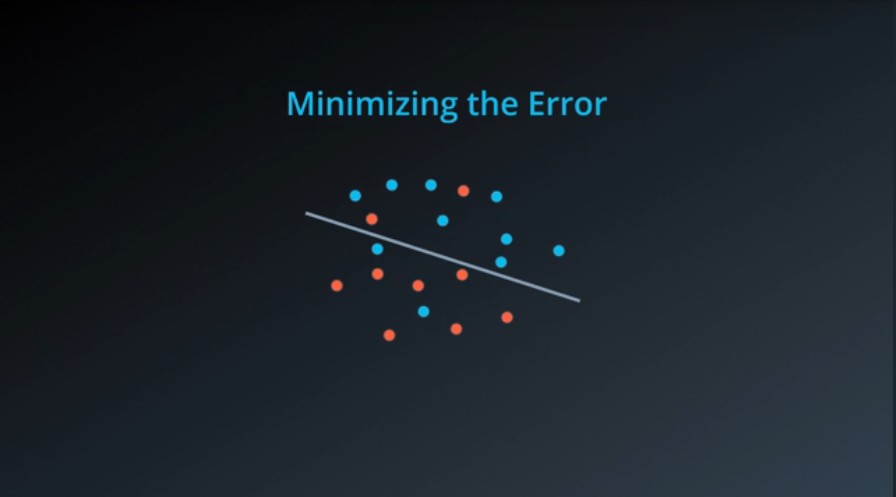
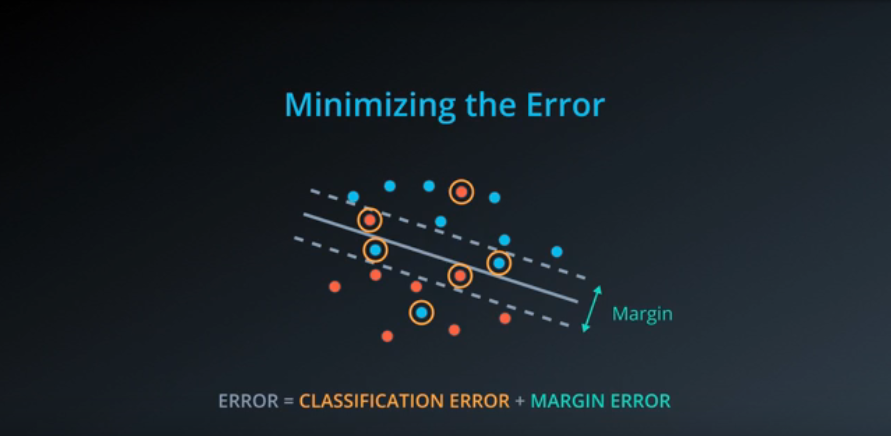
**Support Vector Machines**

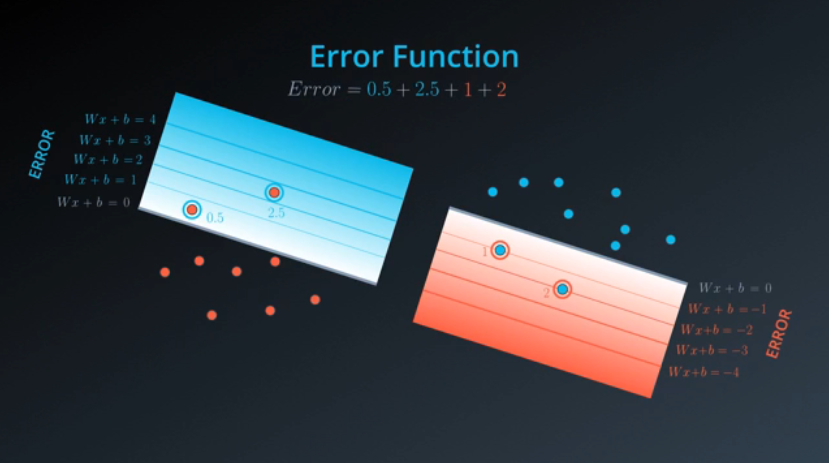
For SVM’s we want the boundary to be the furthest possible away from all the points. A normal classification algorithm would split a data set something like this:

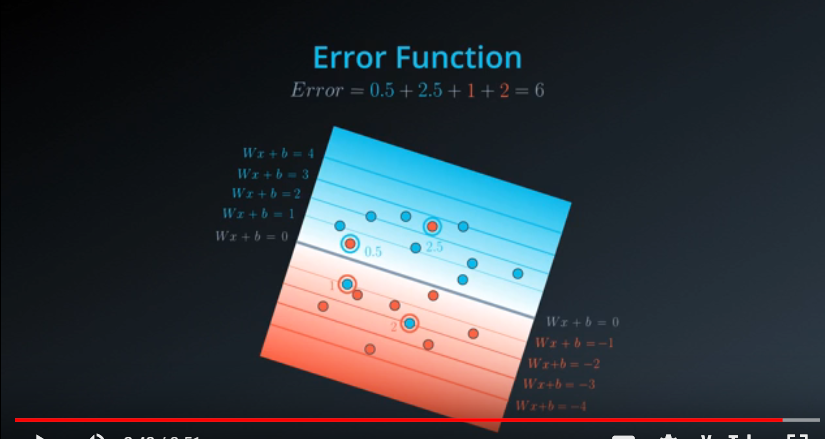


However a SVM will take a look at the original line as well as taking into account points that are in a marginal distance from this line:

