, markersize=10)
```

```python
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);
```



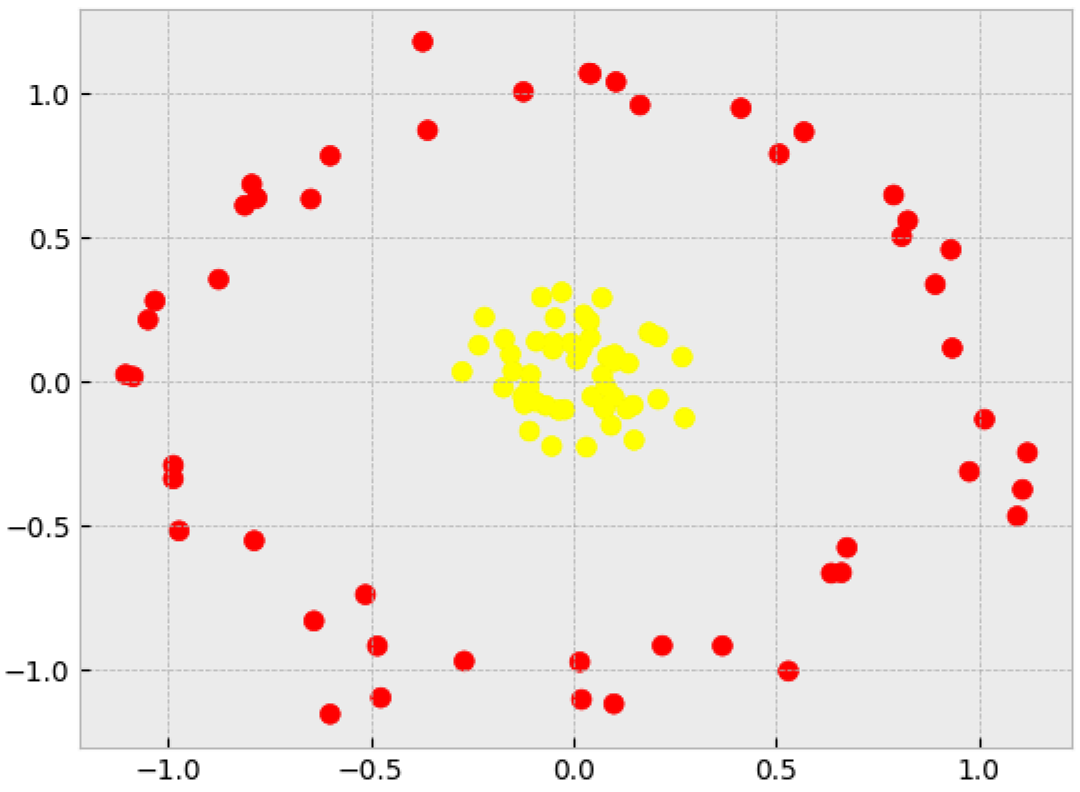
These are three very different separators which, nevertheless, perfectly discriminate between these samples. Depending on which you choose, a new data point (e.g., the one marked by the "X" in this plot) will be assigned a different label! Evidently our simple intuition of "drawing a line between classes" is not enough, and we need to think a bit deeper.

Kernel Machine

Consider the following data set:

Kernel methods = algorithms that learn using similarity instead of raw features.

In [30]:
```python
from sklearn.datasets import make_circles

X, t = make_circles(100, factor=.1, noise=.1)
plt.scatter(X[:, 0], X[:, 1], c=t, s=50, cmap='autumn');
```



The two classes are clearly not linearly separable. What would you do?

In [31]:
```
X
```

```
Out[31]:  array([[ 0.8927245 ,  0.33601474],
                 [ 0.10527674,  1.03901798],
                 [-0.0673822 , -0.08460478],
                 [ 0.08103995, -0.01948111],
                 [ 0.2092276 , -0.06282096],
                 [-0.10582555,  0.02321962],
                 [-0.97236841, -0.51945031],
                 [-0.35950523,  0.87096828],
                 [-0.1081843 , -0.17350213],
                 [-0.12299629,  1.00455899],
                 [-0.12108286, -0.07915284],
                 [ 0.0255289 ,  0.23055124],
                 [ 0.66143367, -0.66437803],
                 [-0.05059962,  0.11069242],
                 [ 0.16454949,  0.95835909],
                 [ 0.03864141,  1.06903335],
                 [ 0.09939573,  0.06744614],
                 [-0.27431473,  0.03309375],
                 [ 0.41362164,  0.94774613],
                 [-0.12450096, -0.05342397],
                 [ 0.02131359,  0.11176827],
                 [-0.07809523,  0.29181148],
                 [ 0.93132881,  0.45669752],
                 [-0.78701741, -0.55340599],
                 [-0.98677074, -0.33873165],
                 [-1.08525664,  0.01598213],
                 [ 0.09999004, -1.11988288],
                 [-0.05135623,  0.13488142],
                 [ 0.00790517,  0.07457203],
                 [ 0.5689361 ,  0.86533141],
                 [ 0.04619653, -0.05303618],
                 [-1.03181499,  0.27873396],
                 [ 0.3676589 , -0.91824279],
                 [-0.09315109, -0.07095709],
                 [ 0.0438118 ,  1.0675465 ],
                 [ 0.07687705, -0.09616826],
                 [ 0.10051218, -0.05500768],
                 [-0.10899962, -0.01999875],
                 [ 0.07583983, -0.08889118],
                 [-0.02852994,  0.30934303],
                 [ 0.03972629,  0.20797006],
                 [-0.63968868, -0.8320334 ],
                 [-0.03621608, -0.10010089],
                 [ 0.13197595, -0.09648902],
                 [-0.15487685,  0.09358478],
                 [ 0.20833168,  0.15446938],
                 [ 0.67495348, -0.57752172],
                 [ 0.53058575, -1.00616731],
                 [-0.04476437,  0.21884856],
                 [-1.10442541,  0.02342619],
                 [-0.26914757, -0.97173397],
                 [-0.05325799, -0.22556692],
                 [ 0.07095382,  0.28929034],
                 [-0.59974334, -1.15552214],
                 [ 0.63622775, -0.66620971],
                 [-0.1699839 ,  0.14594268],
                 [-0.87472684,  0.35349313],
                 [ 0.07156616,  0.02087018],
                 [-0.17242947, -0.02201338],
                 [ 0.02024289, -1.10476415],
                 [-0.21914691,  0.22221076],
                 [ 0.18660895,  0.16933113],
                 [-0.02142224, -0.09844417],
                 [-0.78050624,  0.63545945],
                 [ 0.04179371,  0.1502402 ],
                 [ 0.26882083,  0.0840351 ],
                 [ 0.08430074,  0.08421185],
                 [ 1.11981614, -0.24776616],
                 [-0.64739348,  0.63244004],
                 [ 0.0324829 , -0.22862143],
                 [-0.37142012,  1.17843992],
                 [-0.14904393,  0.03567769],
                 [ 0.09325388, -0.15352772],
                 [ 0.01586967, -0.97473385],
                 [ 0.79006193,  0.64627197],
                 [-0.47482473, -1.09851561],
                 [ 0.13592671,  0.06236304],
                 [ 0.82483249,  0.55705346],
                 [-0.59959047,  0.78243168],
                 [ 0.14760019, -0.08370778],
                 [-0.23313847,  0.12489943],
                 [-0.98657382, -0.29191479],
                 [-0.09115689,  0.13816581],
                 [-0.81058653,  0.60994702],
                 [ 0.15000552, -0.20430323],
                 [ 1.09520783, -0.46832381],
                 [-0.51280441, -0.74052147],
                 [ 0.93479902,  0.11501766],
                 [ 0.21941043, -0.91762254],
                 [ 0.10182663,  0.09291358],
                 [-1.0488793 ,  0.21320436],
                 [ 0.27440015, -0.12751141],
                 [-0.48334601, -0.91887153],
                 [ 1.01426274, -0.13225321],
                 [ 0.81016749,  0.50274297],
                 [-0.79305991,  0.68383054],
                 [ 1.10789386, -0.37593141],
                 [-0.00571204,  0.13249264],
                 [ 0.97679214, -0.3137839 ],
                 [ 0.50768404,  0.78897012]])
```

In [32]:
```
t
```

Out[32]:
```
array([0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1,
       0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1,
       1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1,
       1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0,
       0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0], dtype=int64)
```

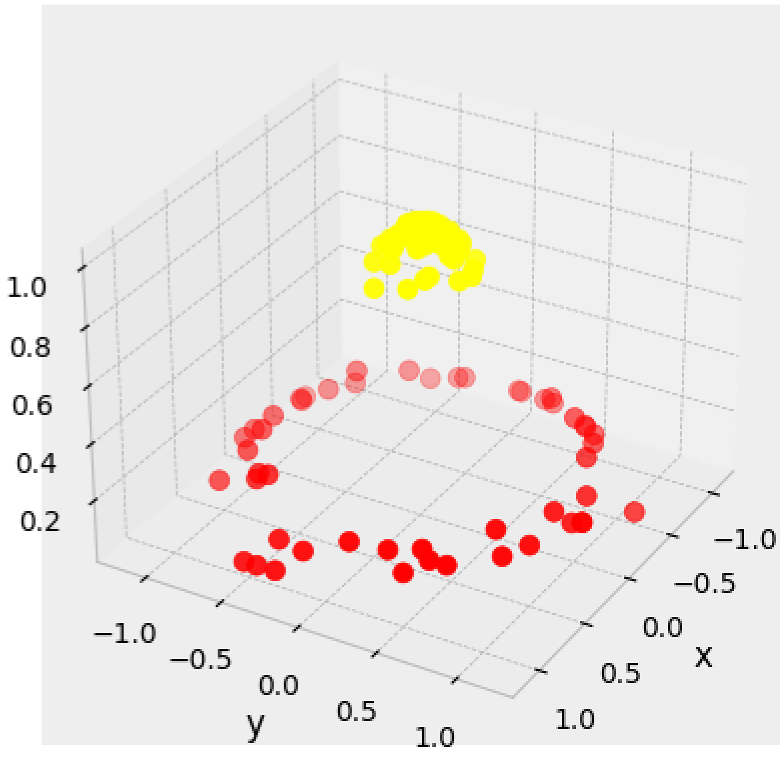
In [38]:
```
r = np.exp(-((X - X.mean(axis=0))** 2/(2*X.std(axis=0, ddof=1)**2)).sum(axis=1))
```

ri=exp(−1/2∑_j(x_ij−μ_j)^2/ σ_j^2) This produces a value between 0 and 1, where:

Samples close to the mean → r ≈ 1

Samples far from the mean → r ≈ 0

In [46]:
```
from mpl_toolkits import mplot3d

def plot_3D(elev=30, azim=30, X=X, y=t):
    ax = plt.subplot(projection='3d')
    ax.scatter3D(X[:, 0], X[:, 1], r, c=t, s=50, cmap='autumn')
    ax.view_init(elev=elev, azim=azim)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('r')

