Introduction

Mastering machine learning algorithms isn’t a myth at all. Most of the beginners start by learning regression. It is simple to learn and use, but does that solve our purpose? Of course not! Because, you can do so much more than just Regression!

Think of machine learning algorithms as an armory packed with axes, sword, blades, bow, dagger etc. You have various tools, but you ought to learn to use them at the right time. As an analogy, think of ‘Regression’ as a sword capable of slicing and dicing data efficiently, but incapable of dealing with highly complex data. On the contrary, ‘Support Vector Machines’ is like a sharp knife – it works on smaller datasets, but on them, it can be much more stronger and powerful in building models.

By now, I hope you’ve now mastered [Random Forest](https://www.analyticsvidhya.com/blog/2015/09/random-forest-algorithm-multiple-challenges/), [Naive Bayes Algorithm](https://www.analyticsvidhya.com/blog/2015/09/naive-bayes-explained/) and [Ensemble Modeling](https://www.analyticsvidhya.com/blog/2015/09/questions-ensemble-modeling/). If not, I’d suggest you to take out few minutes and read about them as well. In this article, I shall guide you through the basics to advanced knowledge of a crucial machine learning algorithm, support vector machines.

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/learn5.jpg)

Table of Contents

1. What is Support Vector Machine?
2. How does it work?
3. How to implement SVM in Python and R?
4. How to tune Parameters of SVM?
5. Pros and Cons associated with SVM

What is Support Vector Machine?

“Support Vector Machine” (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However,  it is mostly used in classification problems. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiate the two classes very well (look at the below snapshot).

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_1.png)

Support Vectors are simply the co-ordinates of individual observation. Support Vector Machine is a frontier which best segregates the two classes (hyper-plane/ line).

You can look at [definition of support vectors](https://www.analyticsvidhya.com/blog/2014/10/support-vector-machine-simplified/) and a few examples of its working here.

How does it work?

Above, we got accustomed to the process of segregating the two classes with a hyper-plane. Now the burning question is “How can we identify the right hyper-plane?”. Don’t worry, it’s not as hard as you think!

Let’s understand:

* **Identify the right hyper-plane (Scenario-1):**Here, we have three hyper-planes (A, B and C). Now, identify the right hyper-plane to classify star and circle.  
  You need to remember a thumb rule to identify the right hyper-plane: “Select the hyper-plane which segregates the two classes better”. In this scenario, hyper-plane “B” has excellently performed this job.
* **Identify the right hyper-plane (Scenario-2):**Here, we have three hyper-planes (A, B and C) and all are segregating the classes well. Now, How can we identify the right hyper-plane?

Here, maximizing the distances between nearest data point (either class) and hyper-plane will help us to decide the right hyper-plane. This distance is called as **Margin**. Let’s look at the below snapshot:[[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_4.png)](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_4.png)Above, you can see that the margin for hyper-plane C is high as compared to both A and B. Hence, we name the right hyper-plane as C. Another lightning reason for selecting the hyper-plane with higher margin is robustness. If we select a hyper-plane having low margin then there is high chance of miss-classification.

* **Identify the right hyper-plane (Scenario-3):**Hint:Use the rules as discussed in previous section to identify the right hyper-plane

**[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_5.png)**Some of you may have selected the hyper-plane **B**as it has higher margin compared to **A.**But, here is the catch, SVM selects the hyper-plane which classifies the classes accurately prior to maximizing margin. Here, hyper-plane B has a classification error and A has classified all correctly. Therefore, the right hyper-plane is **A.**

* **Can we classify two classes (Scenario-4)?:**Below, I am unable to segregate the two classes using a straight line, as one of star lies in the territory of other(circle) class as an outlier.  **[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_61.png)**As I have already mentioned, one star at other end is like an outlier for star class. SVM has a feature to ignore outliers and find the hyper-plane that has maximum margin. Hence, we can say, SVM is robust to outliers.  
  **[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_71.png)**
* **Find the hyper-plane to segregate to classes (Scenario-5):**In the scenario below, we can’t have linear hyper-plane between the two classes, so how does SVM classify these two classes? Till now, we have only looked at the linear hyper-plane.**[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_8.png)**SVM can solve this problem. Easily! It solves this problem by introducing additional feature. Here, we will add a new feature z=x^2+y^2. Now, let’s plot the data points on axis x and z:  
  [[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_9.png)](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_9.png)In above plot, points to consider are:
  + All values for z would be positive always because z is the squared sum of both x and y
  + In the original plot, red circles appear close to the origin of x and y axes, leading to lower value of z and star relatively away from the origin result to higher value of z.