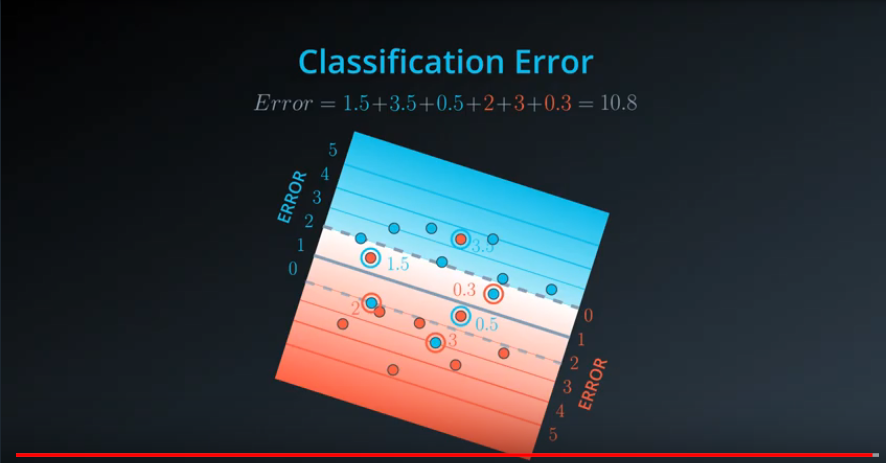


To build this error function we would make use of the “Perceptron Algorithm”. Using this we can calculate the error at every point of a normal model (not an SVM model):





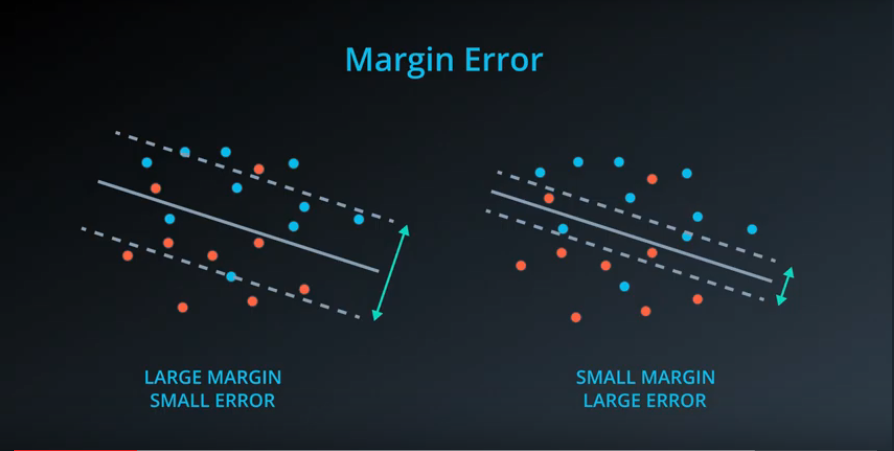
To use this with an SVM model we will do something similar while making use of the margin we had used previously. In this example the blue error starts from the bottom dotted line and the red error starts from the top dotted line.

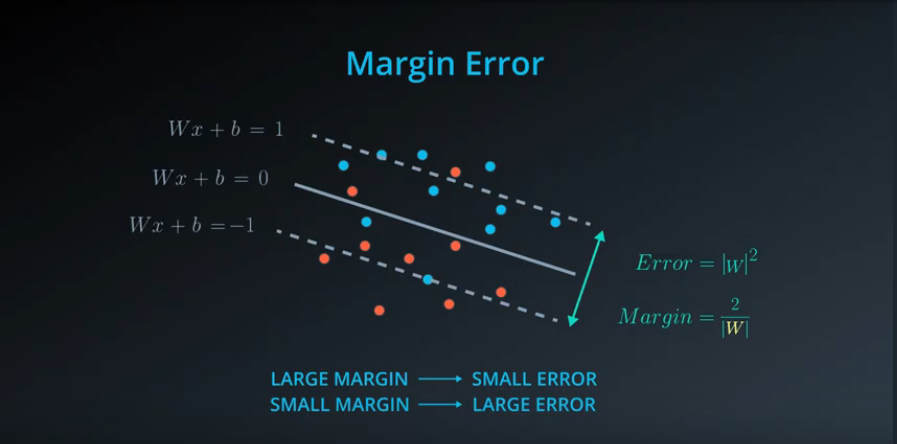


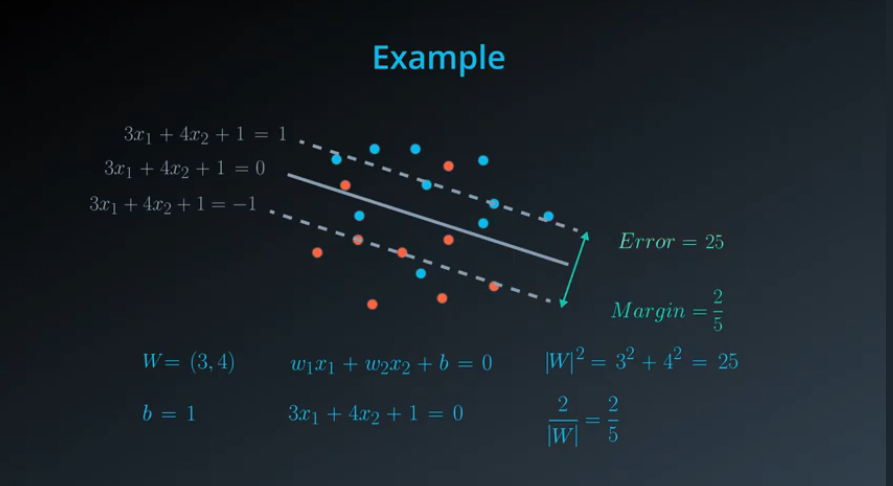
In a nutshell this is how we find the classification error for SVM’s.