plot_3D();
```



Kernel Function

The kernel function is defined as:

$$\begin{aligned} k : \mathbb{R}^d \times \mathbb{R}^d &\longrightarrow \mathbb{R} \\ (\mathbf{x}, \mathbf{y}) &\longmapsto k(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^T \phi(\mathbf{y}) \end{aligned}$$

where $\phi(x)$ is a fixed nonlinear feature space mapping

$$\begin{aligned} \phi : \mathbb{R}^d &\longrightarrow \mathbb{R}^{\mathbb{D}} \\ \mathbf{x} &\longmapsto \phi(\mathbf{x}) \end{aligned}$$

where $D > d$.

A popular kernel function is the **radial basis function** (RBF kernel):

$$k_{\text{RBF}}(\mathbf{x}, \mathbf{y}) = \exp\left(-\gamma \|\mathbf{x} - \mathbf{y}\|^2\right)$$

typical values for $\gamma = \frac{1}{2\sigma^2}$. As you can see, the RBF kernel is measures the **similarity** between two points as a function of the kernnel *bandwidth* σ.

Here is a **super simple, beginner-friendly explanation** of everything shown in your screenshot — no math background needed.

🌟 What is a Kernel Function? (Newbie Explanation)

A **kernel** is just a *similarity calculator*.

It tells you **how similar two data points are**.

Examples in real life:

- Face recognition compares two faces → similarity
- Spotify recommends songs similar to ones you like
- Google Photos groups similar objects together

Kernel functions do the same thing for **machine learning data**.

🧠 Why do we need kernels?

Some data is **not linearly separable**. For example, circles inside circles.

A kernel lets us: ✔ Pretend the data is in a **higher-dimensional space** ✔ Without ever **actually** doing the transformation

This trick is called the **Kernel Trick**.

🔧 The Math in the Image (Explained Simply)

### 1️⃣ The kernel function is written as:

[k(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^T\phi(\mathbf{y})]

This looks scary, but here’s the easy version:

👉 Meaning:

Instead of comparing x and y directly, we first **convert** them into a more complex feature space using:

[\phi(x)]

Think of ϕ(x) as a transformation like:

🟦 “Take the original features and generate more complicated ones.”

But **you don't have to compute ϕ(x)**. The kernel function **does it for you automatically**.

2️⃣ What is ϕ(x)?

[\phi: \mathbb{R}^d \rightarrow \mathbb{R}^D]

This simply means:

- Your original data has **d features**
- The transformed space has **D features**, where **D is much bigger**

Example: A 2D point (x, y) might become a huge feature vector after transformation.

But the kernel trick avoids doing this manually.

🌈 The RBF Kernel — Most Popular One

The document shows the RBF kernel:

[k_{\text{RBF}}(x,y) = \exp\left(-\gamma |x - y|^2\right)]

❗ SUPER SIMPLE MEANING:

- If two points are **close** → kernel value ≈ **1**
- If they are **far** → kernel value ≈ **0**

This is exactly like asking:

"How close are x and y?"

It's like a **smooth distance measure**.

🎨 Why Does This Work?

Because instead of saying:

> "Draw a straight line to separate classes"

You can say:

> "Use similarities between points to create curved boundaries."

That's why RBF kernels create **curvy decision boundaries**.

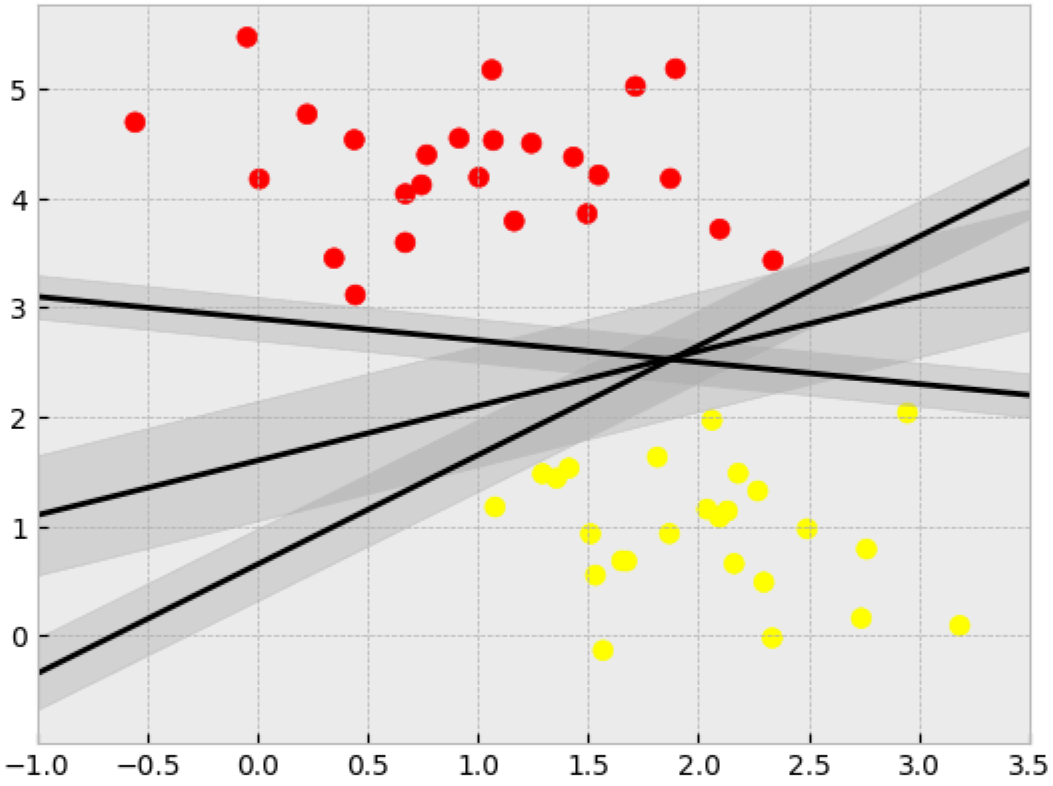
🟢 Summary (For Newbies)

Concept	Easy Explanation
Kernel	A function that tells how similar two data points are.
ϕ(x)	A fancy transformation to a higher-dimensional space (you never compute it).
Kernel Trick	Compute similarity *as if* data was transformed — without doing the work.
RBF Kernel	A smooth similarity measure based on distance.
Why use it?	Helps ML models learn nonlinear patterns.

Support Vector Machines: Maximizing the Margin

SVMs 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. Here is an example of how this might look:

In [47]:
```
X, t = make_blobs(n_samples=50, centers=2,
                  random_state=0, cluster_std=0.60)
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=t, 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 support vector machines, the line that maximizes this margin is the one we will choose as the optimal model. Support vector machines are an example of such a maximum margin estimator.

Support Vector Machine: Separable Classes

Let's start with the two-class linearly separable task and then we will extend the method to more general cases where data are not separable. Let $\phi(x_i)$, $i = 1, 2, \ldots, N$, be the feature vectors of the training set, X, and corresponding target values $t_1, t_2, \cdots, t_N$ where $t_n \in \{-1, 1\}$. These belong to either of two classes, C_1, C_2, which are *assumed to be linearly separable*.

The goal, once more, is to design a hyperplane

$$y(x) = w^T \phi(x) + b = 0$$

that classifies correctly all the training vectors.

Because the training data is linearly separable in the feature space, by definition there exists at least one choice of the parameters w and b such that $y(x)$ satisfies $y(x_n) > 0$ for points having $t_n = +1$ and $y(x_n) < 0$ for points having $t_n = -1$, so that $t_n y(x_n) > 0$ for all training data points.

Such a hyperplane is not unique. The perceptron algorithm may converge to any one of the possible solutions. Having gained in experience, this time we will be more demanding.

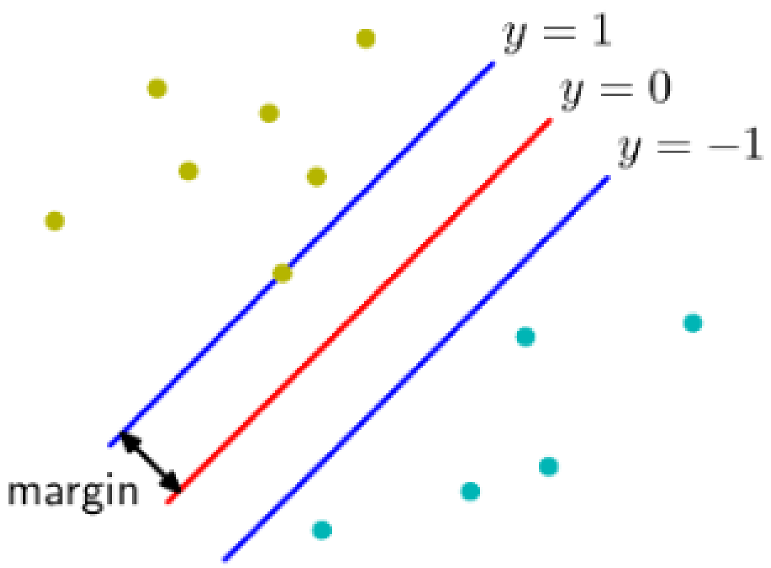
- Which hyperplane would any sensible engineer choose as the classifier for operation in practice, where data outside the training set will be fed to it?
- Once again, the hyperplane that leaves more “room” on either side, so that data in both classes can move a bit more freely, with less risk of causing an error.
- A sensible choice for the hyperplane classifier would be the one that leaves the maximum margin from both classes.
- Thus such a hyperplane can be trusted more, when it is faced with the challenge of operating with unknown data.
    - It has a higher generalizarion performance.

Quantifying the "Margin"

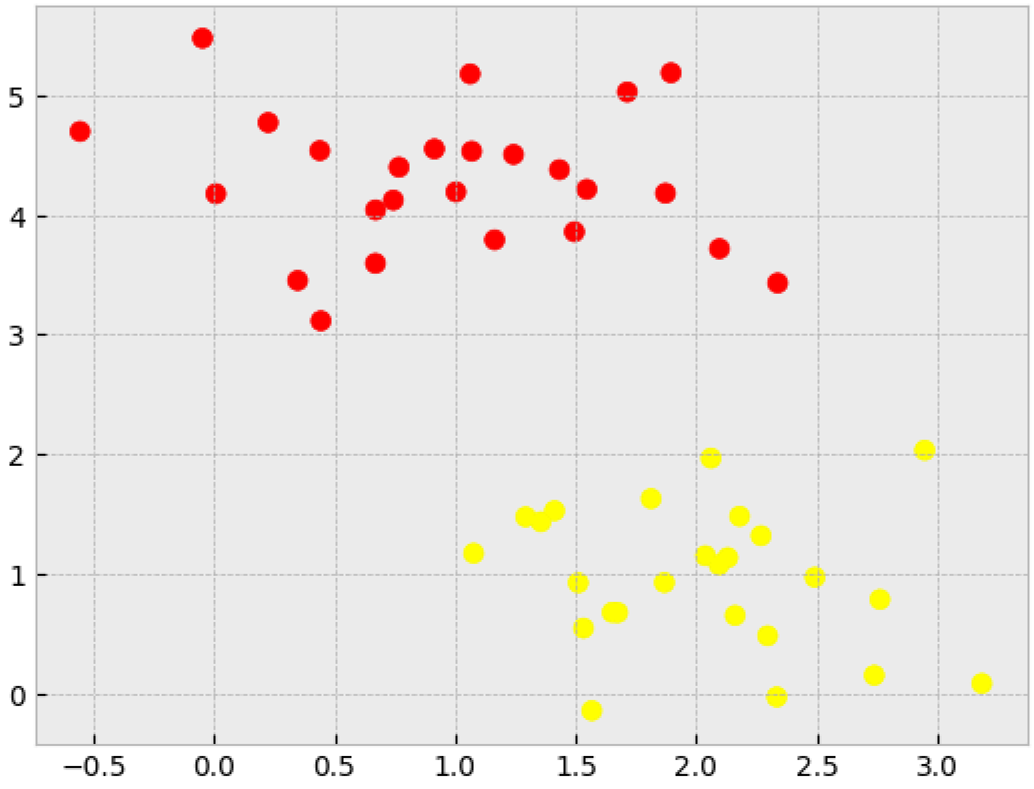
Let us now quantify the term "margin" that a hyperplane leaves from both classes. Every hyperplane is characterized by its direction (determined by w) and its exact position in space (determined by b). Since we want to give no preference to either of the classes, then it is reasonable for each direction to select that hyperplane which has the same distance from the respective nearest points in c_1 and c_2.

In [49]:
```
from IPython.display import Image
Image('image.png', width=400)
#Source: Bishop, Pattern Recognition and Machine Learning, 2006
```

Out[49]:



In [50]:
```
X, t = make_blobs(n_samples=50, centers=2,
                  random_state=0, cluster_std=0.60)

plt.scatter(X[:, 0], X[:, 1], c=t, s=50, cmap='autumn');
```