In SVM, it is easy to have a linear hyper-plane between these two classes. But, another burning question which arises is, should we need to add this feature manually to have a hyper-plane. No, SVM has a technique called the [**kernel**](https://en.wikipedia.org/wiki/Kernel_method)**trick**. These are functions which takes low dimensional input space and transform it to a higher dimensional space i.e. it converts not separable problem to separable problem, these functions are called kernels. It is mostly useful in non-linear separation problem. Simply put, it does some extremely complex data transformations, then find out the process to separate the data based on the labels or outputs you’ve defined.

When we look at the hyper-plane in original input space it looks like a circle:  
[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_10.png)

Now, let’s  look at the methods to apply SVM algorithm in a data science challenge.

How to implement SVM in Python and R?

In Python, scikit-learn is a widely used library for implementing machine learning algorithms, SVM is also available in scikit-learn library and follow the same structure (Import library, object creation, fitting model and prediction). Let’s look at the below code:

#Import Library

from sklearn import svm

#Assumed you have, X (predictor) and Y (target) for training data set and x\_test(predictor) of test\_dataset

# Create SVM classification object

model = svm.svc(kernel='linear', c=1, gamma=1)

# there is various option associated with it, like changing kernel, gamma and C value. Will discuss more # about it in next section.Train the model using the training sets and check score

model.fit(X, y)

model.score(X, y)

#Predict Output

predicted= model.predict(x\_test)

The e1071 package in R is used to create Support Vector Machines with ease. It has helper functions as well as code for the Naive Bayes Classifier. The creation of a support vector machine in R and Python follow similar approaches, let’s take a look now at the following code:

#Import Library

require(e1071) #Contains the SVM

Train <- read.csv(file.choose())

Test <- read.csv(file.choose())

# there are various options associated with SVM training; like changing kernel, gamma and C value.

# create model

model <- svm(Target~Predictor1+Predictor2+Predictor3,data=Train,kernel='linear',gamma=0.2,cost=100)

#Predict Output

preds <- predict(model,Test)

table(preds)

How to tune Parameters of SVM?

Tuning parameters value for machine learning algorithms effectively improves the model performance. Let’s look at the list of parameters available with SVM.

sklearn.svm.SVC(*C=1.0*, *kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *shrinking=True*, *probability=False*,*tol=0.001*, *cache\_size=200*, *class\_weight=None*, *verbose=False*, *max\_iter=-1*, *random\_state=None*)

I am going to discuss about some important parameters having higher impact on model performance, “kernel”, “gamma” and “C”.

**kernel**: We have already discussed about it. Here, we have various options available with kernel like, “linear”, “rbf”,”poly” and others (default value is “rbf”).  Here “rbf” and “poly” are useful for non-linear hyper-plane. Let’s look at the example, where we’ve used linear kernel on two feature of iris data set to classify their class.

**Example:**Have linear kernel

import numpy as np

import matplotlib.pyplot as plt

from sklearn import svm, datasets

# import some data to play with

iris = datasets.load\_iris()

X = iris.data[:, :2] # we only take the first two features. We could

# avoid this ugly slicing by using a two-dim dataset

y = iris.target

# we create an instance of SVM and fit out data. We do not scale our

# data since we want to plot the support vectors

C = 1.0 # SVM regularization parameter

svc = svm.SVC(kernel='linear', C=1,gamma=0).fit(X, y)

# create a mesh to plot in

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

h = (x\_max / x\_min)/100

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h),

np.arange(y\_min, y\_max, h))

plt.subplot(1, 1, 1)

Z = svc.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)