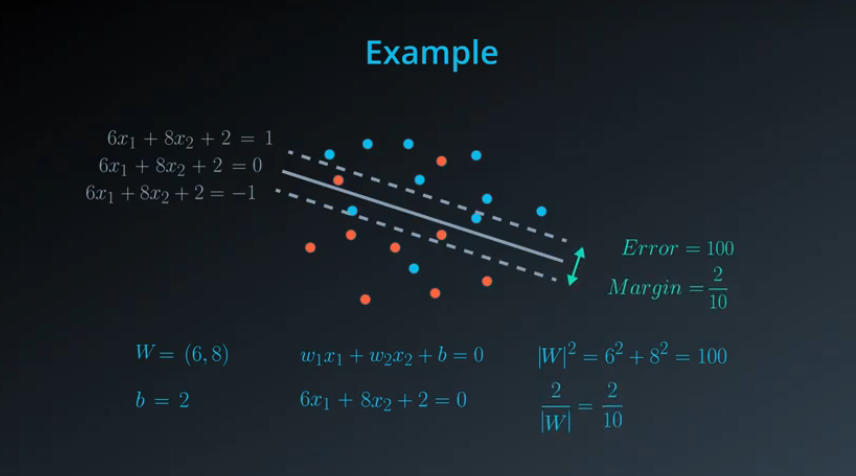
**Margin Error**

We can make decisions for our SVM models on how small or large we want our margins to be:



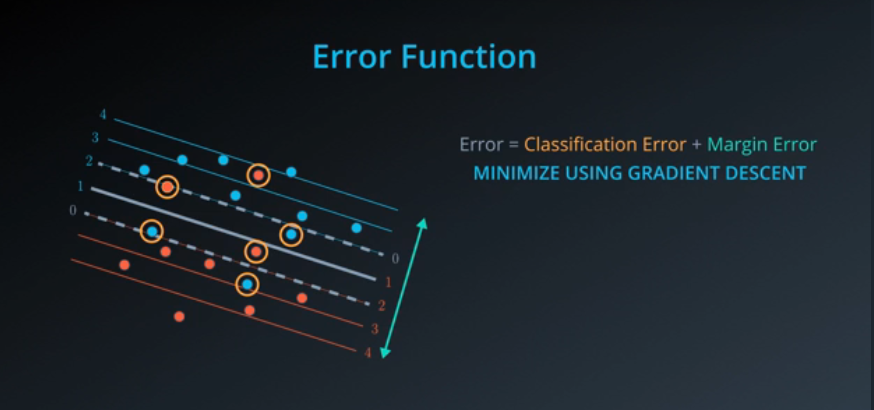








The classification error and margin error combined will give us our error function

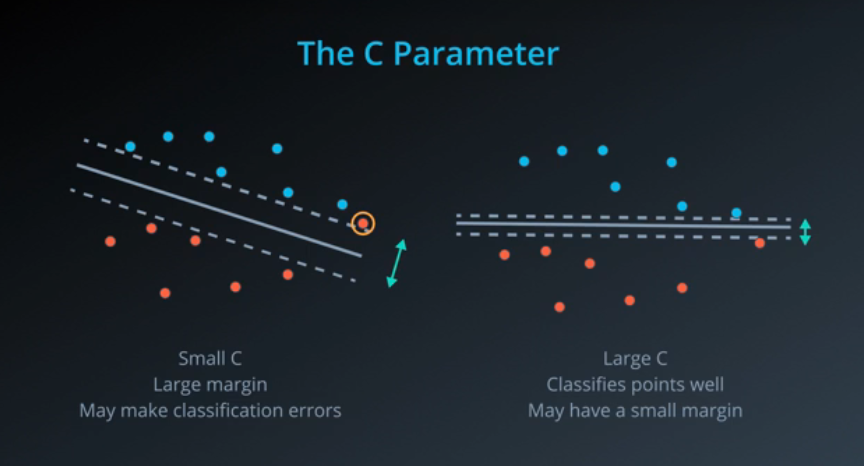


**The C Parameter**

The C parameter is a hyperparameter that is used in SVM’s. This C parameter is a constant that attaches itself to the classification error.

* If we have a large C we care more about classifying points then getting a good (large)margins
* If we have a small C we care more about getting a large margin then correctly classifying every point.