In [52]:
```
from sklearn.svm import SVC # "Support vector classifier"

SVC?
```

```
Init signature:
SVC(
    *,
    C=1.0,
    kernel='rbf',
    degree=3,
    gamma='scale',
    coef0=0.0,
    shrinking=True,
    probability=False,
    tol=0.001,
    cache_size=200,
    class_weight=None,
    verbose=False,
    max_iter=-1,
    decision_function_shape='ovr',
    break_ties=False,
    random_state=None,
)
Docstring:
C-Support Vector Classification.

The implementation is based on libsvm. The fit time scales at least
quadratically with the number of samples and may be impractical
beyond tens of thousands of samples. For large datasets
consider using :class:`~sklearn.svm.LinearSVC` or
:class:`~sklearn.linear_model.SGDClassifier` instead, possibly after a
:class:`~sklearn.kernel_approximation.Nystroem` transformer or
other :ref:`kernel_approximation`.

The multiclass support is handled according to a one-vs-one scheme.

For details on the precise mathematical formulation of the provided
kernel functions and how `gamma`, `coef0` and `degree` affect each
other, see the corresponding section in the narrative documentation:
:ref:`svm_kernels`.

To learn how to tune SVC's hyperparameters, see the following example:
:ref:`sphx_glr_auto_examples_model_selection_plot_nested_cross_validation_iris.py`

Read more in the :ref:`User Guide <svm_classification>`.

Parameters
----------
C : float, default=1.0
    Regularization parameter. The strength of the regularization is
    inversely proportional to C. Must be strictly positive. The penalty
    is a squared l2 penalty.

kernel : {'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'} or callable,          d
efault='rbf'
    Specifies the kernel type to be used in the algorithm. If
    none is given, 'rbf' will be used. If a callable is given it is used to
    pre-compute the kernel matrix from data matrices; that matrix should be
    an array of shape ``(n_samples, n_samples)``. For an intuitive
    visualization of different kernel types see
    :ref:`sphx_glr_auto_examples_svm_plot_svm_kernels.py`.

degree : int, default=3
    Degree of the polynomial kernel function ('poly').
    Must be non-negative. Ignored by all other kernels.

gamma : {'scale', 'auto'} or float, default='scale'
    Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

    - if ``gamma='scale'`` (default) is passed then it uses
      1 / (n_features * X.var()) as value of gamma,
    - if 'auto', uses 1 / n_features
    - if float, must be non-negative.

    .. versionchanged:: 0.22
       The default value of ``gamma`` changed from 'auto' to 'scale'.

coef0 : float, default=0.0
    Independent term in kernel function.
    It is only significant in 'poly' and 'sigmoid'.

shrinking : bool, default=True
    Whether to use the shrinking heuristic.
    See the :ref:`User Guide <shrinking_svm>`.

probability : bool, default=False
    Whether to enable probability estimates. This must be enabled prior
    to calling `fit`, will slow down that method as it internally uses
    5-fold cross-validation, and `predict_proba` may be inconsistent with
    `predict`. Read more in the :ref:`User Guide <scores_probabilities>`.

tol : float, default=1e-3
    Tolerance for stopping criterion.

cache_size : float, default=200
    Specify the size of the kernel cache (in MB).

class_weight : dict or 'balanced', default=None
    Set the parameter C of class i to class_weight[i]*C for
    SVC. If not given, all classes are supposed to have
    weight one.
    The "balanced" mode uses the values of y to automatically adjust
    weights inversely proportional to class frequencies in the input data
    as ``n_samples / (n_classes * np.bincount(y))``.

verbose : bool, default=False
    Enable verbose output. Note that this setting takes advantage of a
    per-process runtime setting in libsvm that, if enabled, may not work
    properly in a multithreaded context.

max_iter : int, default=-1
    Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape : {'ovo', 'ovr'}, default='ovr'
    Whether to return a one-vs-rest ('ovr') decision function of shape
    (n_samples, n_classes) as all other classifiers, or the original
    one-vs-one ('ovo') decision function of libsvm which has shape
    (n_samples, n_classes * (n_classes - 1) / 2). However, note that
    internally, one-vs-one ('ovo') is always used as a multi-class strategy
    to train models; an ovr matrix is only constructed from the ovo matrix.
    The parameter is ignored for binary classification.

    .. versionchanged:: 0.19
        decision_function_shape is 'ovr' by default.

    .. versionadded:: 0.17
       *decision_function_shape='ovr'* is recommended.

    .. versionchanged:: 0.17
       Deprecated *decision_function_shape='ovo' and None*.

break_ties : bool, default=False
    If true, ``decision_function_shape='ovr'``, and number of classes > 2,
    :term:`predict` will break ties according to the confidence values of
    :term:`decision_function`; otherwise the first class among the tied
    classes is returned. Please note that breaking ties comes at a
    relatively high computational cost compared to a simple predict.

    .. versionadded:: 0.22

random_state : int, RandomState instance or None, default=None
    Controls the pseudo random number generation for shuffling the data for
    probability estimates. Ignored when `probability` is False.
    Pass an int for reproducible output across multiple function calls.
    See :term:`Glossary <random_state>`.

Attributes
----------
class_weight_ : ndarray of shape (n_classes,)
    Multipliers of parameter C for each class.
    Computed based on the ``class_weight`` parameter.

classes_ : ndarray of shape (n_classes,)
    The classes labels.

coef_ : ndarray of shape (n_classes * (n_classes - 1) / 2, n_features)
    Weights assigned to the features (coefficients in the primal
    problem). This is only available in the case of a linear kernel.

    `coef_` is a readonly property derived from `dual_coef_` and
    `support_vectors_`.

dual_coef_ : ndarray of shape (n_classes -1, n_SV)
    Dual coefficients of the support vector in the decision
    function (see :ref:`sgd_mathematical_formulation`), multiplied by
    their targets.
    For multiclass, coefficient for all 1-vs-1 classifiers.
    The layout of the coefficients in the multiclass case is somewhat
    non-trivial. See the :ref:`multi-class section of the User Guide
    <svm_multi_class>` for details.

fit_status_ : int
```

```
        0 if correctly fitted, 1 otherwise (will raise warning)

    intercept_ : ndarray of shape (n_classes * (n_classes - 1) / 2,)
        Constants in decision function.

    n_features_in_ : int
        Number of features seen during :term:`fit`.

        .. versionadded:: 0.24

    feature_names_in_ : ndarray of shape (`n_features_in_`,)
        Names of features seen during :term:`fit`. Defined only when `X`
        has feature names that are all strings.

        .. versionadded:: 1.0

    n_iter_ : ndarray of shape (n_classes * (n_classes - 1) // 2,)
        Number of iterations run by the optimization routine to fit the model.
        The shape of this attribute depends on the number of models optimized
        which in turn depends on the number of classes.

        .. versionadded:: 1.1

    support_ : ndarray of shape (n_SV)
        Indices of support vectors.

    support_vectors_ : ndarray of shape (n_SV, n_features)
        Support vectors. An empty array if kernel is precomputed.

    n_support_ : ndarray of shape (n_classes,), dtype=int32
        Number of support vectors for each class.

    probA_ : ndarray of shape (n_classes * (n_classes - 1) / 2)
    probB_ : ndarray of shape (n_classes * (n_classes - 1) / 2)
        If `probability=True`, it corresponds to the parameters learned in
        Platt scaling to produce probability estimates from decision values.
        If `probability=False`, it's an empty array. Platt scaling uses the
        logistic function
        ``1 / (1 + exp(decision_value * probA_ + probB_))``
        where ``probA_`` and ``probB_`` are learned from the dataset [2]_. For
        more information on the multiclass case and training procedure see
        section 8 of [1]_.

    shape_fit_ : tuple of int of shape (n_dimensions_of_X,)
        Array dimensions of training vector ``X``.

    See Also
    --------
    SVR : Support Vector Machine for Regression implemented using libsvm.

    LinearSVC : Scalable Linear Support Vector Machine for classification
        implemented using liblinear. Check the See Also section of
        LinearSVC for more comparison element.

    References
    ----------
```

```
.. [1] `LIBSVM: A Library for Support Vector Machines
    <http://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf>`_

.. [2] `Platt, John (1999). "Probabilistic Outputs for Support Vector
    Machines and Comparisons to Regularized Likelihood Methods"
    <https://citeseerx.ist.psu.edu/doc_view/pid/42e5ed832d4310ce4378c44d05570439df28
a393>`_

Examples
--------
>>> import numpy as np
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.preprocessing import StandardScaler
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from sklearn.svm import SVC
>>> clf = make_pipeline(StandardScaler(), SVC(gamma='auto'))
>>> clf.fit(X, y)
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('svc', SVC(gamma='auto'))])

>>> print(clf.predict([[-0.8, -1]]))
[1]
File:           c:\users\hp\appdata\local\programs\python\python312\lib\site-package
s\sklearn\svm\_classes.py
Type:           ABCMeta
Subclasses:
```

In [53]:

```
svm = SVC(kernel='linear')
svm.fit(X, t)
```

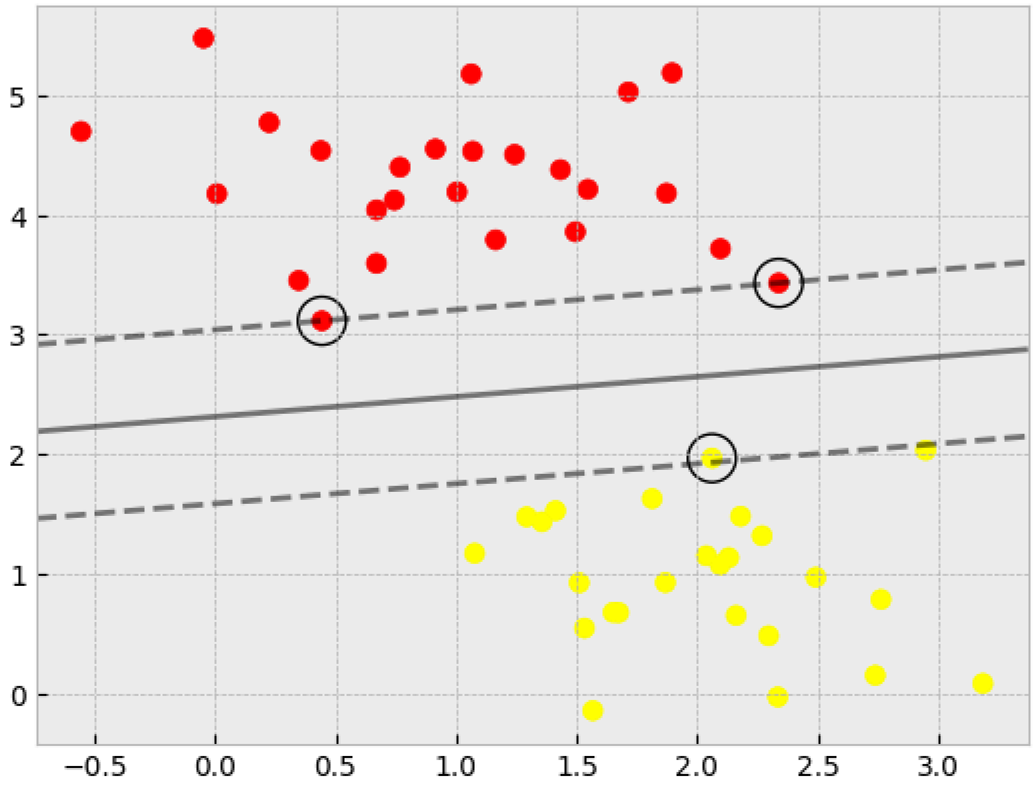
Out[53]:

```
SVC
SVC(kernel='linear')
```

In [54]:

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

In [55]:
```
plt.scatter(X[:, 0], X[:, 1], c=t, s=50, cmap='autumn')
plot_svc_decision_function(svm);
```

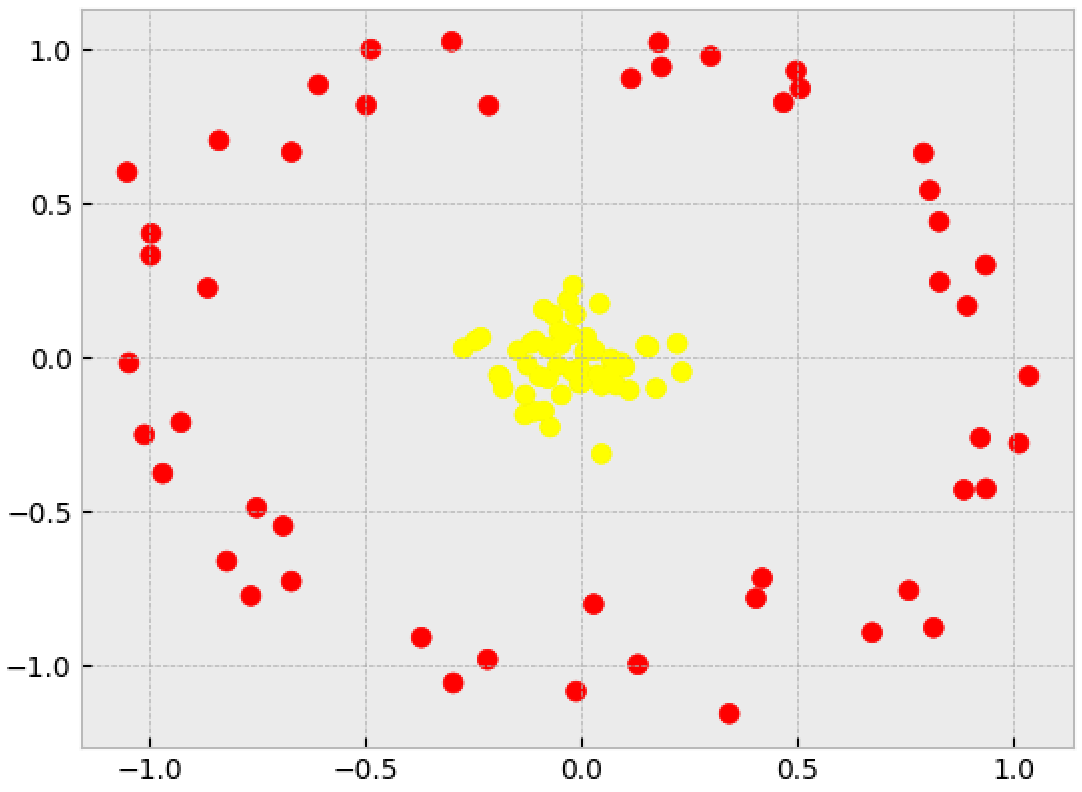


In [56]:
```
## support vectors

svm.support_vectors_
```

Out[56]:
```
array([[0.44359863, 3.11530945],
       [2.33812285, 3.43116792],
       [2.06156753, 1.96918596]])
```

What if the classes are not linearly separable?