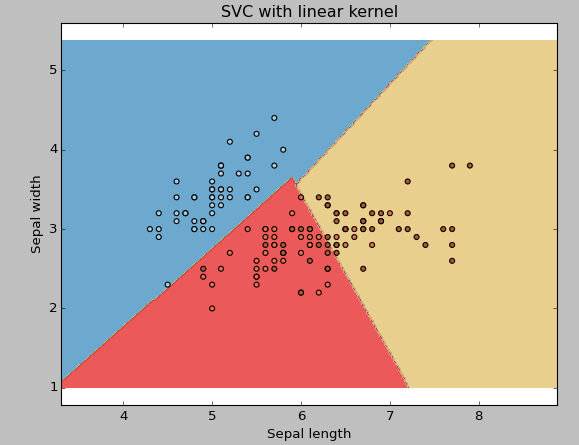
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)

plt.xlabel('Sepal length')

plt.ylabel('Sepal width')

plt.xlim(xx.min(), xx.max())

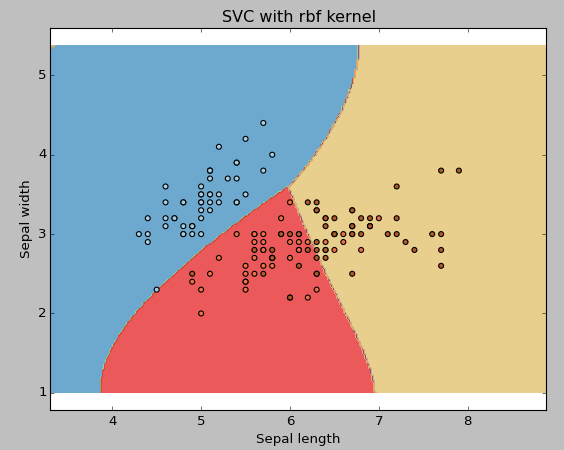
plt.title('SVC with linear kernel')

plt.show()[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_111.png)

**Example:**Have rbf kernel

Change the kernel type to rbf in below line and look at the impact.

svc = svm.SVC(kernel='rbf', C=1,gamma=0).fit(X, y)

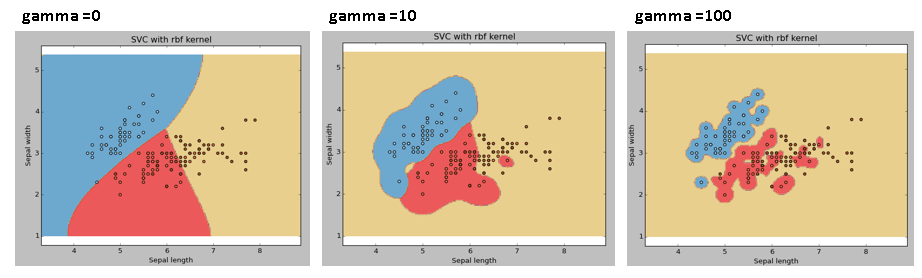
[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_12.png)

I would suggest you to go for linear kernel if you have large number of features (>1000) because it is more likely that the data is linearly separable in high dimensional space. Also, you can RBF but do not forget to cross validate for its parameters as to avoid over-fitting.

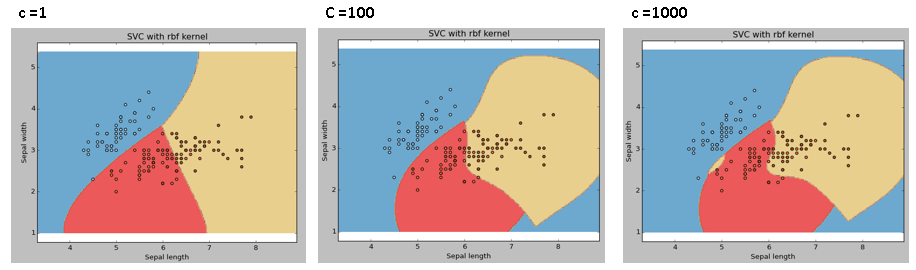
**gamma**: Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’. Higher the value of gamma, will try to exact fit the as per training data set i.e. generalization error and cause over-fitting problem.