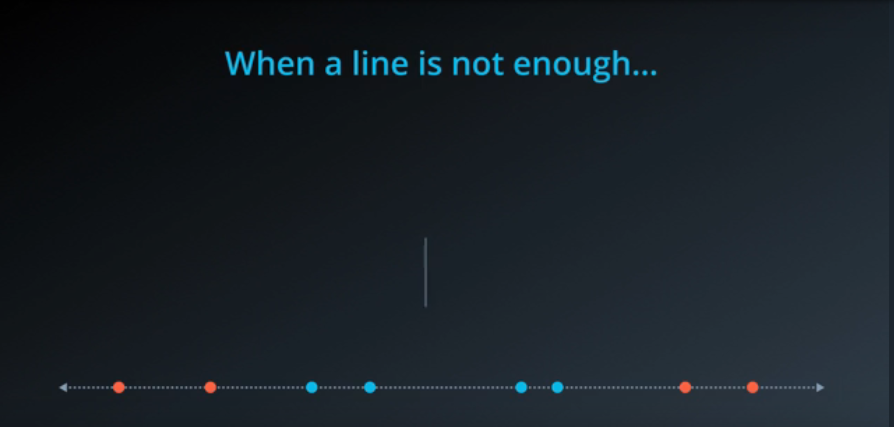
We will use techniques such as GridSearch to help us to tune this hyperparameter.



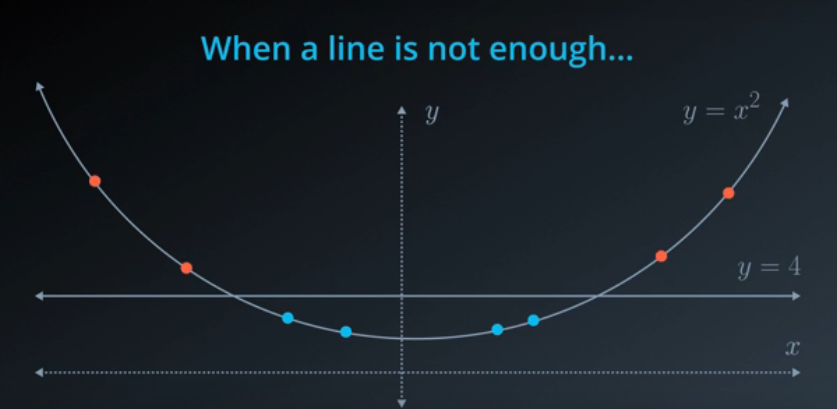
**Kernels –** Kernels are a way of taking linear data, morphing it into a different type of data and then being able to create models around it.

**Polynomial Kernels**

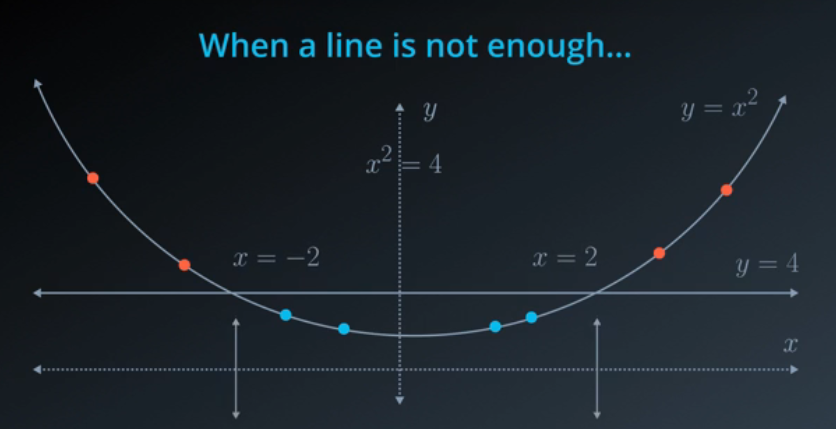
For example if we had the following arrangement of points we wouldn’t be able to split our points with one line:



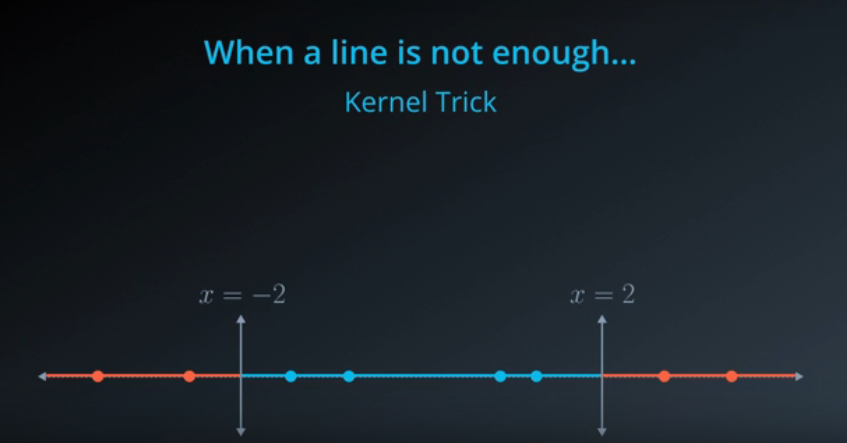
If we add a y-axis and add a parabola we could have the following where our lines become nicely separable:



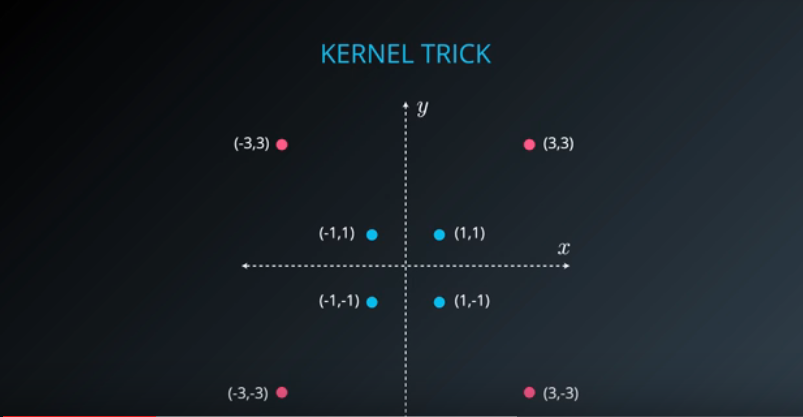
We can combine these two equations to get x^2=4 which gives us x=2 and x=-2.



