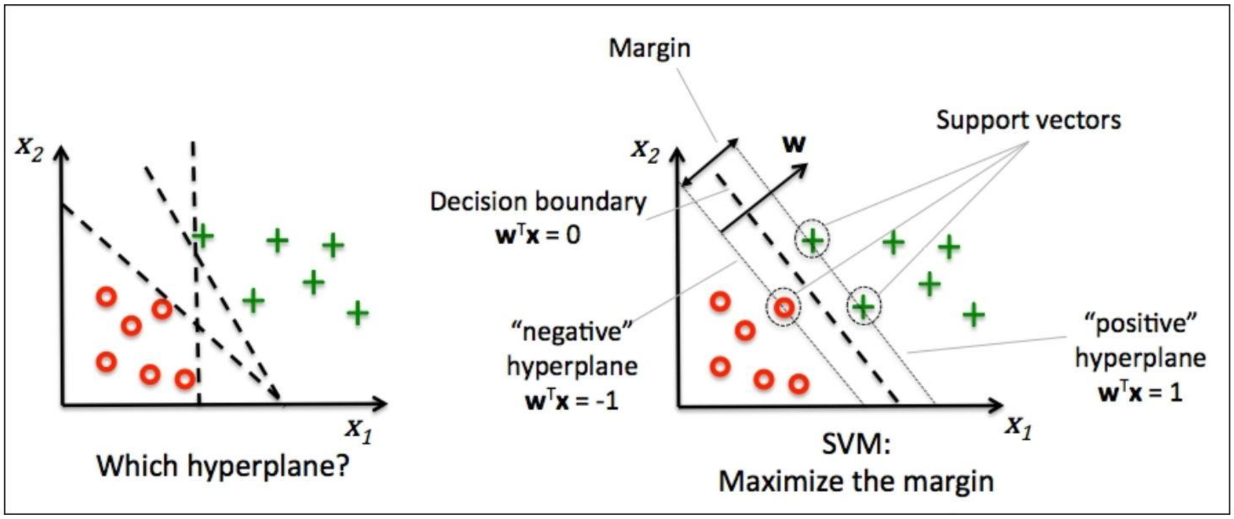
Understanding Support Vector Machines (SVM)

Based on: SVM chapter of the Python Data Science Handbook



Support vector machines (SVMs) are a particularly powerful and flexible class of supervised algorithms for both classification and regression. In this section, we will develop the intuition behind support vector machines and their use in classification problems.

We begin with the standard imports:

```
import numpy as np

import matplotlib.pyplot as plt

from scipy import stats

# use seaborn plotting defaults

import seaborn as sns; sns.set()
```

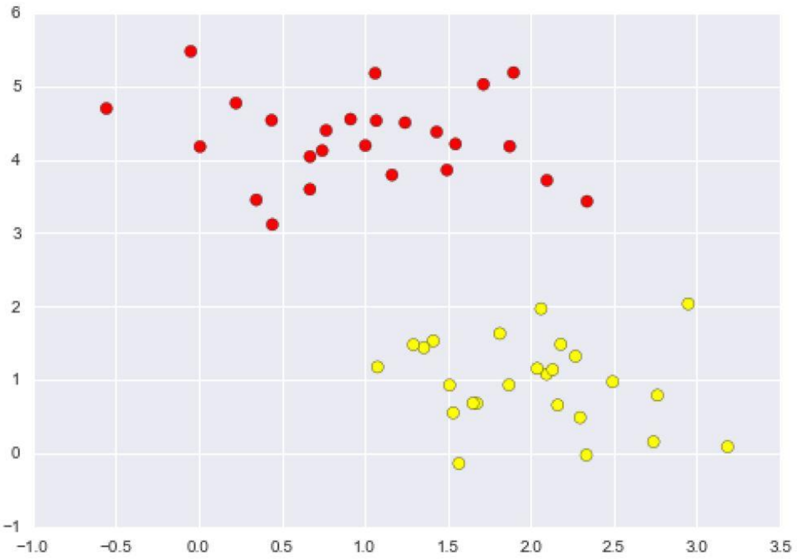
Motivating Support Vector Machines

A classification problem can be solved by using a generative approach or a discriminative approach. An example of generative classification is Bayesian classification where the objective is to find a model that describes the distribution of each underlying class. Then we can use the generative model to probabilistically determine the label for new points.