**Example:**Let’s difference if we have gamma different gamma values like 0, 10 or 100.

svc = svm.SVC(kernel='rbf', C=1,gamma=0).fit(X, y)

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_15.png)

**C:**Penalty parameter C of the error term. It also controls the trade off between smooth decision boundary and classifying the training points correctly.

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_18.png)

We should always look at the cross validation score to have effective combination of these parameters and avoid over-fitting.

In R, SVMs can be tuned in a similar fashion as they are in Python. Mentioned below are the respective parameters for e1071 package:

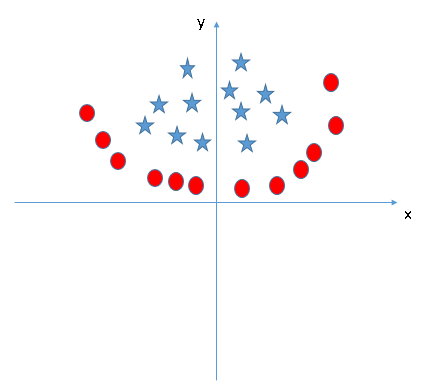
* The kernel parameter can be tuned to take “Linear”,”Poly”,”rbf” etc.
* The gamma value can be tuned by setting the “Gamma” parameter.
* The C value in Python is tuned by the “Cost” parameter in R.

Pros and Cons associated with SVM

* **Pros:**
  + It works really well with clear margin of separation
  + It is effective in high dimensional spaces.
  + It is effective in cases where number of dimensions is greater than the number of samples.
  + It uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
* **Cons:**
  + It doesn’t perform well, when we have large data set because the required training time is higher
  + It also doesn’t perform very well, when the data set has more noise i.e. target classes are overlapping
  + SVM doesn’t directly provide probability estimates, these are calculated using an expensive five-fold cross-validation. It is related SVC method of Python scikit-learn library.

Practice Problem

Find right additional feature to have a hyper-plane for segregating the classes in below snapshot:

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_19.png)

Answer the variable name in the comments section below. I’ll shall then reveal the answer.

End Notes

In this article, we looked at the machine learning algorithm, Support Vector Machine in detail.  I discussed its concept of working, process of implementation in python, the tricks to make the model efficient by tuning its parameters, Pros and Cons, and finally a problem to solve. I would suggest you to use SVM and analyse the power of this model by tuning the parameters. I also want to hear your experience with SVM, how have you tuned parameters to avoid over-fitting and reduce the training time?

# n-Depth: Support Vector Machines

< [In Depth: Linear Regression](https://jakevdp.github.io/PythonDataScienceHandbook/05.06-linear-regression.html) | [Contents](https://jakevdp.github.io/PythonDataScienceHandbook/index.html) | [In-Depth: Decision Trees and Random Forests](https://jakevdp.github.io/PythonDataScienceHandbook/05.08-random-forests.html) >

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:

In [1]:

%**matplotlib** inline

**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

As part of our disussion of Bayesian classification (see [In Depth: Naive Bayes Classification](https://jakevdp.github.io/PythonDataScienceHandbook/05.05-naive-bayes.html)), we learned a simple model describing the distribution of each underlying class, and used these generative models to probabilistically determine labels for new points. That was an example of generative classification; here we will consider instead discriminative classification: rather than modeling each class, we simply find a line or curve (in two dimensions) or manifold (in multiple dimensions) that divides the classes from each other.

As an example of this, consider the simple case of a classification task, in which the two classes of points are well separated:

In [2]:

**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:

In [3]:

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);

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.

## Support Vector Machines: Maximizing the Margin

Support vector machines offer one way to improve on this. The intuition is this: rather than simply drawing a zero-width line between the classes, we can draw around each line a margin of some width, up to the nearest point. Here is an example of how this might look:

In [4]:

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 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).

In [5]:

**from** **sklearn.svm** **import** SVC *# "Support vector classifier"*

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

model.fit(X, y)

Out[5]:

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:

In [6]:

**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 [7]:

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

plot\_svc\_decision\_function(model);

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:

In [8]:

model.support\_vectors\_

Out[8]:

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:

In [9]:

**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 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 vectors 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.