Bringing this back to the line they split the data very well, this is the Kernel trick:

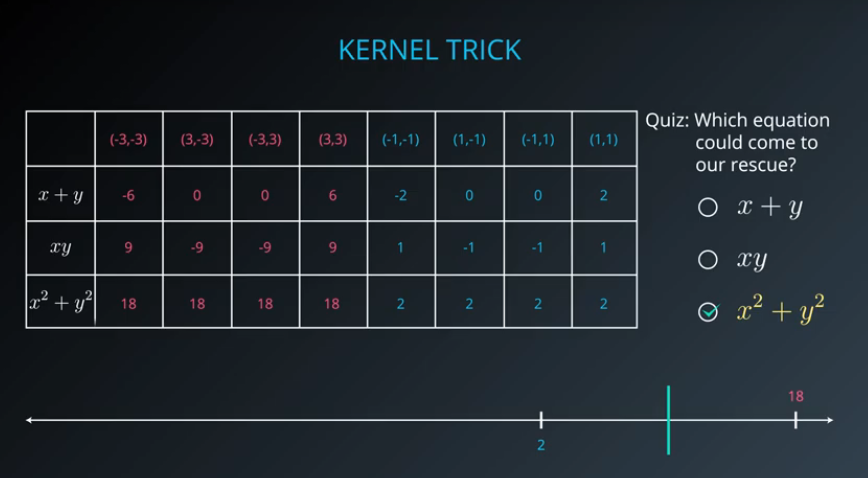


What if we wanted to split the following data set, we would use the kernel trick again:



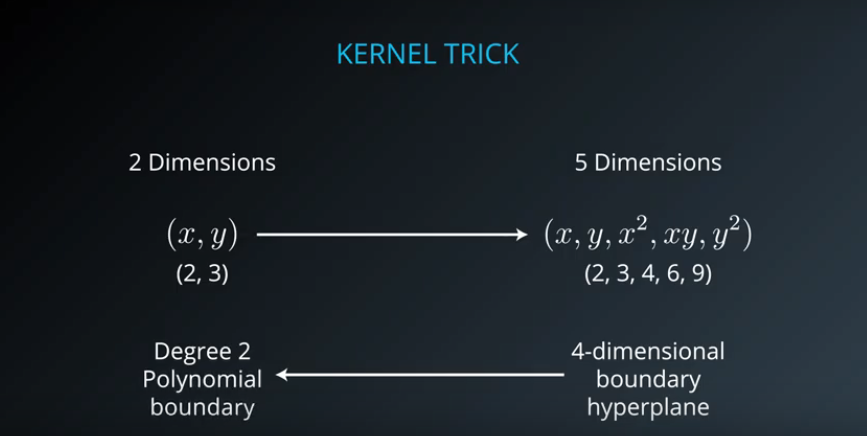
We will try to use a circular boundary to separate the blue points from the red points. To do this we will use a higher degree polynomial equation.

For this example the best function to use would be the x^2 + y^2:

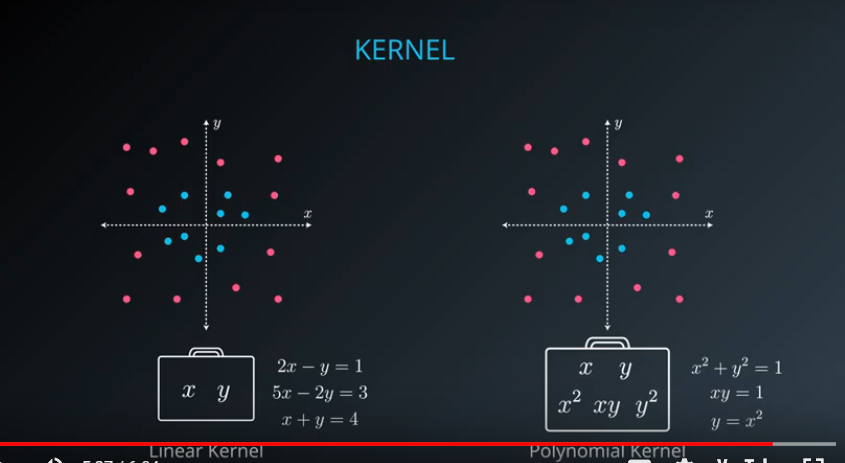


In this specific example the best circle to split this data would be x^2+y^2=10.