In this tutorial we consider discriminative classification: rather than modeling each class, we find a line or curve (in two dimensions) or manifold (in multiple dimensions) that divides the classes from each other.

Consider the simple case of a classification task in which the two classes of points are well separated:

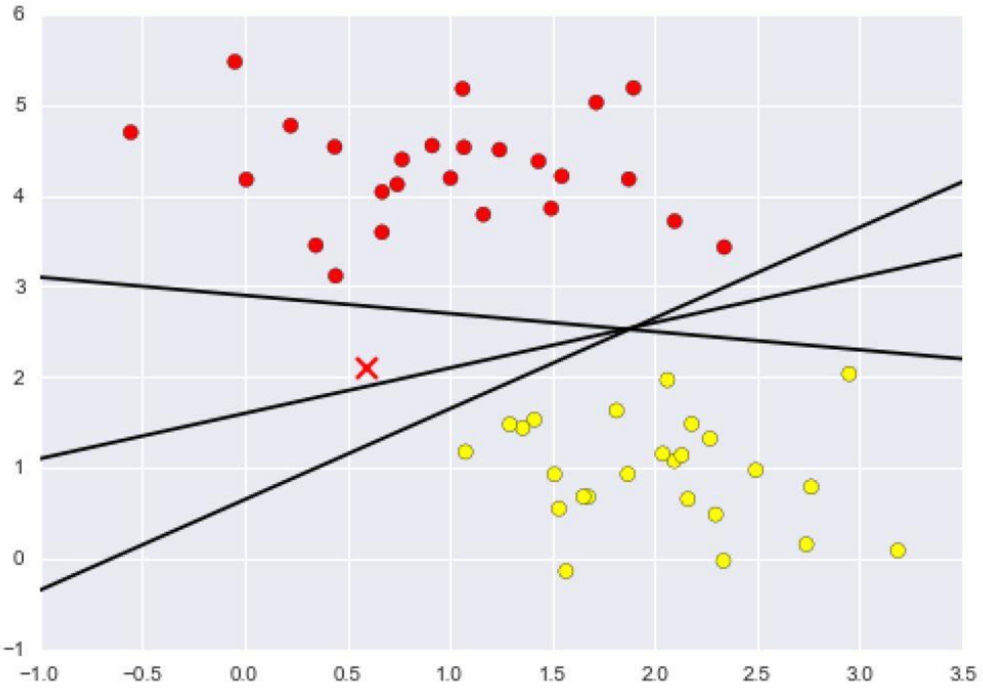
```
from sklearn.datasets.samples_generator import make_blobs

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='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:

```
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)

