**We're going to be using the SVC (support vector classifier) SVM (support vector machine). Our kernel is going to be linear, and C is equal to 1.0**

**Support vector machines** (**SVMs**, also **support vector networks**[1]) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier (although methods such as Platt scaling exist to use SVM in a probabilistic classification setting). An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.

When data are not labeled, supervised learning is not possible, and an unsupervised learning approach is required, which attempts to find natural clustering of the data to groups, and then map new data to these formed groups. The **support vector clustering**[2] algorithm created by Hava Siegelmann and Vladimir Vapnik, applies the statistics of support vectors, developed in the support vector machines algorithm, to categorize unlabeled data, and is one of the most widely used clustering algorithms in industrial applications.

Basic dependencies:

Matplotlib here is not truly necessary for Linear SVC. The reason why we're using it here is for the eventual data visualization

import numpy as np

import matplotlib.pyplot as plt

from matplotlib import style

style.use("ggplot")

from sklearn import svm

let's consider that we have two features to consider. These features will be visualized as axis on our graph. So something like:

x = [1, 5, 1.5, 8, 1, 9]

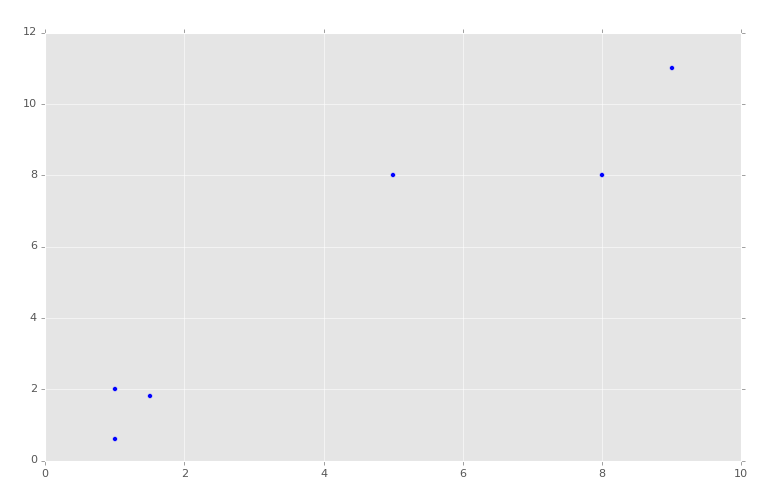
y = [2, 8, 1.8, 8, 0.6, 11]

let's consider that we have two features to consider. These features will be visualized as axis on our graph. So something like:

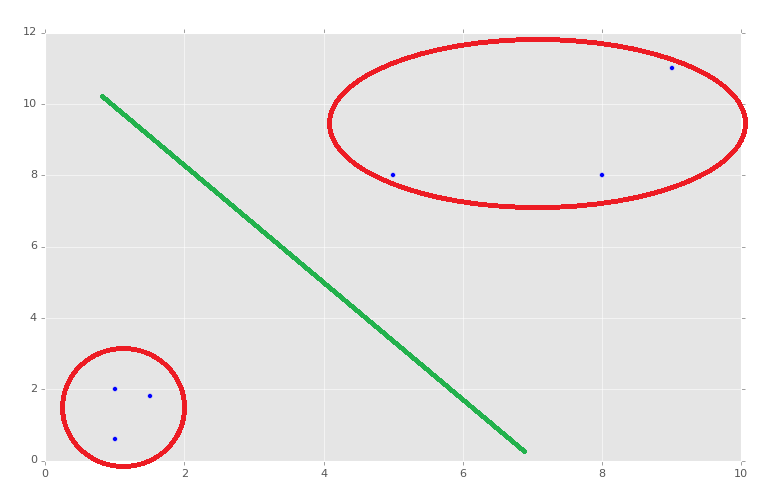
x = [1, 5, 1.5, 8, 1, 9]

y = [2, 8, 1.8, 8, 0.6, 11]

The result is:



Now, of course, we can see with our own eyes how these groups should be divided, though exactly where we might draw the dividing line might be debated:



Generally, you will see the feature list being stored in a capital X variable. Let's translate our above x and y coordinates into an array that is compiled of the x and y coordinates, where x is a feature and y is a feature.

X = np.array([[1,2],

[5,8],

[1.5,1.8],

[8,8],

[1,0.6],

[9,11]])

Now that we have this array, we need to label it for training purposes. There are forms of machine learning called "unsupervised learning," where data labeling isn't used, as is the case with clustering, though this example is a form of supervised learning.

For our labels, sometimes referred to as "targets," we're going to use 0 or 1.

y = [0,1,0,1,0,1]

Just by looking at our data set, we can see we have coordinate pairs that are "low" numbers and coordinate pairs that are "higher" numbers. We've then assigned 0 to the lower coordinate pairs and 1 to the higher feature pairs.

These are the labels. In the case of our project, we will wind up having a list of numerical features that are various statistics about stock companies, and then the "label" will be either a 0 or a 1, where 0 is under-perform the market and a 1 is out-perform the market.

Moving along, we are now going to define our classifier:

clf = svm.SVC(kernel='linear', C = 1.0)

We're going to be using the SVC (support vector classifier) SVM (support vector machine). Our kernel is going to be linear, and C is equal to 1.0. What is C you ask? Don't worry about it for now, but, if you must know, C is a valuation of "how badly" you want to properly classify, or fit, everything. The machine learning field is relatively new, and experimental. There exist many debates about the value of C, as well as how to calculate the value for C. We're going to just stick with 1.0 for now, which is a nice default parameter.