In summary if we have set of points that can’t be split linearly we will try to bring it to higher dimensions. If we can split it in these higher dimensions we will then project that split back into the original 2 dimensional plane:



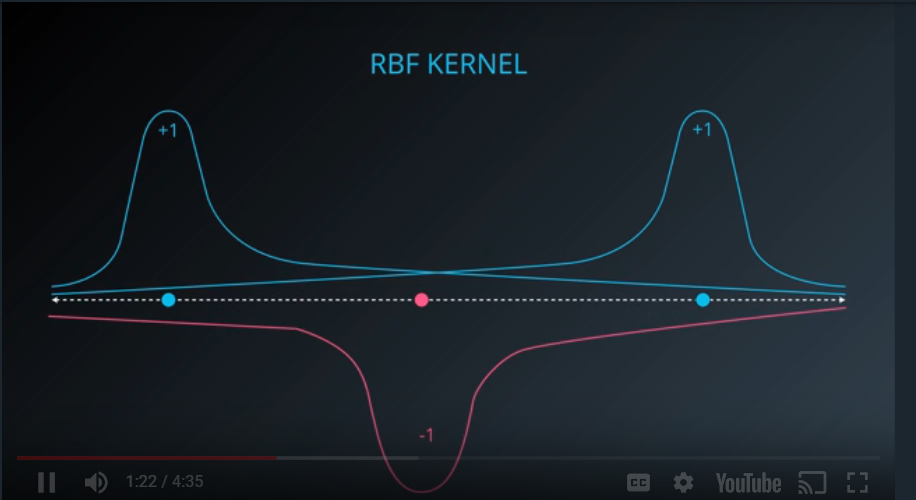
With just x and y we can only really create lines. If we allow ourselves to extend to a polynomial kenrel we can build stronger functions to separate the data:



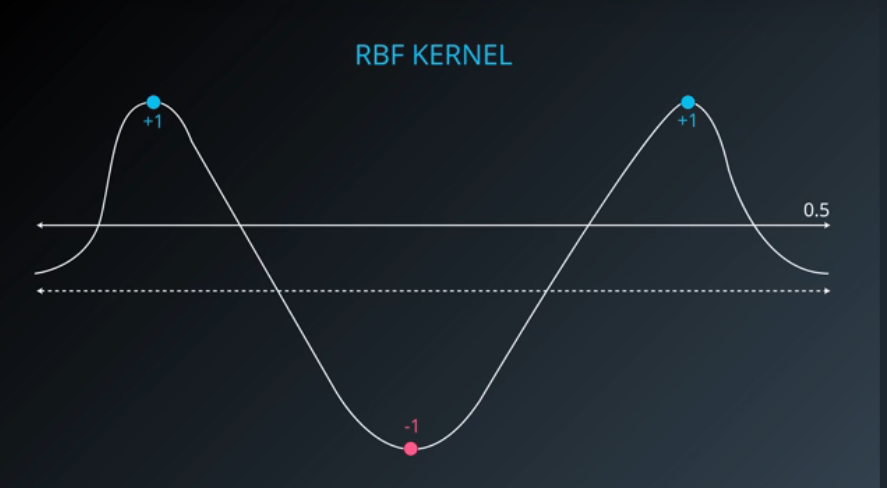
The degree of a polynomial kernel is something that can be changed as a hyperparameter of our model.

**RBF (Radial Basis Function)** **Kernels**

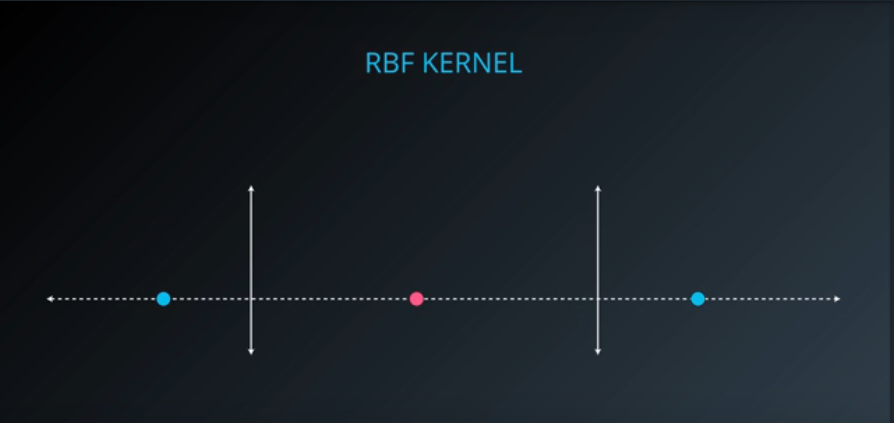
For RBF kernels imagine imagine we draw a mountain at every point and then create a function from this. These mountains are called radial basis functions.



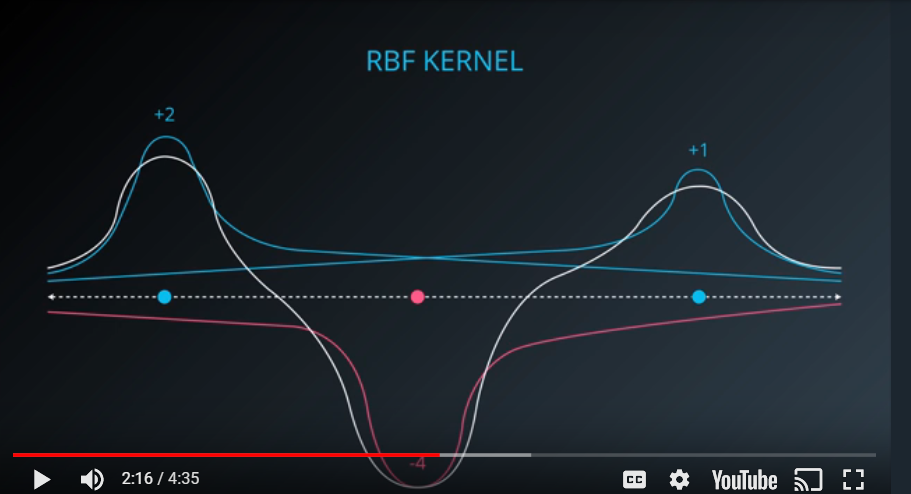
If we add each of these functions we get the following:



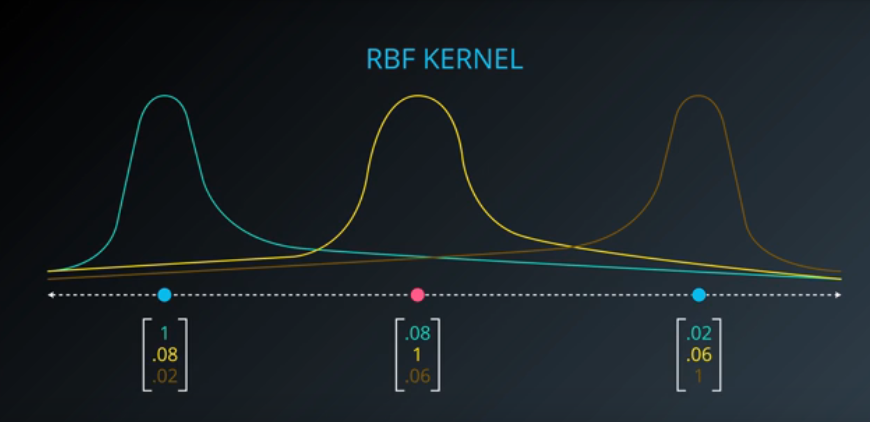
Therefore when we project back down to the line it will separate the blue and red points correctly:



For a more detailed example we could multiply our curves by larger numbers, white curve below is the sum of all the curves:



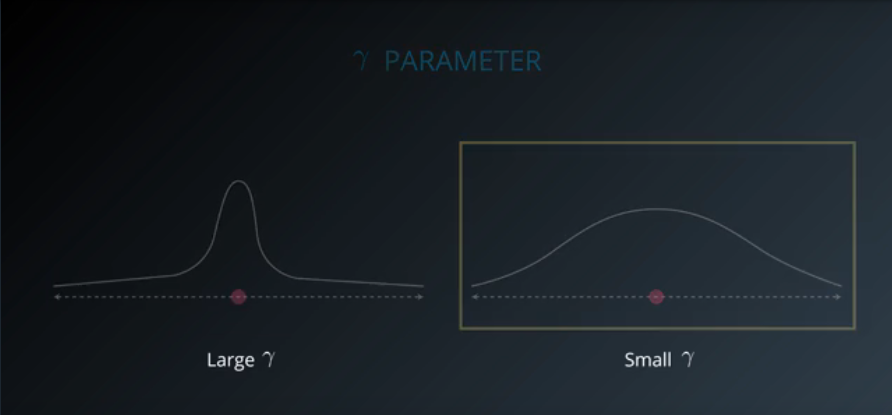
A line at the height -1 would cut this. We construct the weights used above using the following. We get the heights of our points at random curves using the following:



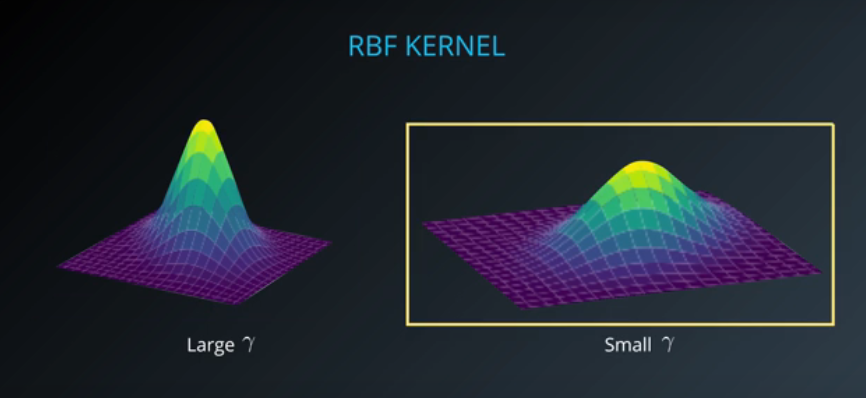
Then we plot these vectors in a 3 dimensional space:

………………………….

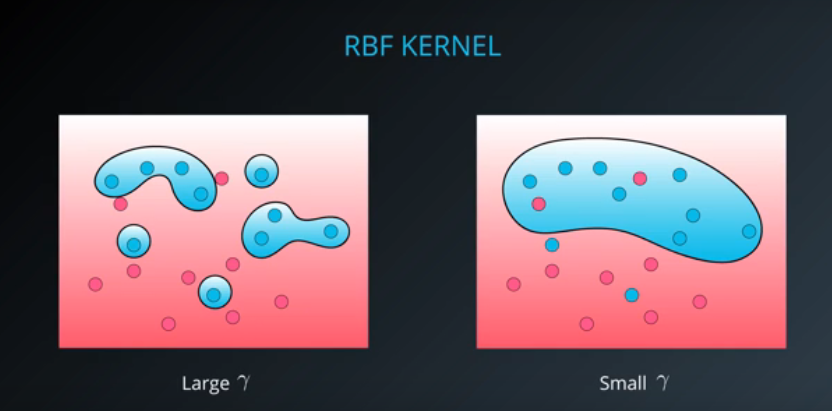
For RBF functions we use a hyperparameter called the gamma parameter. In a nutshell a large and small gamma would give us curves like below:



In higher dimensions this looks like:



This choice of gamma will matter a lot in the algorithm as it will guide how we group items:



**SVMs in sklearn**

Here are some of the tools we would use when using SVC’s in sklearn:

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

>>> model = SVC()

>>> model.fit(x\_values, y\_values)

In the example above, the model variable is a support vector machine model that has been fitted to the data x\_values and y\_values. Fitting the model means finding the best boundary that fits the training data. Let's make two predictions using the model's predict() function.