plt.show()
```

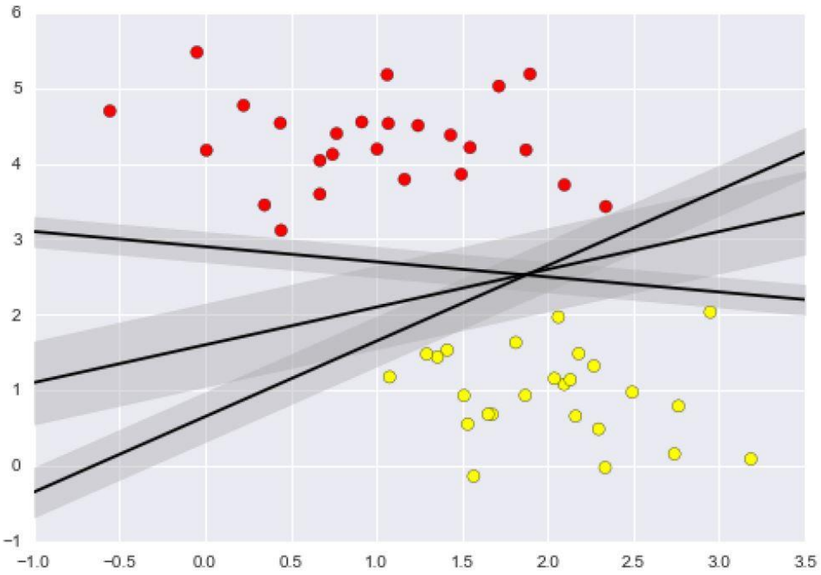


These are three very different separators which perfectly discriminate between these samples. Depending on which you choose, a new data point (e.g., the one marked by the "X" in this plot) will be assigned a different label! Evidently our simple intuition of "drawing a line between classes" is not enough.

Maximizing the Margin

Support vector machines offer one way to improve on 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.

```
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)

plt.show()
```



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.

Fitting a support vector machine

Let's 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. For the time being, we will use a linear kernel and set the C parameter to a very large number (we'll discuss the meaning of these in more depth momentarily).

```
from sklearn.svm import SVC

model = SVC(kernel='linear', C=1E10)

model.fit(X, y)
```

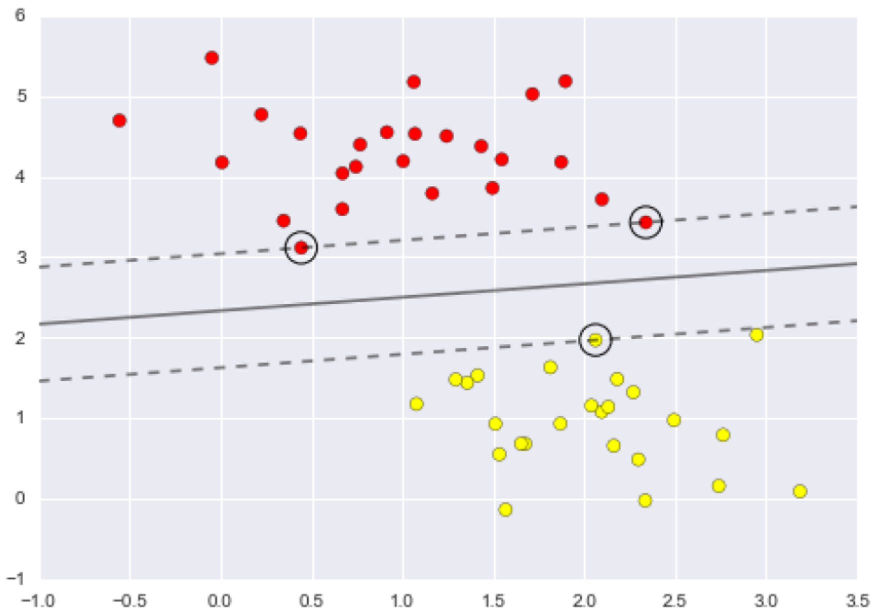
```
SVC(C=10000000000.0, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape=None,

  degree=3, gamma='auto', kernel='linear', max_iter=-1, probability=False, random_state=None,

  shrinking=True, tol=0.001, verbose=False)
```

To better visualize what's happening here, let's create a quick convenience function that will plot SVM decision boundaries for us:

```
def plot_svc_decision_function(model, ax=None, plot_support=True):

   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)
```

To call the function do:

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

plot_svc_decision_function(model)

plt.show()
```



This is the dividing line that maximizes the margin between the two sets of points. Notice that a few of the training points just touch the margin: they are indicated by the black 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:

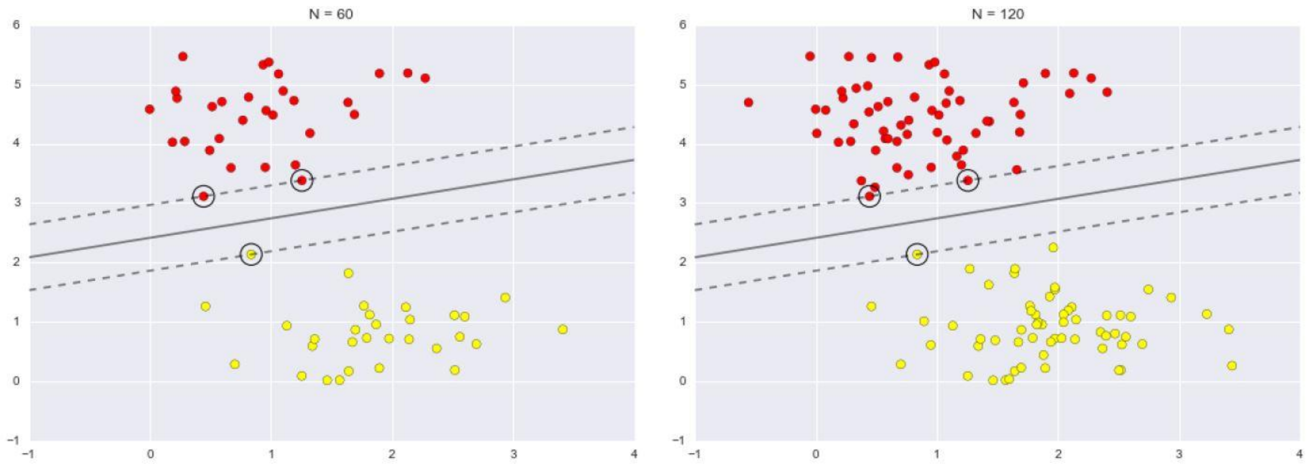
```
model.support_vectors_
```

```
array([[ 0.44359863, 3.11530945],

       [ 2.33812285, 3.43116792],

       [ 2.06156753, 1.96918596]])
```

A key to this classifier's success is that for the fit, 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:

```
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))
plt.show()
```



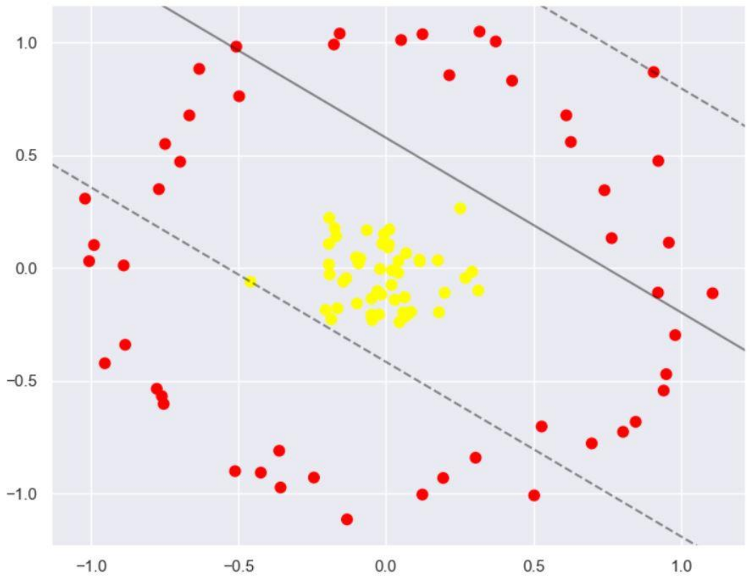
In the left panel, we see the model and the support vectors for 60 training points. In the right panel, we have doubled the number of training points, but the model has not changed: the three support ectors from the left panel are still the support vectors from the right panel. This insensitivity to the exact behavior of distant points is one of the strengths of the SVM model.