If you are running this notebook live, you can use IPython's interactive widgets to view this feature of the SVM model interactively:

In [10]:

**from** **ipywidgets** **import** interact, fixed

interact(plot\_svm, N=[10, 200], ax=fixed(**None**));

### Beyond linear boundaries: Kernel SVM

Where SVM becomes extremely powerful is when it is combined with kernels. We have seen a version of kernels before, in the basis function regressions of [In Depth: Linear Regression](https://jakevdp.github.io/PythonDataScienceHandbook/05.06-linear-regression.html). 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:

In [11]:

**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**);

It is clear that no linear discrimination will ever be able to separate this data. But we can draw a lesson from the basis function regressions in [In Depth: Linear Regression](https://jakevdp.github.io/PythonDataScienceHandbook/05.06-linear-regression.html), and 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 functioncentered on the middle clump:

In [12]:

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

We can visualize this extra data dimension using a three-dimensional plot—if you are running this notebook live, you will be able to use the sliders to rotate the plot:

In [13]:

**from** **mpl\_toolkits** **import** mplot3d

**def** plot\_3D(elev=30, azim=30, X=X, y=y):

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

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

ax.view\_init(elev=elev, azim=azim)

ax.set\_xlabel('x')

ax.set\_ylabel('y')

ax.set\_zlabel('r')

interact(plot\_3D, elev=[-90, 90], azip=(-180, 180),

X=fixed(X), y=fixed(y));

We can see that with this additional dimension, the data becomes trivially linearly separable, by drawing a separating plane at, say, 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 sift 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 NN points into NN dimensions—is that it might become very computationally intensive as NN grows large. However, because of a neat little procedure known as the [kernel trick](https://en.wikipedia.org/wiki/Kernel_trick), a fit on kernel-transformed data can be done implicitly—that is, without ever building the full NN-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 our linear kernel to an RBF (radial basis function) kernel, using the kernel model hyperparameter:

In [14]:

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

clf.fit(X, y)

Out[14]:

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 [15]:

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');

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:

In [16]:

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

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

In [17]:

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)

The optimal value of the CC parameter will depend on your dataset, and should be tuned using cross-validation or a similar procedure (refer back to [Hyperparameters and Model Validation](https://jakevdp.github.io/PythonDataScienceHandbook/05.03-hyperparameters-and-model-validation.html)).

## Example: Face Recognition

As an example of support vector machines in action, let's take a look at the facial recognition problem. We will use the Labeled Faces in the Wild dataset, which consists of several thousand collated photos of various public figures. A fetcher for the dataset is built into Scikit-Learn:

In [18]:

**from** **sklearn.datasets** **import** fetch\_lfw\_people

faces = fetch\_lfw\_people(min\_faces\_per\_person=60)

print(faces.target\_names)

print(faces.images.shape)

['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'

'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']

(1348, 62, 47)

Let's plot a few of these faces to see what we're working with:

In [19]:

fig, ax = plt.subplots(3, 5)

**for** i, axi **in** enumerate(ax.flat):

axi.imshow(faces.images[i], cmap='bone')

axi.set(xticks=[], yticks=[],

xlabel=faces.target\_names[faces.target[i]])

Each image contains [62×47] or nearly 3,000 pixels. We could proceed by simply using each pixel value as a feature, but often it is more effective to use some sort of preprocessor to extract more meaningful features; here we will use a principal component analysis (see [In Depth: Principal Component Analysis](https://jakevdp.github.io/PythonDataScienceHandbook/05.09-principal-component-analysis.html)) to extract 150 fundamental components to feed into our support vector machine classifier. We can do this most straightforwardly by packaging the preprocessor and the classifier into a single pipeline:

In [20]:

**from** **sklearn.svm** **import** SVC

**from** **sklearn.decomposition** **import** RandomizedPCA

**from** **sklearn.pipeline** **import** make\_pipeline

pca = RandomizedPCA(n\_components=150, whiten=**True**, random\_state=42)

svc = SVC(kernel='rbf', class\_weight='balanced')

model = make\_pipeline(pca, svc)