>>> print(model.predict([ [0.2, 0.8], [0.5, 0.4] ]))

[[ 0., 1.]]

The model returned an array of predictions, one prediction for each input array. The first input, [0.2, 0.8], got a prediction of 0.. The second input, [0.5, 0.4], got a prediction of 1..

**Hyperparameters**

When we define the model, we can specify the hyperparameters. As we've seen in this section, the most common ones are

* C: The C parameter.
* kernel: The kernel. The most common ones are 'linear', 'poly', and 'rbf'.
* degree: If the kernel is polynomial, this is the maximum degree of the monomials in the kernel.
* gamma : If the kernel is rbf, this is the gamma parameter.

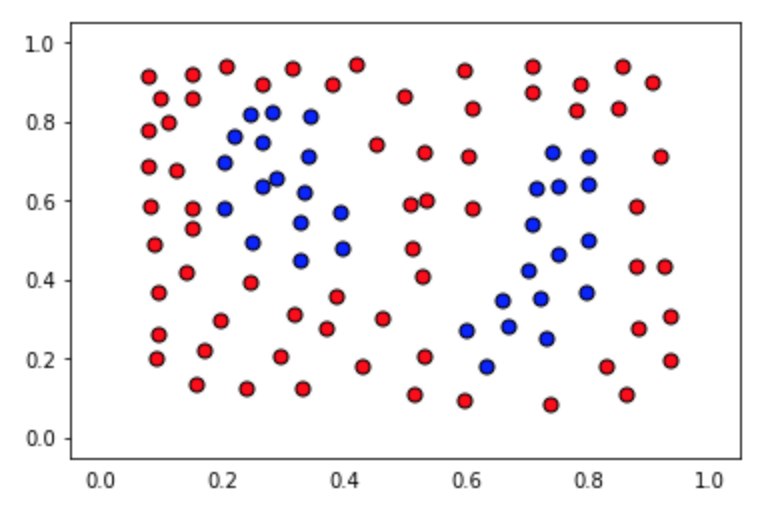
For example, here we define a model with a polynomial kernel of degree 4, and a C parameter of 0.1.

>>> model = SVC(kernel='poly', degree=4, C=0.1)

**QUESTION**

## Support Vector Machines Quiz

In this quiz, you'll be given with the following sample dataset, and your goal is to define a model that gives 100% accuracy on it.

[[](https://classroom.udacity.com/nanodegrees/nd009-ent/parts/e2907194-a763-44ca-9a5c-976bdc957119/modules/435676be-33e4-4ec8-877a-e4ee1f1904ca/lessons/e47029ee-a0de-405f-8b57-7cbfad02019c/concepts/d5a3f58b-29d7-4227-965f-edfe7315dd08)](https://classroom.udacity.com/nanodegrees/nd009-ent/parts/e2907194-a763-44ca-9a5c-976bdc957119/modules/435676be-33e4-4ec8-877a-e4ee1f1904ca/lessons/e47029ee-a0de-405f-8b57-7cbfad02019c/concepts/d5a3f58b-29d7-4227-965f-edfe7315dd08)

The data file can be found under the "data.csv" tab in the quiz below. It includes three columns, the first 2 comprising of the coordinates of the points, and the third one of the label.

The data will be loaded for you, and split into features X and labels y.

### You'll need to complete each of the following steps:

**1. Build a support vector machine model**

* Create a support vector machine classification model using scikit-learn's [SVC](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html) and assign it to the variablemodel.

**2. Fit the model to the data**

* If necessary, specify some of the hyperparameters. The goal is to obtain an accuracy of 100% in the dataset. Hint: Not every kernel will work well.

**3. Predict using the model**

* Predict the labels for the training set, and assign this list to the variable y\_pred.

**4. Calculate the accuracy of the model**

* For this, use the function sklearn function [accuracy\_score](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html" \t "_blank).

When you hit **Test Run**, you'll be able to see the boundary region of your model, which will help you tune the correct parameters, in case you need them.

**Note:** This quiz requires you to find an accuracy of 100% on the training set. Of course, this screams overfitting! If you pick very large values for your parameters, you will fit the training set very well, but it may not be the best model. Try to find the smallest possible parameters that do the job, which has less chance of overfitting, although this part won't be graded.

**SOLUTION**

# Import statements

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score

import pandas as pd

import numpy as np

# Read the data.

data = np.asarray(pd.read\_csv('data.csv', header=None))

# Assign the features to the variable X, and the labels to the variable y.

X = data[:,0:2]

y = data[:,2]

# TODO: Create the model and assign it to the variable model.

# Find the right parameters for this model to achieve 100% accuracy on the dataset.

model = SVC(kernel='rbf', gamma=27)

# TODO: Fit the model.

model.fit(X,y)

# TODO: Make predictions. Store them in the variable y\_pred.

y\_pred = model.predict(X)

# TODO: Calculate the accuracy and assign it to the variable acc.

acc = accuracy\_score(y, y\_pred)