Next, we call:

clf.fit(X,y)

Note: this is an older tutorial, and Scikit-Learn has since deprecated this method. By version 0.19, this code will cause an error because it needs to be a numpy array, and re-shaped. To see an example of converting to a NumPy array and reshaping, check out this [**K Nearest Neighbors**](https://pythonprogramming.net/k-nearest-neighbors-application-machine-learning-tutorial/) tutorial, near the end. You do not need to follow along with that series to mimic what is done there with the reshaping, and continue along with this series.

From here, the learning is done. It should be nearly-instant, since we have such a small data set.

Next, we can predict and test. Let's print a prediction:

print(clf.predict([0.58,0.76]))

We're hoping this predicts a 0, since this is a "lower" coordinate pair.

Sure enough, the prediction is a classification of 0. Next, what if we do:

print(clf.predict([10.58,10.76]))

And again, we have a theoretically correct answer of 1 as the classification. This was a blind prediction, though it was really a test as well, since we knew what the hopeful target was. Congratulations, you have 100% accuracy!

Now, to visualize your data:

w = clf.coef\_[0]

print(w)

a = -w[0] / w[1]

xx = np.linspace(0,12)

yy = a \* xx - clf.intercept\_[0] / w[1]

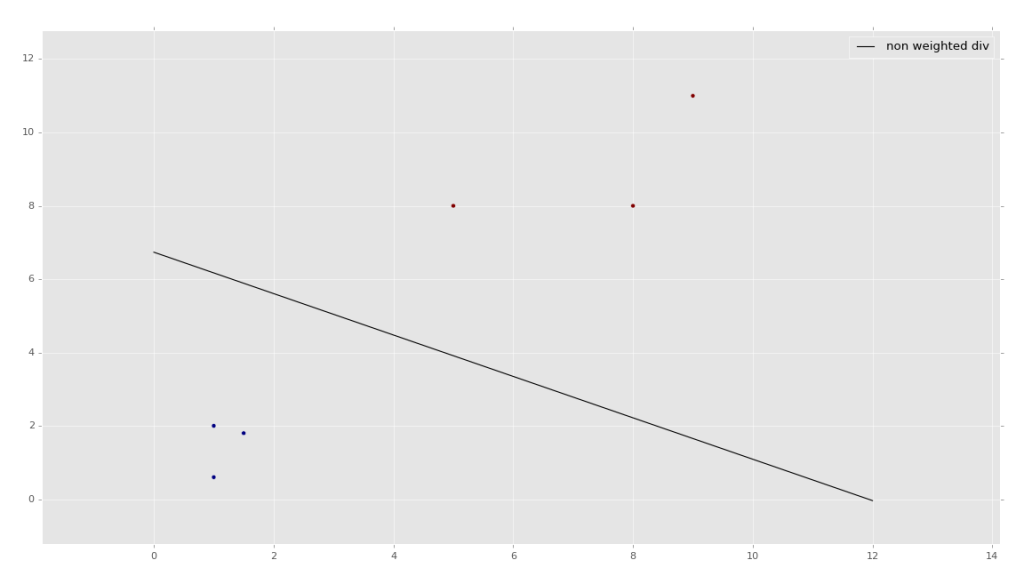
h0 = plt.plot(xx, yy, 'k-', label="non weighted div")

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

plt.legend()

plt.show()

The result:



If you'd like a bit more explanation on how the graphing code works, watch the second-half of the embedded video. Visualizing the data is somewhat useful to see what the program is doing in the background, but is not really necessary to understand how to visualize it specifically at this point. You will likely find that the problems you are trying to solve simply cannot be visualized due to having too many features and thus too many dimensions to graph.

Feel free to play around with the code and test more samples. Adjust gamma a bit. You should notice speed goes up the larger gamma, but accuracy declines. You should notice the opposite if you decrease gamma (do this by factors of 10).

So, something like:

clf = svm.SVC(gamma=0.01, C=100)

This is likely to be inaccurate, or less accurate than before.

clf = svm.SVC(gamma=0.0001, C=100)

The above here is likely to be more accurate, as it is basically "paying more attention to detail."

**Example:**

**SVM for Classification Problems**

The [iris dataset](https://archive.ics.uci.edu/ml/datasets/iris) is a simple dataset of contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other two; the latter are NOT linearly separable from each other. Each instance has 4 features:

1. sepal length
2. sepal width
3. petal length
4. petal width

A typical problem to solve is to predict the *class* of the iris plant based on these 4 features. For brevity and visualization, in this example we will be using only the first two features.

import pandas as pd

import numpy as np

from sklearn import svm, datasets

import matplotlib.pyplot as plt

%matplotlib inline

iris = datasets.load\_iris()

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

y = iris.target

# Plot resulting Support Vector boundaries with original data

# Create fake input data for prediction that we will use for plotting

# 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))

X\_plot = np.c\_[xx.ravel(), yy.ravel()]

# Create the SVC model object

C = 1.0 # SVM regularization parameter

svc = svm.SVC(kernel='linear', C=C, decision\_function\_shape='ovr').fit(X, y)

Z = svc.predict(X\_plot)

Z = Z.reshape(xx.shape)

plt.figure(figsize=(15, 5))

plt.subplot(121)

plt.contourf(xx, yy, Z, cmap=plt.cm.tab10, alpha=0.3)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1)

plt.xlabel('Sepal length')

plt.ylabel('Sepal width')

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

plt.title('SVC with linear kernel')

# Create the SVC model object

C = 1.0 # SVM regularization parameter

svc = svm.SVC(kernel='rbf', C=C, decision\_function\_shape='ovr').fit(X, y)

Z = svc.predict(X\_plot)

Z = Z.reshape(xx.shape)

plt.subplot(122)

plt.contourf(xx, yy, Z, cmap=plt.cm.tab10, alpha=0.3)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1)

plt.xlabel('Sepal length')

plt.ylabel('Sepal width')

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

plt.title('SVC with RBF kernel')

plt.show()