For the sake of testing our classifier output, we will split the data into a training and testing set:

In [21]:

**from** **sklearn.cross\_validation** **import** train\_test\_split

Xtrain, Xtest, ytrain, ytest = train\_test\_split(faces.data, faces.target,

random\_state=42)

Finally, we can use a grid search cross-validation to explore combinations of parameters. Here we will adjust C (which controls the margin hardness) and gamma (which controls the size of the radial basis function kernel), and determine the best model:

In [22]:

**from** **sklearn.grid\_search** **import** GridSearchCV

param\_grid = {'svc\_\_C': [1, 5, 10, 50],

'svc\_\_gamma': [0.0001, 0.0005, 0.001, 0.005]}

grid = GridSearchCV(model, param\_grid)

%**time** grid.fit(Xtrain, ytrain)

print(grid.best\_params\_)

CPU times: user 47.8 s, sys: 4.08 s, total: 51.8 s

Wall time: 26 s

{'svc\_\_gamma': 0.001, 'svc\_\_C': 10}

The optimal values fall toward the middle of our grid; if they fell at the edges, we would want to expand the grid to make sure we have found the true optimum.

Now with this cross-validated model, we can predict the labels for the test data, which the model has not yet seen:

In [23]:

model = grid.best\_estimator\_

yfit = model.predict(Xtest)

Let's take a look at a few of the test images along with their predicted values:

In [24]:

fig, ax = plt.subplots(4, 6)

**for** i, axi **in** enumerate(ax.flat):

axi.imshow(Xtest[i].reshape(62, 47), cmap='bone')

axi.set(xticks=[], yticks=[])

axi.set\_ylabel(faces.target\_names[yfit[i]].split()[-1],

color='black' **if** yfit[i] == ytest[i] **else** 'red')

fig.suptitle('Predicted Names; Incorrect Labels in Red', size=14);

Out of this small sample, our optimal estimator mislabeled only a single face (Bush’s face in the bottom row was mislabeled as Blair). We can get a better sense of our estimator's performance using the classification report, which lists recovery statistics label by label:

In [25]:

**from** **sklearn.metrics** **import** classification\_report

print(classification\_report(ytest, yfit,

target\_names=faces.target\_names))

precision recall f1-score support

Ariel Sharon 0.65 0.73 0.69 15

Colin Powell 0.81 0.87 0.84 68

Donald Rumsfeld 0.75 0.87 0.81 31

George W Bush 0.93 0.83 0.88 126

Gerhard Schroeder 0.86 0.78 0.82 23

Hugo Chavez 0.93 0.70 0.80 20

Junichiro Koizumi 0.80 1.00 0.89 12

Tony Blair 0.83 0.93 0.88 42

avg / total 0.85 0.85 0.85 337

We might also display the confusion matrix between these classes:

In [26]:

**from** **sklearn.metrics** **import** confusion\_matrix

mat = confusion\_matrix(ytest, yfit)

sns.heatmap(mat.T, square=**True**, annot=**True**, fmt='d', cbar=**False**,

xticklabels=faces.target\_names,

yticklabels=faces.target\_names)

plt.xlabel('true label')

plt.ylabel('predicted label');

This helps us get a sense of which labels are likely to be confused by the estimator.

For a real-world facial recognition task, in which the photos do not come pre-cropped into nice grids, the only difference in the facial classification scheme is the feature selection: you would need to use a more sophisticated algorithm to find the faces, and extract features that are independent of the pixellation. For this kind of application, one good option is to make use of [OpenCV](http://opencv.org/), which, among other things, includes pre-trained implementations of state-of-the-art feature extraction tools for images in general and faces in particular.