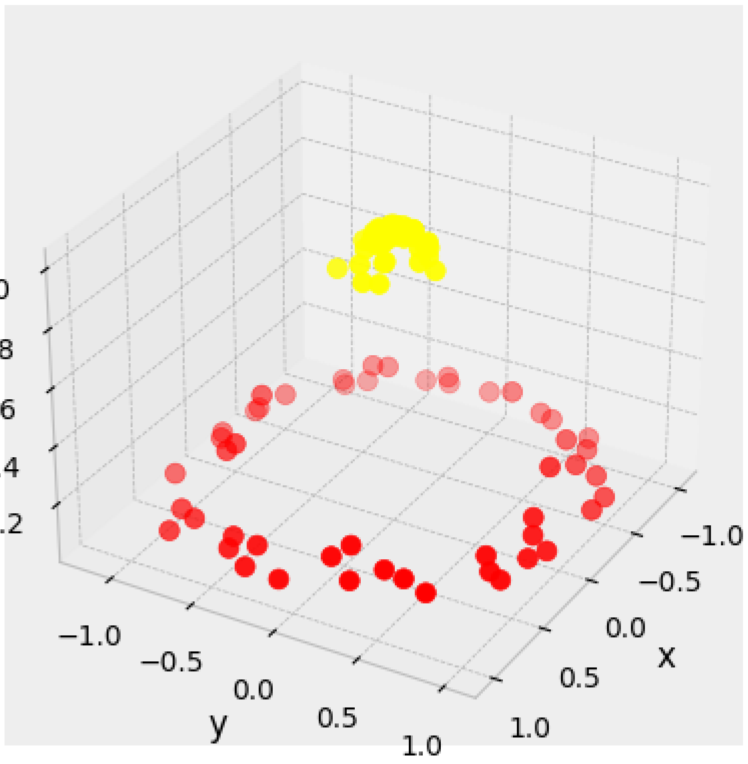
In [57]:
```
from sklearn.datasets import make_circles

X, t = make_circles(100, factor=.1, noise=.1)
plt.scatter(X[:, 0], X[:, 1], c=t, s=50, cmap='autumn');
```



The two classes are clearly not linearly separable. What would you do?

```
In [58]: r = np.exp(-((X - X.mean(axis=0))** 2/(2*X.std(axis=0, ddof=1)**2)).sum(axis=1))
```

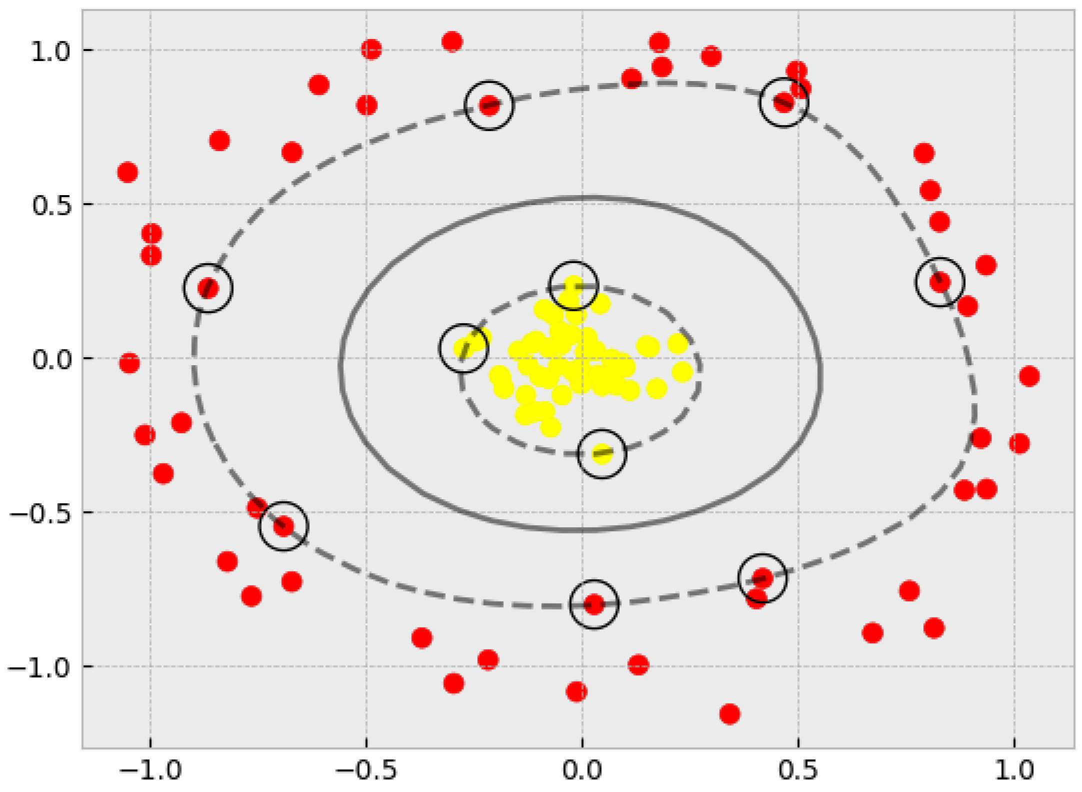
```
In [59]: from mpl_toolkits import mplot3d

def plot_3D(elev=30, azim=30, X=X, y=t):
    ax = plt.subplot(projection='3d')
    ax.scatter3D(X[:, 0], X[:, 1], r, c=t, s=50, cmap='autumn')
    ax.view_init(elev=elev, azim=azim)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('r')

plot_3D();
```



In [12]:
```
from sklearn.svm import SVC

# SVC stands for "Support vector classifier"
```

In [61]:
```
# Helper function
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='black',facecolors='none');
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
```

In [ ]:
```
svm = SVC(kernel='rbf', C=1E6) #1E6 = 1×106 = 1,000,000
svm.fit(X, t)

plt.scatter(X[:, 0], X[:, 1], c=t, s=50, cmap='autumn')
plot_svc_decision_function(svm)
plt.scatter(svm.support_vectors_[:, 0], svm.support_vectors_[:, 1],
            s=300, lw=1, facecolors='none');
```



Measuring the Distance of Points to the Discriminant Function

Recall that the perpendicular distance of a point x from a hyperplane defined by $y(x) = 0$ where $y(x)$ takes the form $y(x) = w^T \phi(x) + w_0$ is given by $\frac{|y(x)|}{||w||}$.

Furthermore, we are only interested in solutions for which all data points are **correctly classified**, so that $t_n y(x_n) > 0, \forall n$. Thus the distance of a point x_n to the decision surface is given by

$$\frac{t_n y(x_n)}{||w||} = \frac{t_n \left(w^t \phi(x_n) + w_0\right)}{||w||}$$

The margin is given by the perpendicular distance to the closest point x_n from the data set, and we wish to optimize the parameters w and b in order to maximize this distance. Thus the maximum margin solution is found by solving

$$\arg_{w,b} \max \left\{ \frac{1}{||w||} \min_n \left[t_n (w^T \phi(x_n) + b) \right] \right\}$$

where the factor $\frac{1}{||w||}$ is taken outside the optimization over n because w does not depend on n.

- Direct solution of this optimization problem would be very complex, and so we shall convert it into an equivalent problem that is much easier to solve.
- To do this we note that if we make the rescaling $w \to \kappa w$ and $b \to \kappa b$, then the distance from any point x_n to the decision surface, given by $\frac{t_n y(x_n)}{||w||}$, is unchanged. We can use this freedom to set

$$t_n(w^T\phi(x_n) + b) = 1$$

for the point that is closest to the surface, called the **support vectors**. In this case, all data points will satisfy the constraints

$$t_n(w^T\phi(x_n) + b) \geq 1, n = 1, 2, \dots, N$$

- This is known as the canonical representation of the decision hyperplane.
- In the case of data points for which the equality holds (support vectors), the constraints are said to be active, whereas for the remainder they are said to be inactive.
- By definition, there will always be at least one active constraint, because there will always be a closest point, and once the margin has been maximized there will be at least two active constraints.

The optimization problem then simply requires that we maximize $||w||^{-1}$, which is equivalent to minimizing $||w||^2$, and so we have to solve the optimization problem

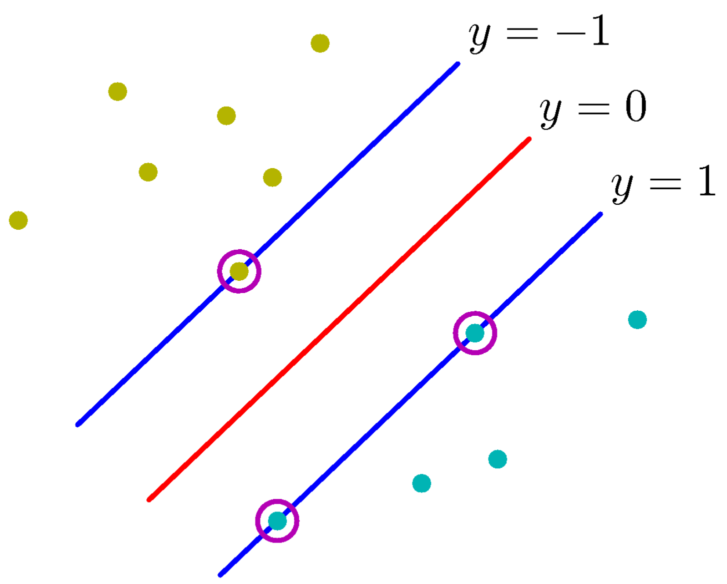
$$\begin{aligned}&\arg_{w,b}\min \frac{1}{2}||w||^2 \\ &\text{subject to } t_n(w^T\phi(x_n) + b) \geq 1\end{aligned}$$

Support Vectors (SV)

Note that a small subset of training points (from either class) will determine where this discriminant function will be placed. The points lying exactly on the margin, on either side of the discriminant function are called the **support vectors**. They are the closest points to the discriminant function $y(x)$ are the *representatives* for each class.

In [2]:
```python
Image('figures/Figure7.1b.png', width=400)
#Source: Bi/absshop, Pattern Recognition and Machine Learning, 2006
```

```
Out[2]:
```



The support vectors (SVs) are the only training points that are needed to defined the discriminant function $y(x) = \mathbf{w}^T\phi(x) + w_0$. And thus, once the SVs are identified from the training data, any other point that is **not** an SV can be **discarded**.

During **inference**, SVMs perform a similarity measure between test points are the **support vectors**. If the test point is most similar with SVs from C_1, then it will be classified as C_1, otherwise it will be classified as C_2.

Measuring Similarity

SVMs use kernel functions to mesure similarity between pairs of points. These kernel functions are defined as the inner product of feature transformation $\phi(x)$. Of course, not all feature transformations will lead to a well-defined kernel function.

- SVMs works with kernel functions equipped with an inner product only.

Kernel Function

The kernel function is defined as:

$$\begin{aligned} k : \mathbb{R}^d \times \mathbb{R}^d &\longrightarrow \mathbb{R} \\ (\mathbf{x}, \mathbf{y}) &\longmapsto k(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^T\phi(\mathbf{y}) \end{aligned}$$

where $\phi(x)$ is a fixed nonlinear feature space mapping

$$\begin{aligned} \phi : \mathbb{R}^d &\longrightarrow \mathbb{R}^{\mathbb{D}} \\ \mathbf{x} &\longmapsto \phi(\mathbf{x}) \end{aligned}$$

where $D > d$.

An inner product is a function that takes two inputs from a space and outputs a single value by combining the inputs, and is expressed as $\langle \mathbf{x}, \mathbf{y} \rangle$. For example, the dot product between two vectors: $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{d} x_i y_i$.

A **kernel** is a function that calculates the inner product in an enlarged feature space using only values from the original feature space, and is expressed as $K(\mathbf{x}, \mathbf{y})$. For example, using the mapping $\phi(\mathbf{x})$ below,

$$\begin{aligned} \phi : \mathbb{R}^2 &\longrightarrow \mathbb{R}^3 \\ [x_1, x_2] &\longmapsto [x_1^2, \sqrt{2}x_1x_2, x_2^2] \end{aligned}$$

the related kernel for the dot product in the enlarged space is

$$\begin{aligned} K((x_1, x_2), (y_1, y_2)) &= \langle (x_1^2, \sqrt{2}x_1x_2, x_2^2), (y_1^2, \sqrt{2}y_1y_2, y_2^2) \rangle \\ &= x_1^2x_2^2 + 2x_1x_2y_1y_2 + y_1^2y_2^2 \\ &= (x_1x_2 + y_1y_2)^2 \end{aligned}$$

- The inner product of two instances is calculated in the feature space.
- A mapping sends these instances to the enlarged space. $[x_1, x_2]$ maps to $[x_1^2, \sqrt{2}x_1x_2, x_2^2]$ and $[y_1, y_2]$ maps to $[y_1^2, \sqrt{2}y_1y_2, y_2^2]$.
- The inner product between the instances is calculated in the enlarged space.
- The kernel, $K(\mathbf{x}, \mathbf{y})$, encodes this mapping and then the inner product. Sometimes the kernel is calculated directly from the feature space.

Kernels used in support vector classification typically measure the similarity or distance between instances. The most commonly used kernels for support vector classifiers are listed below.

Common Kernels

1. **Polynomial kernel**

$$K(\mathbf{x}, \mathbf{y}) = (1 + \langle \mathbf{x}, \mathbf{y} \rangle)^d$$

Instances are mapped into an enlarged space of polynomial powers similar to the example at the section's start. The decision boundary will curve some.

2. **Radial Basis Function (RBF) kernel**

$$K(\mathbf{x}, \mathbf{y}) = e^{-\gamma \|\mathbf{x} - \mathbf{y}\|^2}$$

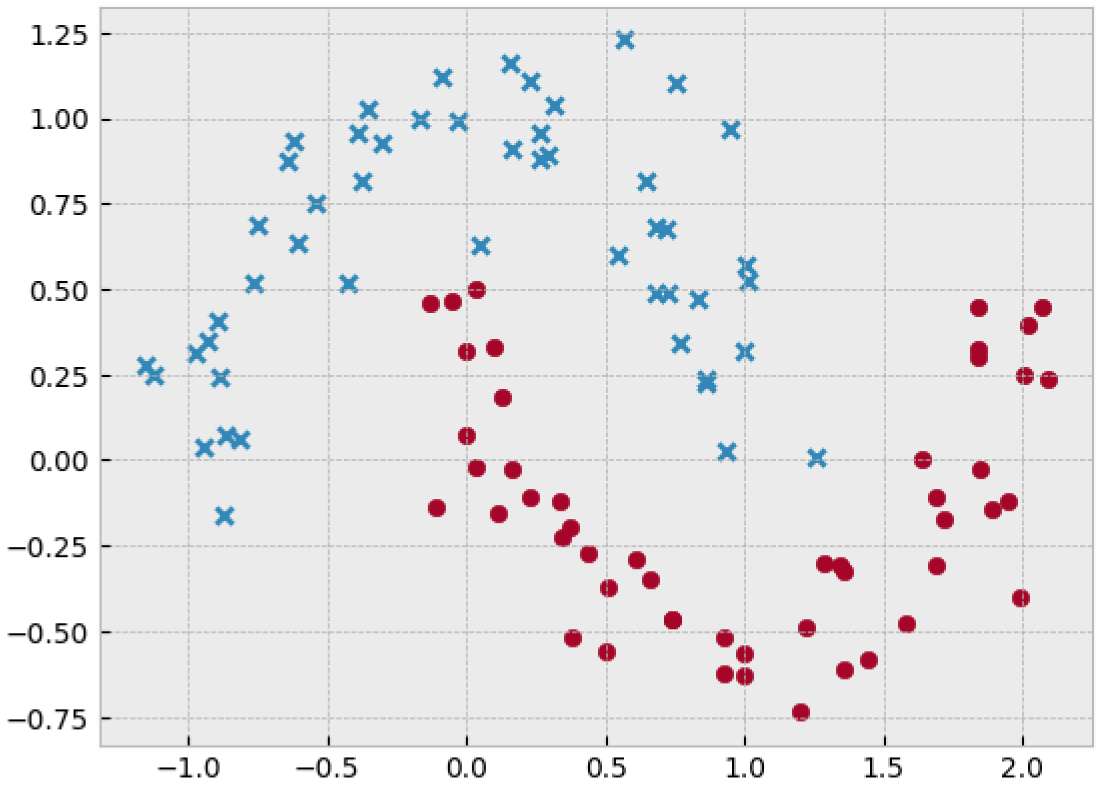
Instances are mapped into an enlarged space where each dimension is related to the distance from one instance. An instance that is closer to a dimension's instance in the feature space will have a larger value. The decision boundary will curve around clusters of instances of the same class.