Beyond linear boundaries: Kernel SVM

SVM becomes extremely powerful when it is combined with kernels. We saw a version of kernels in the basis function regressions in the Linear Regression tutorial. There we projected our data into higher-dimensional space defined by polynomials and Gaussian basis functions, and thereby were able to fit for nonlinear relationships with a linear classifier.

In SVM models we can use a version of the same idea. To motivate the need for kernels, let's look at some data that is not linearly separable:

```
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)

plt.show()
```

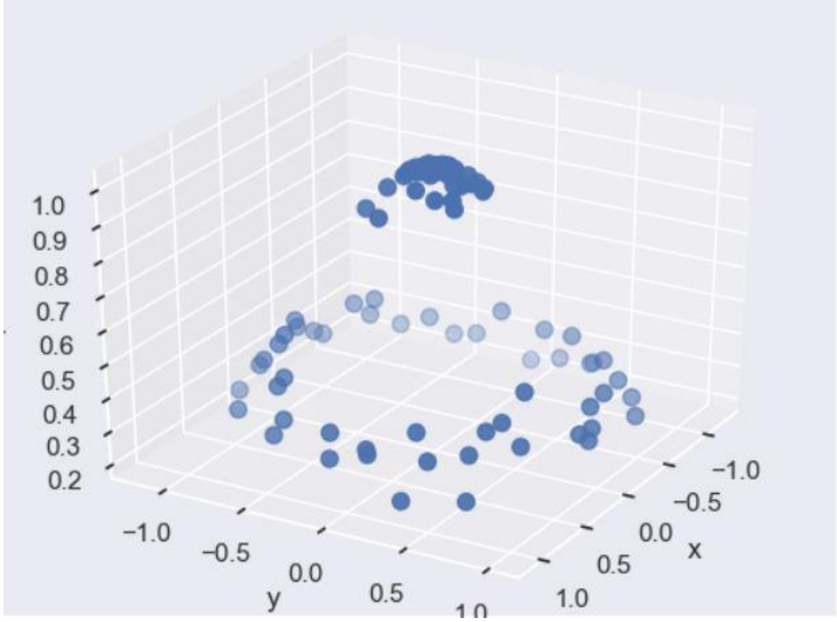


It is clear that no linear discrimination will ever be able to separate this data. But we can think about how we might project the data into a higher dimension such that a linear separator would be sufficient. For example, one simple projection we could use would be to compute a radial basis function centered on the middle clump:

```
r = np.exp(-(X ** 2).sum(1))
```

We can visualize this extra data dimension using a three-dimensional plot:

```
from mpl_toolkits import mplot3d

ax = plt.subplot(projection='3d')

ax.scatter3D(X[:, 0], X[:, 1], r, s=50, cmap='autumn')

ax.view_init(elev=30, azim=30)

ax.set_xlabel('x')

ax.set_ylabel('y')

ax.set_zlabel('r')

plt.show()
```



We can see that with this additional dimension the data becomes trivially linearly separable by drawing a separating plane at r = 0.7

Here we had to choose and carefully tune our projection: if we had not centered our radial basis function in the right location, we would not have seen such clean, linearly separable results. In general, the need to make such a choice is a problem: we would like to somehow automatically find the best basis functions to use.

One strategy to this end is to compute a basis function centered at every point in the dataset, and let the SVM algorithm go through the results. This type of basis function transformation is known as a kernel transformation as it is based on a similarity relationship (or kernel) between each pair of points.