## 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:

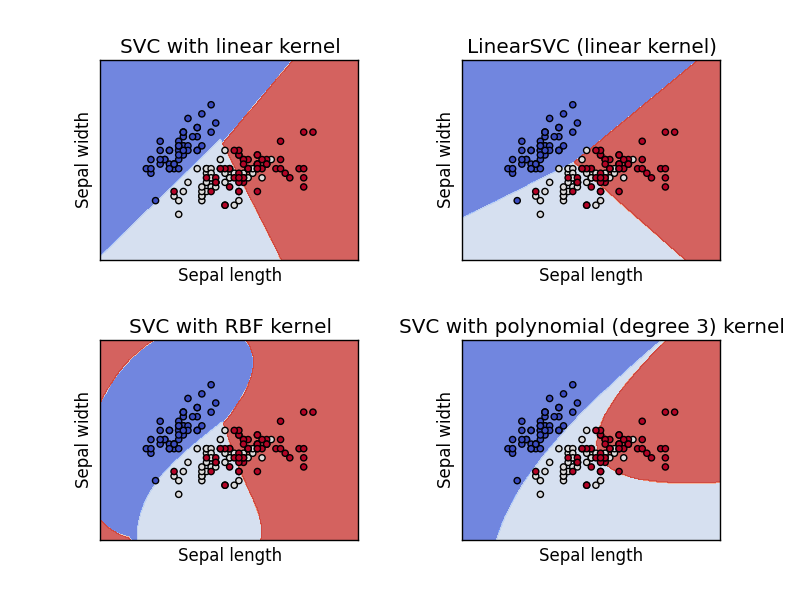
* Their dependence on relatively few support vectors means that they are very compact models, and take up very little memory.
* Once the model is trained, the prediction phase is very fast.
* Because they are affected only by points near the margin, they work well with high-dimensional data—even data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

However, SVMs have several disadvantages as well:

* The scaling with the number of samples NN is O[N3]O[N3] at worst, or O[N2]O[N2]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 CC. 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. This can be estimated via an internal cross-validation (see the probability parameter of SVC), but this extra estimation is costly.

With those traits in mind, I generally only turn to SVMs once other simpler, faster, and less tuning-intensive methods have been shown to be insufficient for my needs. 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.

|  |
| --- |
| **import** matplotlib  matplotlib.use('GTKAgg')    **import** numpy **as** np  **import** matplotlib.pyplot **as** plt  **from** sklearn **import** svm, datasets    *# import some data to play with*  iris = datasets.load\_iris()  X = iris.data[:, :2] *# we only take the first two features. We could*  *# avoid this ugly slicing by using a two-dim dataset*  y = iris.target    h = .02 *# step size in the mesh*    *# we create an instance of SVM and fit out data. We do not scale our*  *# data since we want to plot the support vectors*  C = 1.0 *# SVM regularization parameter*  svc = svm.SVC(kernel='linear', C=C).fit(X, y)  rbf\_svc = svm.SVC(kernel='rbf', gamma=0.7, C=C).fit(X, y)  poly\_svc = svm.SVC(kernel='poly', degree=3, C=C).fit(X, y)  lin\_svc = svm.LinearSVC(C=C).fit(X, y)    *# create a mesh to plot in*  x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1  y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1  xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h),  np.arange(y\_min, y\_max, h))    *# title for the plots*  titles = ['SVC with linear kernel',  'LinearSVC (linear kernel)',  'SVC with RBF kernel',  'SVC with polynomial (degree 3) kernel']      **for** i, clf **in** enumerate((svc, lin\_svc, rbf\_svc, poly\_svc)):  *# Plot the decision boundary. For that, we will assign a color to each*  *# point in the mesh [x\_min, x\_max]x[y\_min, y\_max].*  plt.subplot(2, 2, i + 1)  plt.subplots\_adjust(wspace=0.4, hspace=0.4)    Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])    *# Put the result into a color plot*  Z = Z.reshape(xx.shape)  plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)    *# Plot also the training points*  plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm)  plt.xlabel('Sepal length')  plt.ylabel('Sepal width')  plt.xlim(xx.min(), xx.max())  plt.ylim(yy.min(), yy.max())  plt.xticks(())  plt.yticks(())  plt.title(titles[i])    plt.show() |

[](https://pythonspot-9329.kxcdn.com/wp-content/uploads/2017/03/svm-classify.png)

<https://www.svm-tutorial.com/2014/10/support-vector-regression-r/>