3. **Sigmoid kernel**

$$K(\mathbf{x}, \mathbf{y}) = \tanh(\gamma\langle\mathbf{x}, \mathbf{y}\rangle + 1)$$

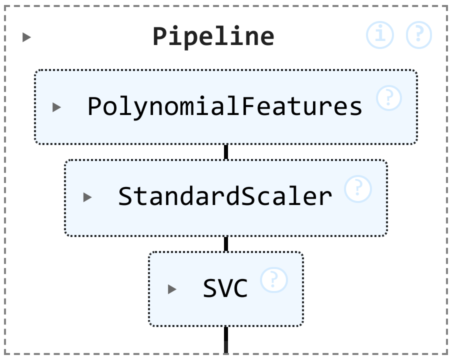
The sigmoid function is used due to a similar response to biological neurons. The sigmoid function restricts the gains from large inputs to between -1 and 1, reducing the spread of values in the enlarged space.

In [8]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PolynomialFeatures
from sklearn.datasets import make_moons

X, t = make_moons(n_samples=100, noise=0.15, random_state=0)
plt.scatter(X[t==0,0], X[t==0,1], marker='x')
plt.scatter(X[t==1,0], X[t==1,1]);
```

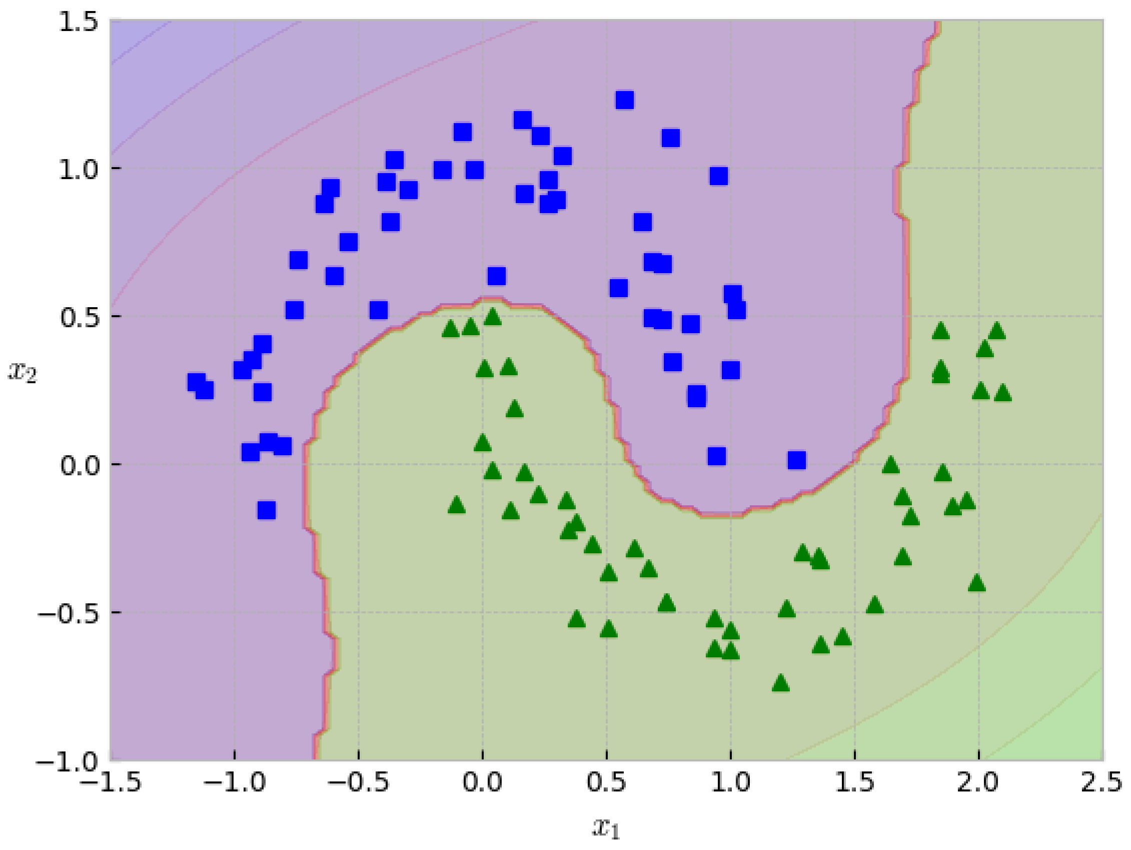


In [13]:
```

poly_svm = Pipeline([('poly_features', PolynomialFeatures(degree=3)),
                      ('scaler', StandardScaler()),
                     ('SVC', SVC(kernel='linear', C=10))])
poly_svm.fit(X, t)
```

Out[13]:



In [14]:
```
def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision, cmap=plt.cm.brg, alpha=0.1)

def plot_dataset(X, t, axes):
    plt.plot(X[:, 0][t==0], X[:, 1][t==0], "bs")
    plt.plot(X[:, 0][t==1], X[:, 1][t==1], "g^")
    plt.axis(axes)
    plt.grid(True)
    plt.xlabel("$x_1$")
    plt.ylabel("$x_2$", rotation=0)

plot_predictions(poly_svm, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, t, [-1.5, 2.5, -1, 1.5])
```



In [15]:
```
poly_svm.named_steps.SVC.coef_
```

Out[15]:
```
array([[ 0.        , -1.36393422, -0.25121301, -4.402802  , -0.44570237,
        -0.50069116,  6.67859633, -1.16995026,  2.19305263, -2.76784647]])
```

Adding polynomial features is simple to implement and can work great with all sorts of Machine Learning algorithms (not just SVMs). That said, at a low polynomial degree, this method cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow.

Fortunately, when using SVMs you can apply an almost miraculous mathematical technique called the **kernel trick** (explained in a moment). The kernel trick makes it possible to get the same result as if you had added many polynomial features, even with very high-degree polynomials, without actually having to add them. So there is no combinatorial explosion of the number of features because you don’t actually add any features. This trick is implemented by the SVC class. Let’s test it on the moons dataset:

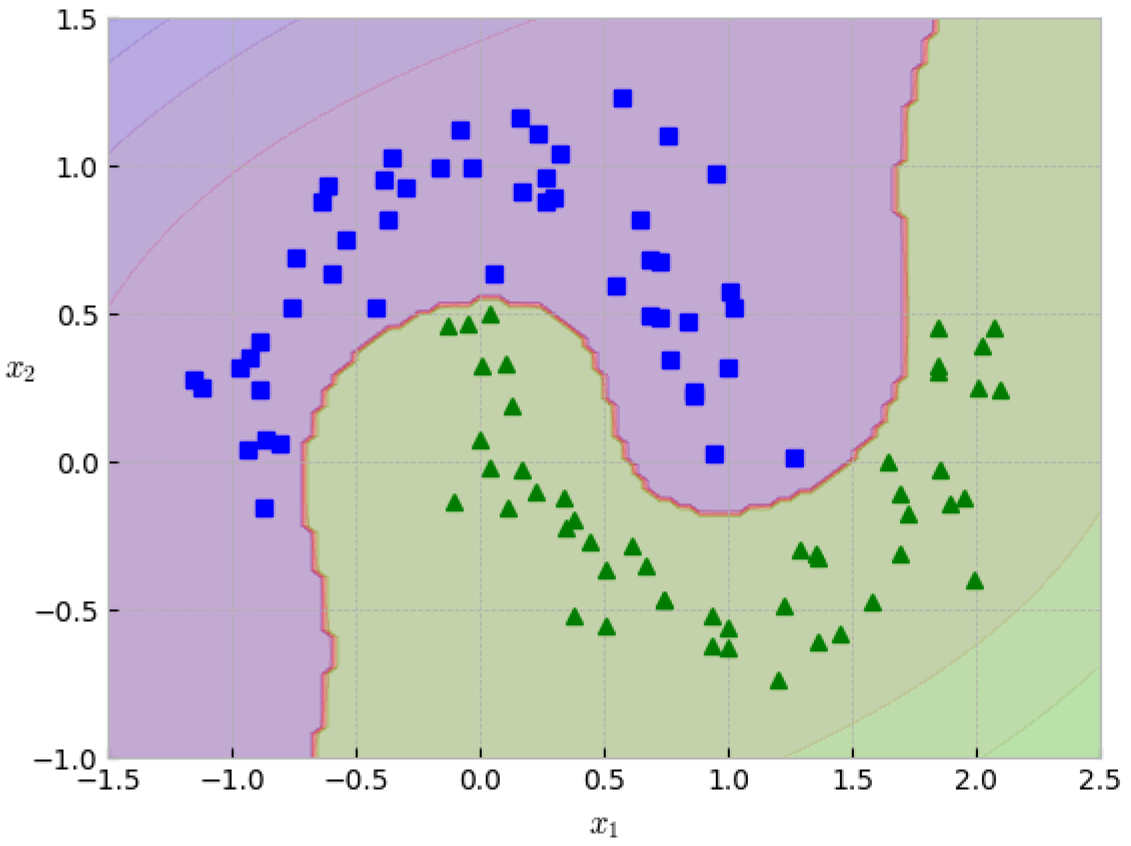
In [16]:
```
poly_svm_clf = Pipeline([('scaler', StandardScaler()),
                        ('SVC', SVC(kernel='poly', degree=3))])
poly_svm_clf.fit(X, t)
```

Out[16]:

```
▸ Pipeline
  ▸ StandardScaler
  ▸ SVC
```

In [17]:

```python
plot_predictions(poly_svm, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, t, [-1.5, 2.5, -1, 1.5])
```



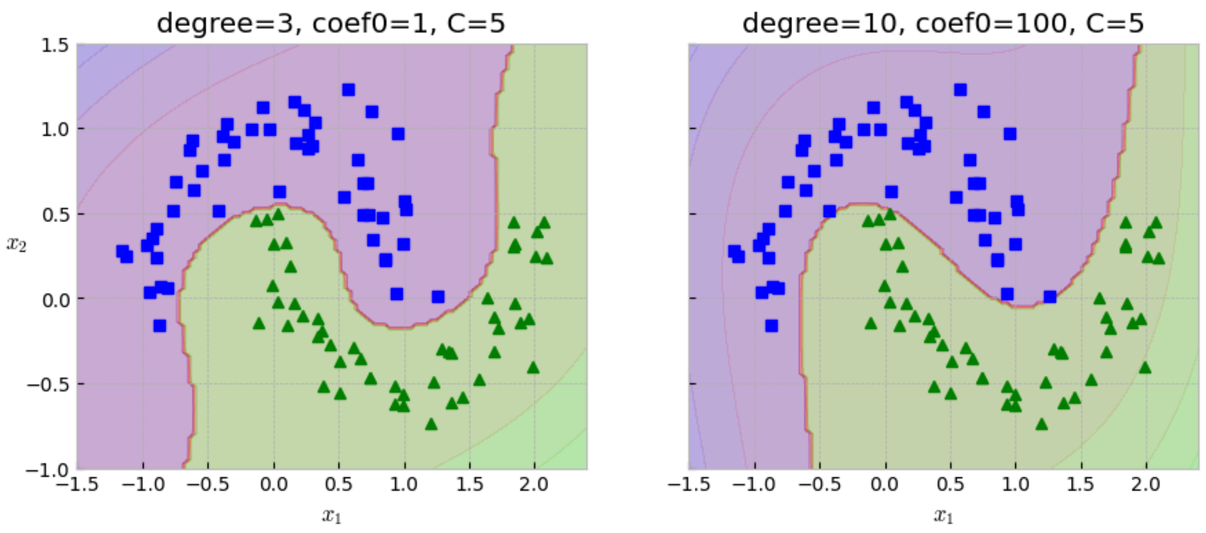
The hyperparameter `coef0` controls how much the model is influenced by high degree polynomials versus low-degree polynomials. This parameter is only significant if the kernel function is `poly` or `sigmoid`.

Note that in the polynomial kernel, the powers of values smaller than one gets closer and closer to 0, while the same power of value bigger than one grows to infinity. You can use `coef0` to *scale* your data so there is no such distinction - you can add $1 - \min \mathbf{x}^T\mathbf{y}$, so no values are smaller than 1.

In [18]:

```python
poly100_svm_clf = Pipeline([('scaler', StandardScaler()),
                            ('SVC', SVC(kernel='poly', degree=3, coef0=100))])
poly100_svm_clf.fit(X, t)

fig, axes = plt.subplots(ncols=2, figsize=(10.5, 4), sharey=True)
```

```
plt.sca(axes[0])
plot_predictions(poly_svm, [-1.5, 2.45, -1, 1.5])
plot_dataset(X, t, [-1.5, 2.4, -1, 1.5])
plt.title("degree=3, coef0=1, C=5")

plt.sca(axes[1])
plot_predictions(poly100_svm_clf, [-1.5, 2.45, -1, 1.5])
plot_dataset(X, t, [-1.5, 2.4, -1, 1.5])
plt.title("degree=10, coef0=100, C=5")
plt.ylabel("");
```



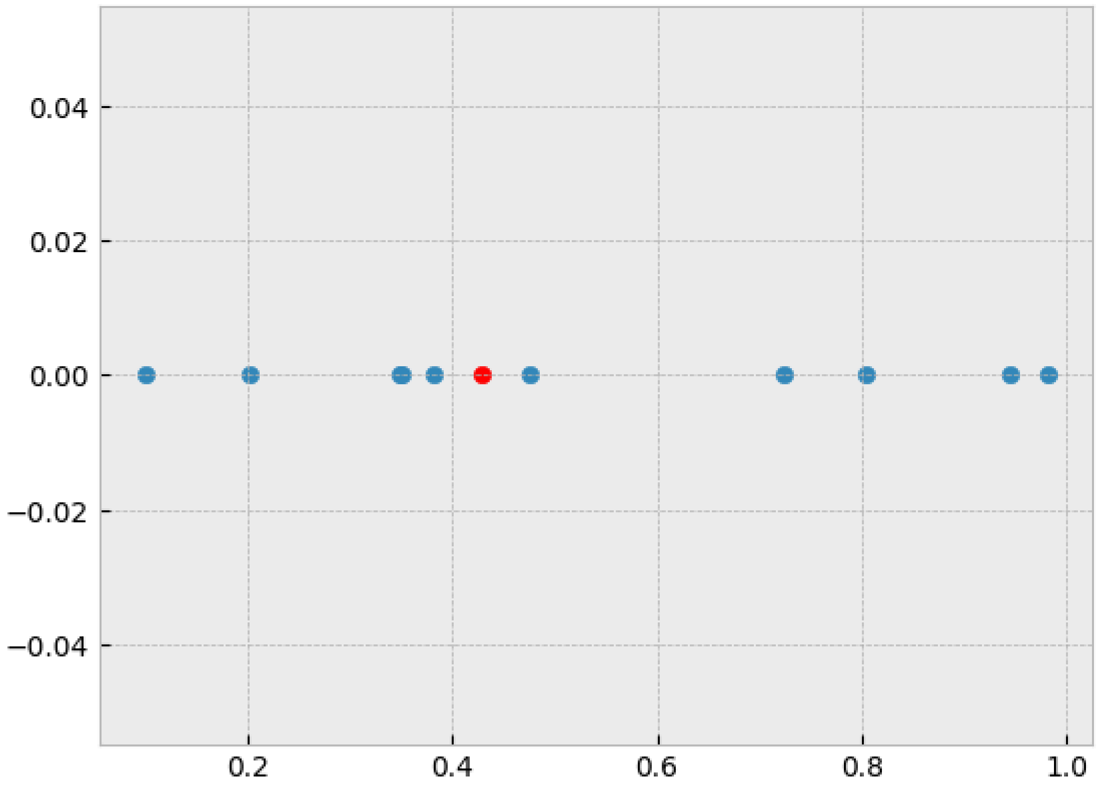
Gaussian RBF Kernel

A popular kernel function is the **radial basis function** (RBF kernel):

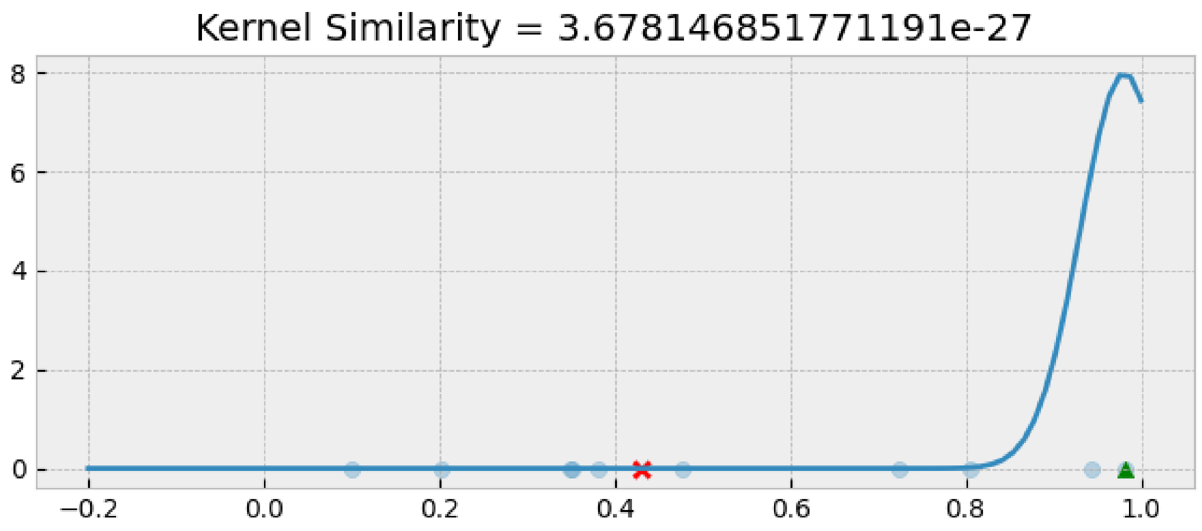
$$k_{\text{RBF}}(\mathbf{x},\mathbf{y}) = \exp\left(-\gamma\|\mathbf{x}-\mathbf{y}\|^2\right)$$

typical values for $\gamma = \frac{1}{2\sigma^2}$. As you can see, the RBF kernel is measures the **similarity** between two points as a function of the kernel **bandwidth** σ.

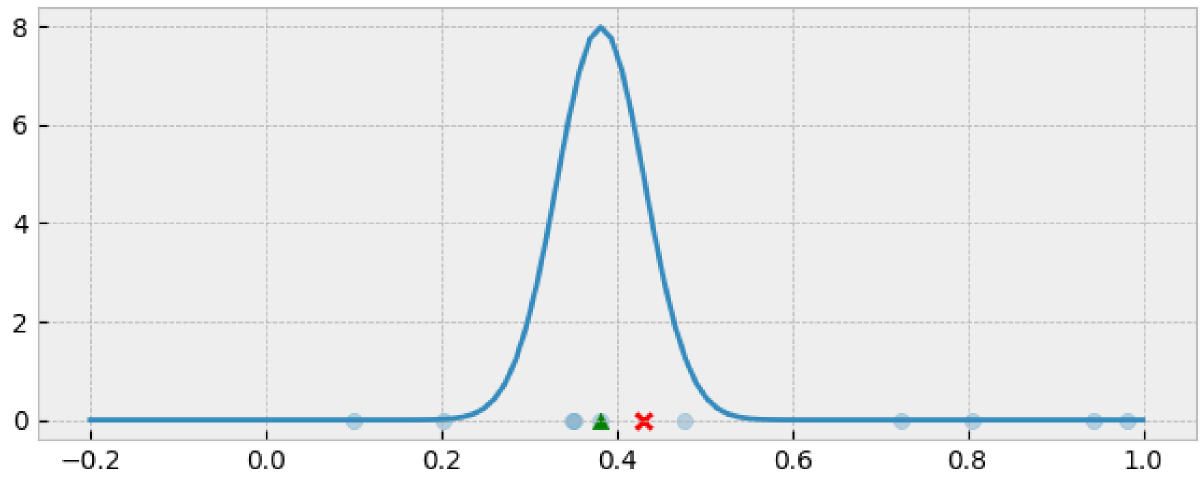
In [19]:
```
x = np.random.rand(10)
x1 = np.median(x)

plt.scatter(x, np.zeros(len(x)))
plt.scatter(x1, 0, c='r');
```



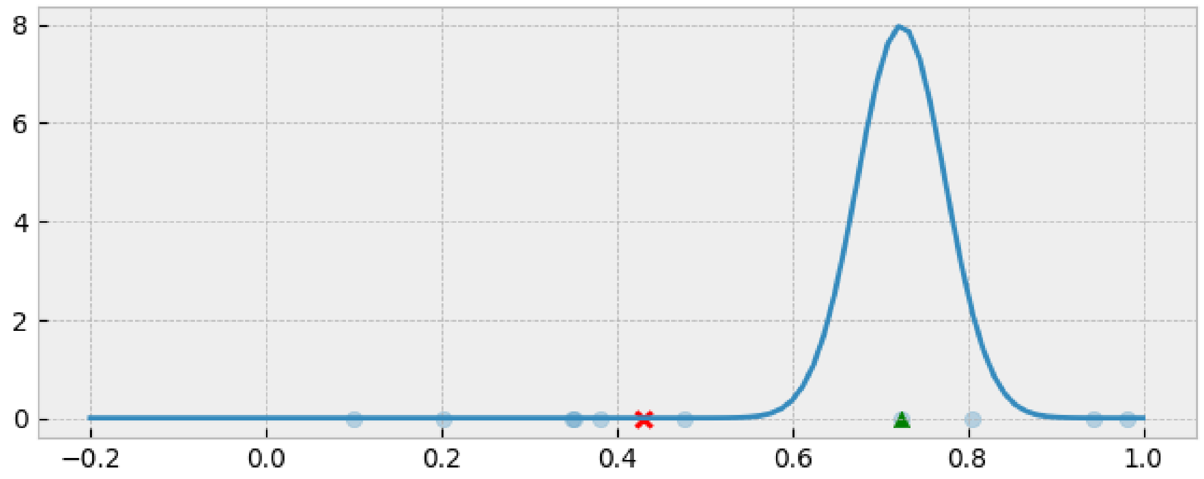
In [20]:
```
sig = 0.05
xline = np.linspace(-0.2,1,100)