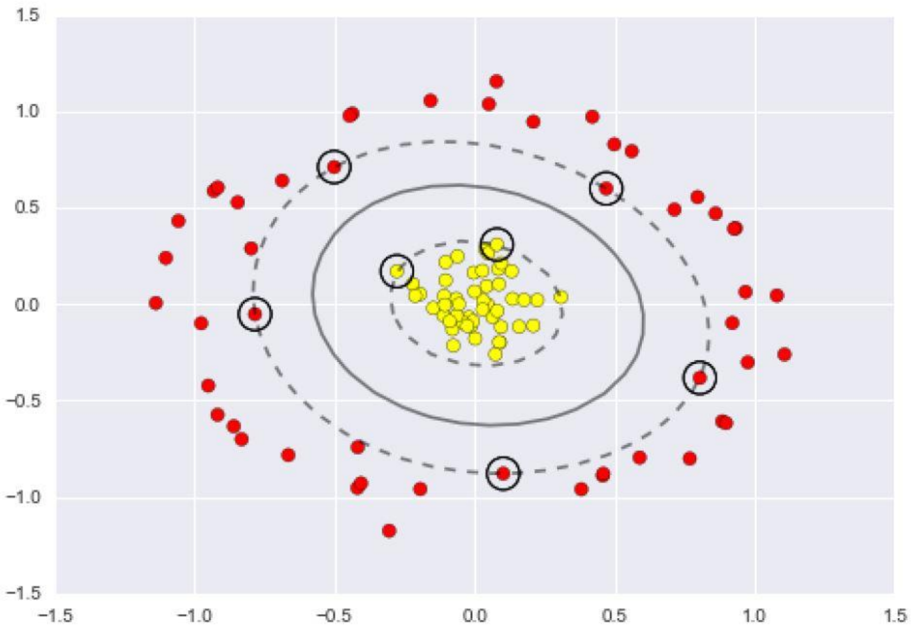
A potential problem with this strategy—projecting points into dimensions—is that it might become very computationally intensive as grows large. However, because of a neat little procedure known as the kernel trick (https://en.wikipedia.org/wiki/Kernel_trick), a fit on kerneltransformed data can be done implicitly—that is, without ever building the full –dimensional representation of the kernel projection! 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 a linear kernel to an RBF (radial basis function) kernel using the kernel model hyperparameter:

```
clf = SVC(kernel='rbf', C=1E6)

clf.fit(X, y)
```

```
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)
```

```
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')

plt.show()
```

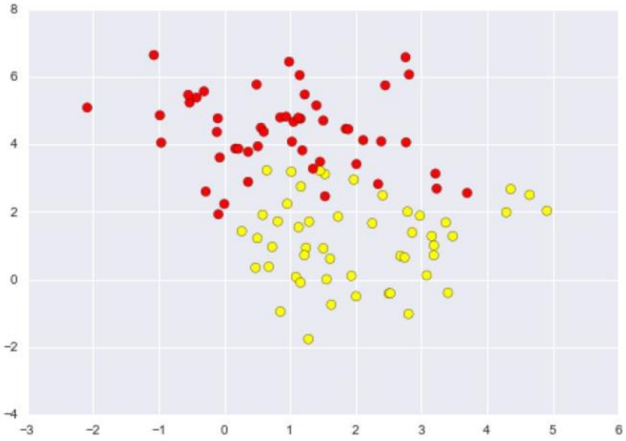


Using this kernelized support vector machine, we learn a suitable nonlinear decision boundary. This kernel transformation strategy is used often in machine learning to turn fast linear methods into fast nonlinear methods especially for models in which the kernel trick can be used.

Tuning the SVM: Softening Margins

Our discussion thus far has centered around very clean datasets in which a perfect decision boundary exists. But what if your data has some amount of overlap? For example, you may have data like this:

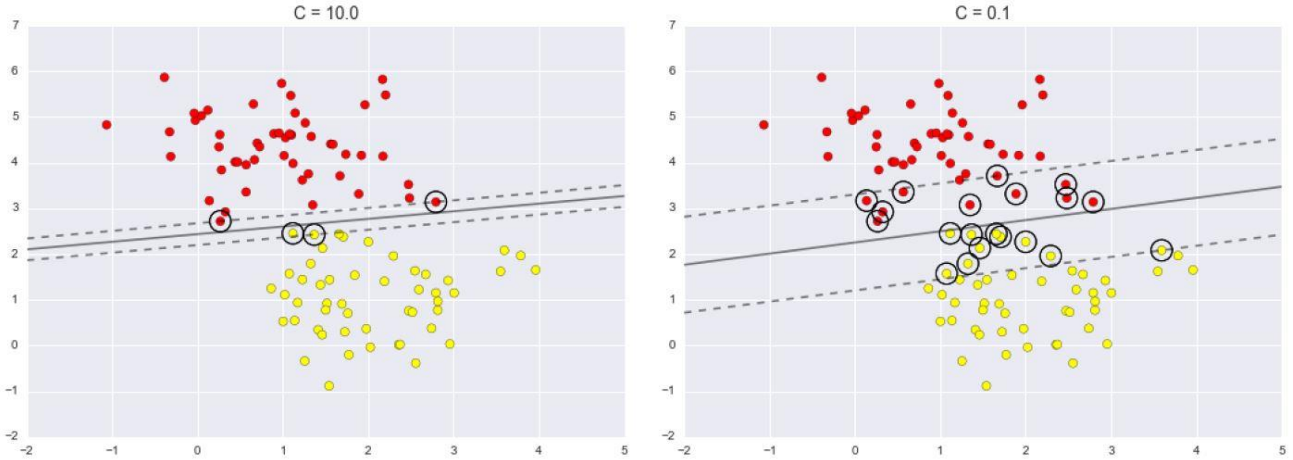
```
X, y = make_blobs(n_samples=100, centers=2, random_state=0, cluster_std=1.2)

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



To handle this case we need to soften the margin in the SVM implementation. That is, to allow some of the points to creep into the margin if that allows for 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 the points cannot lie in it. For smaller C, the margin is softer, and can grow to encompass some points.

The plot shown below gives a visual picture of how a changing parameter affects the final fit via the softening of the margin:

```
X, y = make_blobs(n_samples=100, centers=2, random_state=0, cluster_std=0.8)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))

fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

for axi, C in zip(ax, [10.0, 0.1]):

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

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

   plot_svc_decision_function(model, axi)

   axi.scatter(model.support_vectors_[:, 0],model.support_vectors_[:, 1],s=300, lw=1, facecolors='none')

   axi.set_title('C = {0:.1f}'.format(C), size=14)

plt.show()
```



The optimal value of the parameter will depend on your dataset, and should be tuned using cross-validation or a similar procedure.

Support Vector Machine Summary

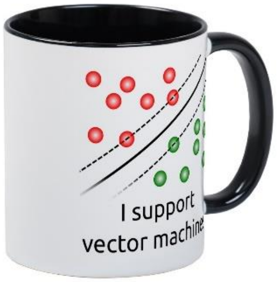
We have seen here a brief intuitive introduction to the principals behind support vector machines. These methods are a powerful classification method for a number of reasons:

- Their dependence on relatively few support vectors means that they are very compact models and take up very little memory.
- Once the model is trained, the prediction phase is very fast.
- Because they are affected only by points near the margin, they work well with high dimensional data; even data with more dimensions than samples - which is a challenge for other algorithms.
- Their integration with kernel methods makes them very versatile, able to adapt to many kind of data.

However, SVM has several disadvantages as well:

- The scaling with the number of samples N is at worst O(N^3), or O(N^2) for efficient implementations. For large numbers of training samples this computational cost can be prohibitive.
- The results are strongly dependent on a suitable choice for the softening parameter C. This must be carefully chosen via cross-validation, which can be expensive as datasets grow in size.

- The results do not have a direct probabilistic interpretation. Although this can be estimated via an internal cross-validation (see the probability parameter of SVC), but this extra estimation is costly.



Conclusion

Given these pros and cons some people only turn to SVMs once other simpler, faster, and less tuning-intensive methods have been shown to be insufficient for the given problem. Nevertheless, if you have the CPU cycles to commit to training and cross-validating an SVM on your data, the method can lead to excellent results.

[Check out if you there is any support to use Google-cloud GPU for SVM]