for i in range(len(x)):
    plt.figure(figsize=(8,3))
    plt.scatter(x, np.zeros(len(x)), alpha=0.3)
    plt.scatter(x[i], 0, c='g', marker='^')
    plt.scatter(x1, 0, c='r',marker='x')
    plt.plot(xline, np.exp(-(xline-x[i])**2/(2*sig**2))/np.sqrt(2*np.pi*sig**2))
    k_similarity=np.exp(-np.linalg.norm(x[i]-x1)**2/(2*sig**2))
    plt.title('Kernel Similarity = '+str(k_similarity));
```



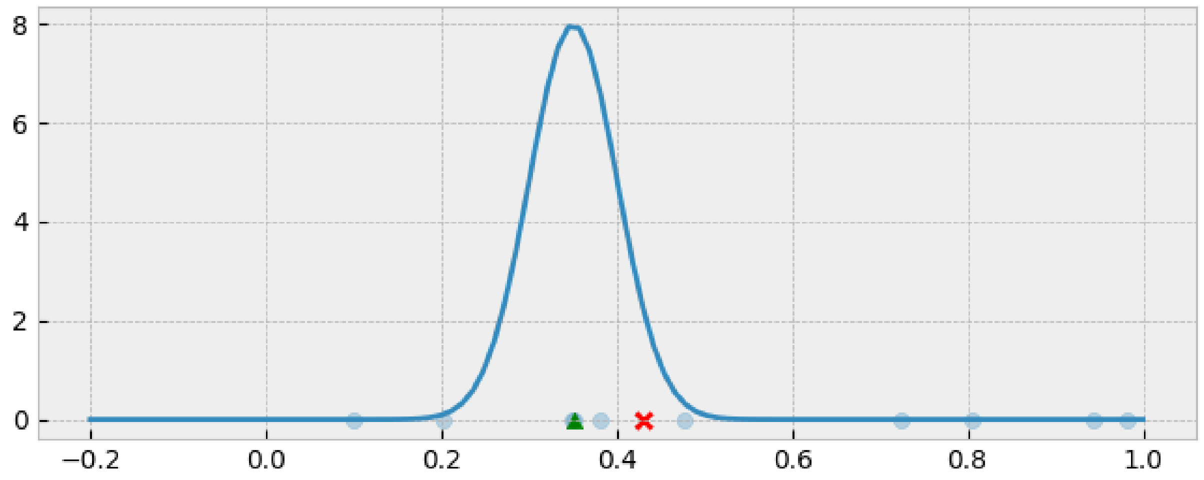
Kernel Similarity = 0.6387953669263055

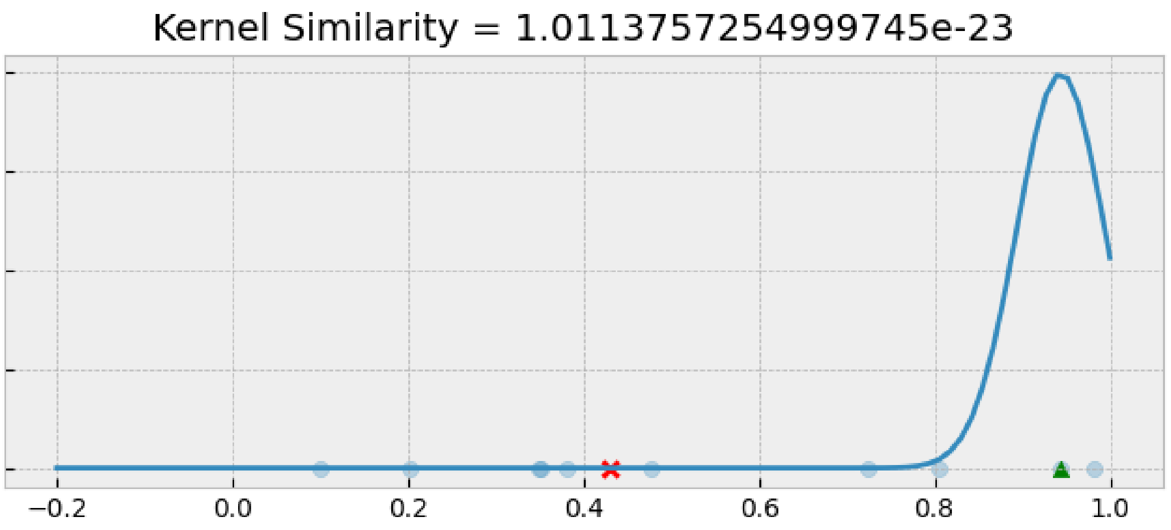


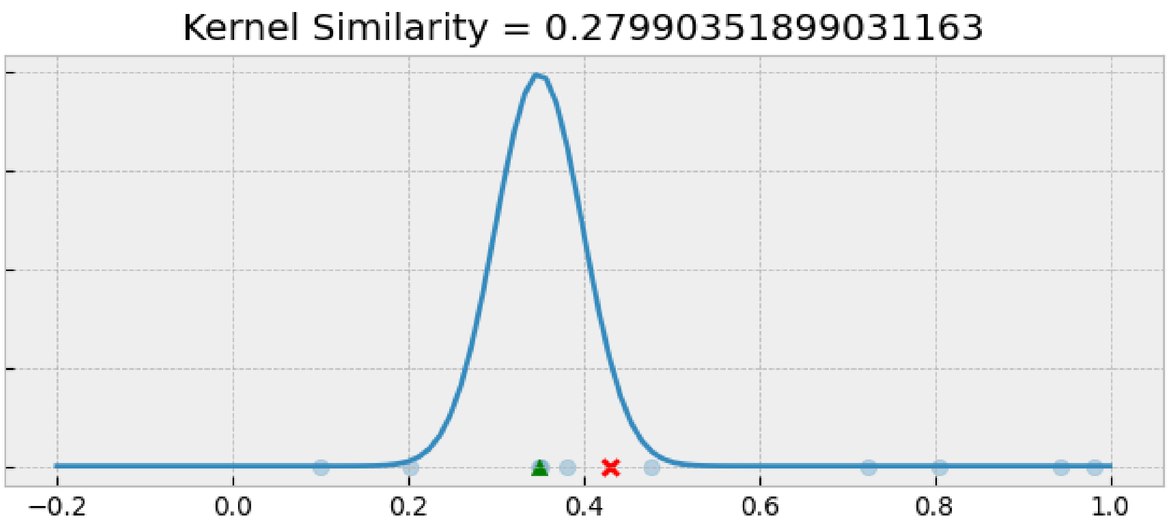
Kernel Similarity = 2.736262104805105e-08

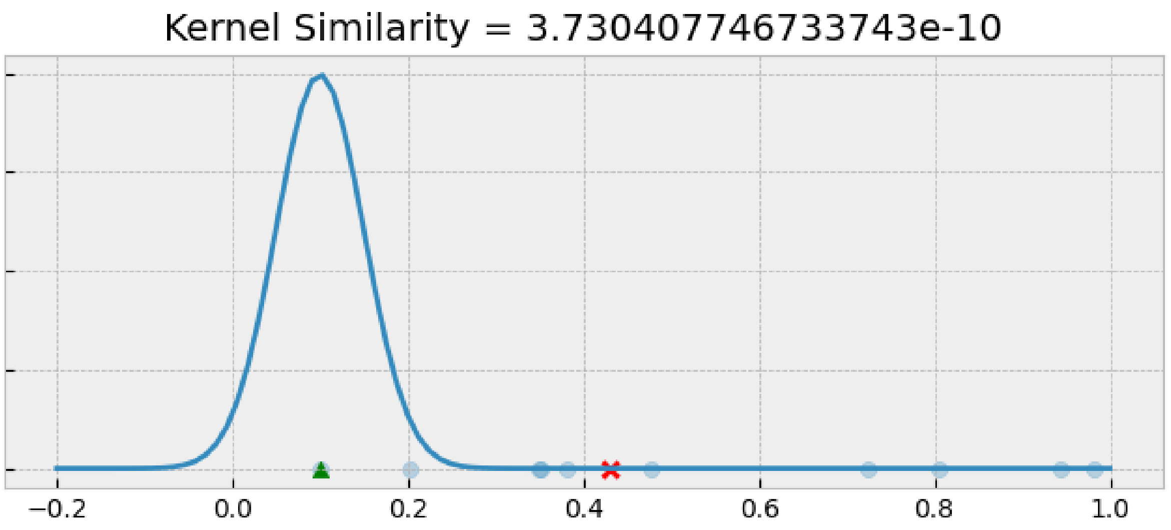


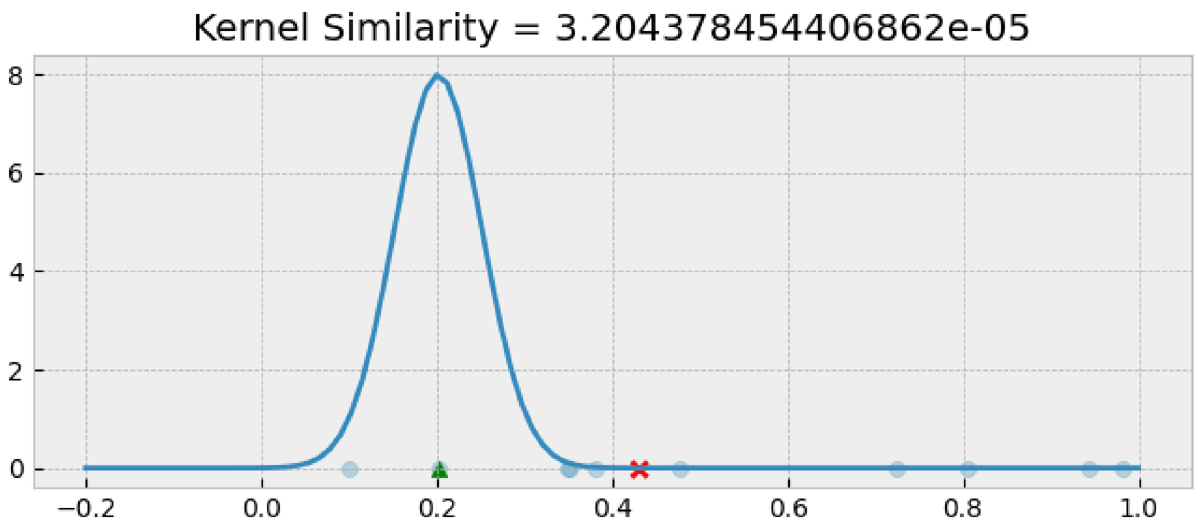
Kernel Similarity = 0.2896354020693192

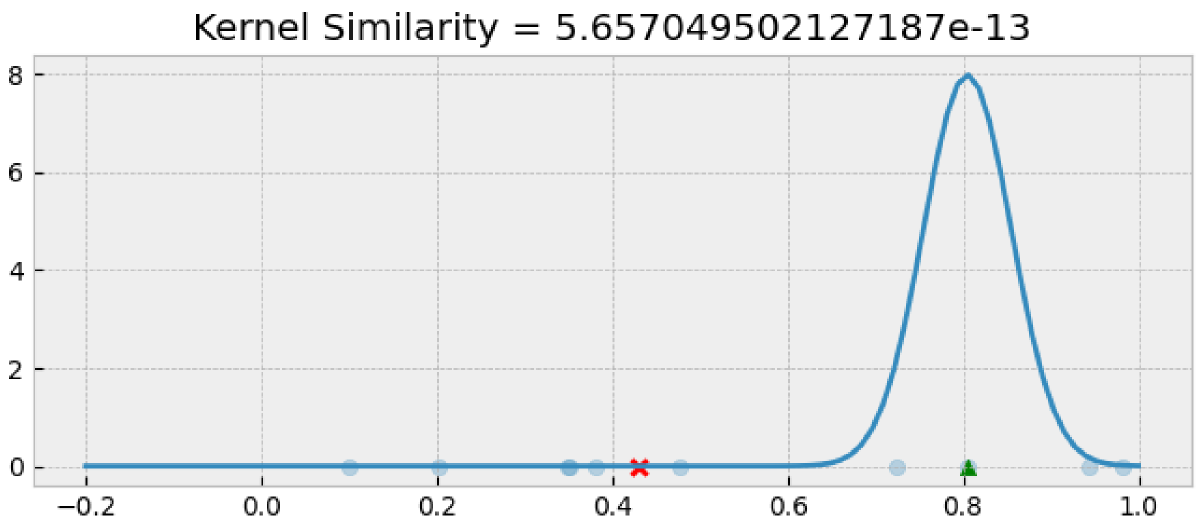


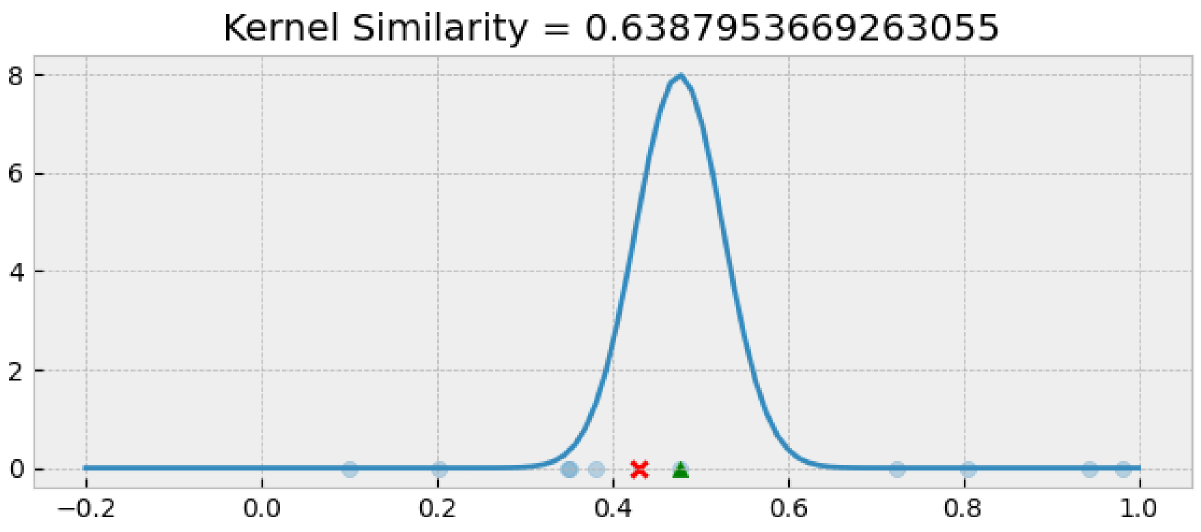

Kernel Similarity = 1.0113757254999745e-23
0
2
4
6
8
−0.2
0.0
0.2
0.4
0.6
0.8
1.0


Kernel Similarity = 0.27990351899031163
0
2
4
6
8
−0.2
0.0
0.2
0.4
0.6
0.8
1.0


Kernel Similarity = 3.730407746733743e-10
0
2
4
6
8
−0.2
0.0
0.2
0.4
0.6
0.8
1.0







Alright! I will explain this ENTIRE code in **BABY LANGUAGE**, like you're 100% new and learning from scratch. No math. No formulas. Just intuition.

🧸 What is happening overall?

You are testing **Polynomial SVMs** on the **moons dataset**.

The goal:

👉 **See how SVM changes its decision boundary when you change polynomial settings.**

That's it.

🧸 FIRST: the moons dataset

```
X, t = make_moons(n_samples=100, noise=0.15, random_state=0)
plt.scatter(...)
```

This creates a dataset that looks like two half-moons:

```
   🌙   🌙
     🌙   🌙
```

Two curved shapes → not linearly separable.

So we need special SVM kernels.

🧸 What is "polynomial SVM"?

Think of it like giving SVM the ability to draw:

- curved boundaries
- twisty boundaries
- more flexible shapes

depending on the polynomial degree.

🧸 PART 1 — PolynomialFeatures + Linear SVM

```
poly_svm = Pipeline([
    ('poly_features', PolynomialFeatures(degree=3)),
    ('scaler', StandardScaler()),
    ('SVC', SVC(kernel='linear', C=10))
])
poly_svm.fit(X, t)
```

BABY LANGUAGE:

- First step: **create new features that allow curves** (Like adding magic powers to your data.)
- Second step: **scale the data** (Make all features the same size.)
- Third step: **train a linear SVM** BUT because of polynomial features, the SVM can now draw **curvy boundaries**.

Even though the kernel is `"linear"`, the polynomial features make the data LOOK linear in a higher dimension.

🧸 PART 2 — plot_predictions()

This function:

```
def plot_predictions(clf, axes):
```

Does this:

👉 **Color the entire background based on the SVM's decision.**

- Blue area = predicted class 0
- Green area = predicted class 1
- Darker regions = stronger confidence
- Lighter = unsure

It creates a smooth "map" of the classifier.

🧸 PART 3 — plot_dataset()

This only draws your data:

- class 0 → blue squares
- class 1 → green triangles

This overlays on the prediction background.

🧸 PART 4 — Show the polynomial decision boundary

```
plot_predictions(poly_svm, ...)
plot_dataset(X, t, ...)
```
You see how the polynomial SVM shapes the boundary.

It will be **curvy**, not straight.

🧸 PART 5 — poly_svm.named_steps.SVC.coef_

This tries to show the SVM weights. Not important for your understanding. Just ignore it.

🧸 PART 6 — LET'S USE REAL polynomial kernel

Before: You used PolynomialFeatures + linear SVM.

Now:

```
poly_svm_clf = Pipeline([
    ('scaler', StandardScaler()),
    ('SVC', SVC(kernel='poly', degree=3))
])
```
This uses the **real polynomial kernel**, which is more direct.

Polynomial kernel creates curved boundaries without manually adding features.

🧸 PART 7 — Try a more extreme polynomial

```
poly100_svm_clf = Pipeline([
    ('scaler', StandardScaler()),
    ('SVC', SVC(kernel='poly', degree=3, coef0=100))
])
```
Here:

- degree = 3 → still cubic

- **coef0 = 100** → makes curves more dramatic (Think: more bendy, more wiggles, more powerful)

This makes the polynomial curve **much more complex**.

🧸 PART 8 — Compare two polynomial SVMs side by side

`fig, axes = plt.subplots(ncols=2, ...)`
This makes a 2-panel plot:

LEFT SIDE

"degree=3, coef0=1, C=5"

👉 moderate curves 👉 smooth 👉 not too wiggly

RIGHT SIDE

"degree=10, coef0=100, C=5"

👉 extreme curves 👉 VERY wiggly 👉 may overfit 👉 boundary twists sharply around points

---

🧸 BABY SUMMARY OF THE WHOLE THING

🌙 You have two moon-shaped clusters.

🎨 You test different polynomial SVMs:

- normal polynomial → simple curves
- extreme polynomial → very twisty curves

🎯 The goal is to SEE:

👉 how polynomial degree and coef0 change the decision boundary.

Higher settings = more flexible = more wiggly = more overfitting.

🧸 SUPER SIMPLE FINAL SUMMARY

- **Polynomial SVM** lets SVM draw **curvy lines**.
- **Higher degree or higher coef0 → more curves and wiggles**.
- You plotted decision boundaries to compare how each SVM behaves.

If you want, I can:

✅ draw a cartoon to explain coef0 and degree ✅ explain what degree actually means ✅ show what coef0 does using pictures Just ask!

Okay! Let me explain **degree** and **coef0** in the polynomial kernel in **SUPER SIMPLE BABY LANGUAGE** with examples.

🎨 Polynomial Kernel Formula (don't worry, we'll make it simple)

[K(x, x') = (x \cdot x' + \text{coef0})^\text{degree}]

This formula has two knobs:

- **degree**
- **coef0**

Let's understand them like everyday objects.

🧸 1. What is DEGREE? (very simple)

👉 **DEGREE tells how BENDY the boundary can be.**

Imagine you draw a line separating two groups:

- **degree = 1** → straight line
- **degree = 2** → simple curved line (U-shape)
- **degree = 3** → more wiggly
- **degree = 5** → even more wiggly
- **degree = 10** → crazy twisty boundary

🔥 Higher degree = more wiggles.

🎯 Baby analogy:

Think of degree like how much “curving power” the SVM has.

- degree 1 → no curve
- degree 2 → small curve
- degree 3 → medium curves
- degree 10 → super bendy noodle

🧸 2. What is COEF0? (even simpler)

👉 **coef0 decides how much influence the curves have vs straight parts.**

Think of it like a **curve booster**.

- **small coef0 (like 1)** → soft curves
- **large coef0 (like 100)** → BIG dramatic curves

🎯 Baby analogy:

coef0 = spice level 🌶️

- coef0 = 1 → mild
- coef0 = 10 → spicy
- coef0 = 100 → super spicy (sharp bends)

🎨 Put together: degree + coef0

- **degree = how many curves**
- **coef0 = how strong the curves are**

Example:

degree	coef0	How the boundary looks
1	any	straight line
3	1	gentle curves
3	100	dramatic curves
10	1	many wiggles
10	100	extremely wiggly (overfits)

🌈 Mini cartoon explanation

With LOW degree, LOW coef0:

```
soft curve
   _
 /   \
```

With LOW degree, HIGH coef0:

```
stronger curve
   __
 /     \
```

With HIGH degree, HIGH coef0:

```
crazy twists
__/\/\/\/\/\__
```

🧠 Very simple memory trick

👉 **degree = how many bends** 👉 **coef0 = how strong the bends**

If you want, I can show:

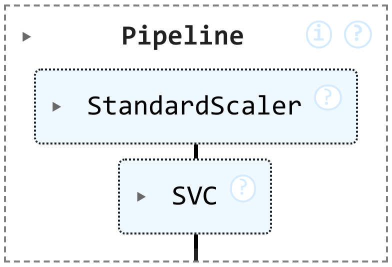
📌 graphs comparing different degrees 📌 graphs comparing different coef0 values 📌 a simple animation idea 📌 a one-sentence definition for your exam notes

Just tell me!

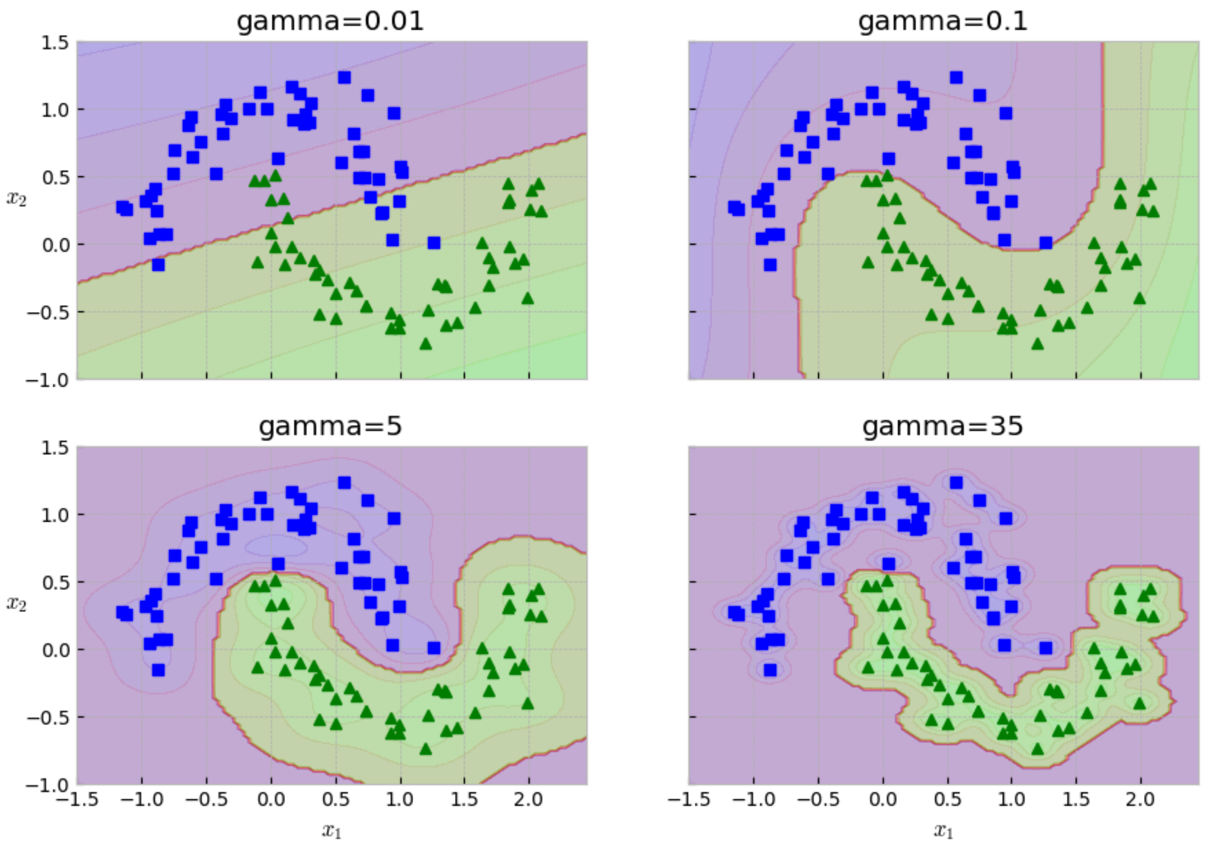
Let's try the `SVC` class with the Gaussian RBF kernel:

```
In [21]: rbf_svm_clf = Pipeline([('scaler', StandardScaler()),
                                 ('svm', SVC(kernel='rbf', gamma=5,
                                             C=0.001))])
         rbf_svm_clf.fit(X,t)
```

Out[21]:



In [22]:
```python
gamma_vals = [0.01, 0.1, 5, 35]

svm_clfs = []
for gamma in gamma_vals:
    rbf_kernel_svm_clf = Pipeline([('scaler', StandardScaler()),
                                   ('svm', SVC(kernel='rbf',
                                               gamma=gamma,
                                               C=100))])

    rbf_kernel_svm_clf.fit(X, t)
    svm_clfs.append(rbf_kernel_svm_clf)

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10.5, 7), sharex=True, sharey=T

for i, svm_clf in enumerate(svm_clfs):
    plt.sca(axes[i // 2, i % 2])
    plot_predictions(svm_clf, [-1.5, 2.45, -1, 1.5])
    plot_dataset(X, t, [-1.5, 2.45, -1, 1.5])
    gamma = gamma_vals[i]
    plt.title(f"gamma={gamma}")
    if i in (0, 1):
        plt.xlabel("")
    if i in (1, 3):
        plt.ylabel("");
```



Perfect — I will explain this **SUPER SIMPLE**, like you're completely new. No math. No jargon. Just intuition. This code is showing **how the SVM RBF kernel behaves when you change gamma**.

🌟 FIRST: What is gamma? (VERY SIMPLE)

👉 **Gamma = how far each point "influences" the boundary.**

Think of every training point dropping a small “paint splatter” on the graph:

- **Small gamma** → paint spreads wide → smooth, simple boundary
- **Big gamma** → paint is tight → boundary wiggles and overfits

That’s it.

⭐ WHAT YOUR CODE DOES (in baby language)

🔹 Step 1 — Train SVM with RBF kernel

```
rbf_svm_clf = Pipeline([
    ('scaler', StandardScaler()),
    ('svm', SVC(kernel='rbf', gamma=5, C=0.001))
])
rbf_svm_clf.fit(X, t)
```

This means:

- scale the data
- train an SVM
- using **RBF kernel**
- with **gamma = 5**
- C = 0.001 (very soft margin)

This is just one example model.

🔹 Step 2 — Try multiple gamma values

```
gamma_vals = [0.01, 0.1, 5, 35]
```

These are **four different strengths** of "paint splatter."

We will train four SVM models — one for each gamma.

🔹 Step 3 — Create SVMs for every gamma value

```
for gamma in gamma_vals:
    rbf_kernel_svm_clf = Pipeline([
        ('scaler', StandardScaler()),
        ('svm', SVC(kernel='rbf', gamma=gamma, C=100))
    ])
    rbf_kernel_svm_clf.fit(X, t)
    svm_clfs.append(rbf_kernel_svm_clf)
```
Translated to simple English:

👉 For each gamma value:

- build an SVM
- train it
- store it in a list

So at the end you have **four different SVMs**:

Model	Gamma
Model 1	0.01
Model 2	0.1
Model 3	5
Model 4	35

Each gamma creates a **different shape of decision boundary**.

🔹 Step 4 — Make a 2×2 grid of plots

```
fig, axes = plt.subplots(2, 2, figsize=(10.5, 7))
```
This just prepares four boxes for four pictures.

🔹 Step 5 — Plot each SVM's predictions

```
plt.sca(axes[i // 2, i % 2])
plot_predictions(svm_clf, [-1.5, 2.45, -1, 1.5])
plot_dataset(X, t, [-1.5, 2.45, -1, 1.5])
plt.title(f"gamma={gamma}")
```
This part fills each box with:

- the decision boundary
- the training points
- a title that says what gamma was used

So you will get something like:

```
+----------------------+----------------------+
|  gamma = 0.01        |   gamma = 0.1        |
| (smooth boundary)    | (more curved)        |
+----------------------+----------------------+
|  gamma = 5           |   gamma = 35         |
| (very curvy)         | (super overfitted)   |
+----------------------+----------------------+
```

⭐ ⭐ VERY SIMPLE SUMMARY ⭐ ⭐

Your entire code is doing this:

👉 Train SVM four times

👉 Each time with a different gamma value

👉 Draw all four results

👉 So you can SEE how gamma changes the model

That's it.

⭐ What you will observe

🔹 gamma = 0.01

Boundary is almost a **straight line** (too simple).

🔹 gamma = 0.1

Boundary becomes **a little curved** (good flexibility).

🔹 gamma = 5

Boundary becomes **very wiggly**.

🔹 gamma = 35

Boundary becomes **crazy wiggly** → **overfits**.

🌈 If gamma is small → smooth

🔥 If gamma is large → crazy and overfit

That’s the whole lesson of this code.

If you want, I can draw a picture showing small vs big gamma in cartoon form.

Computational Complexity

The `SVC` class is based on the `libsvm` library, which implements an algorithm that supports the kernel trick. The training time complexity is usually between $O(m^2 \times N)$ and $O(m^3 \times N)$, where M is the number of features and N is the number of samples. Unfortunately, this means that it gets dreadfully **slow when the number of training instances gets large** (e.g., hundreds of thousands of instances).

This algorithm is perfect for complex small or medium-sized training sets. It scales well with the number of features, especially with sparse features (i.e., when each instance has few nonzero features). In this case, the algorithm scales roughly with the average number of nonzero features per instance.
