Chapter 3 : part 4

**Kernel - SVM**

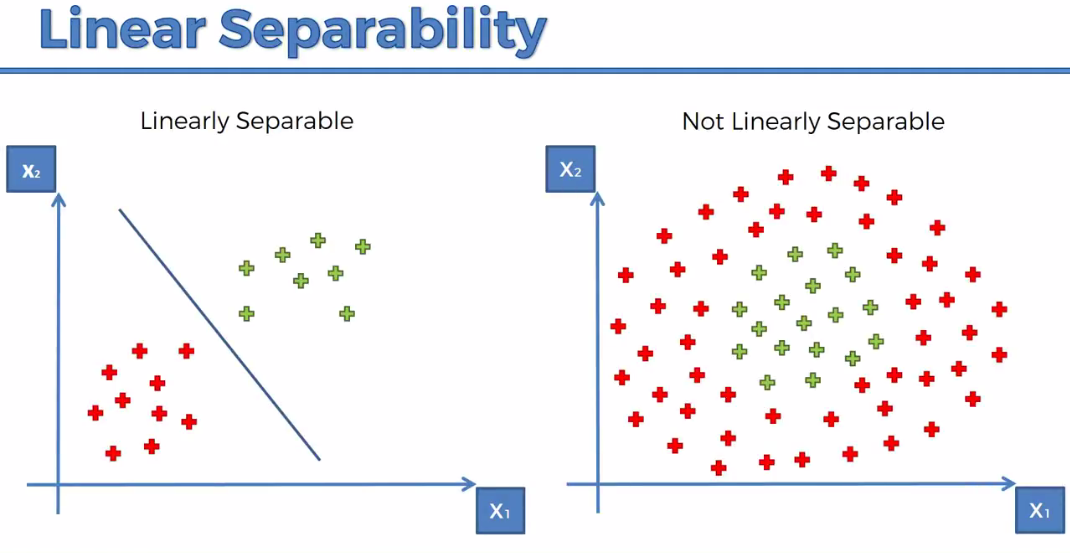
Kernel Support Vector Machine

**3.4.1 Kernel - SVM**

Recall in the SVM situation we had a set of observations which belong to ***different*** ***classes*** and the ***algorithm*** will find the ***decision*** ***boundary*** between them so that any *future* *observations* could be *identified* which *Class* they fall into.

|  |  |
| --- | --- |
|  |  |

* In first case we can see that there a decision boundary and the support vector machine algorithm tells us exactly how to find that boundary.
* Problem is the data is not always linearly separable: In the second case, we cannot separate the points in the same way that the *SVM* *algorithm* told us to. This happens because the data points are not *LINEARLY* *SEPARABLE*.
* Data Is Not Linearly Separable: Well this happens because in this case the *data is not linearly* *separable*. So here we've got the two examples side by side on the left the linearity separable data and on the right the nonlinear separable data.



* SVM algorithm helps us find that Decision Boundary or correctly place that Decision Boundary. But it does have an assumption. The *assumption* is that the *data must be separable*.
* But in that Non Linear Separable Case we can't even draw one single Decision Boundary or Linear Decision Boundary so therefore the Support Vector Machine algorithm ***just won't work by definition***.

**3.4.2 Tricks to deal with Non Linear Separable Data**

1. Mapping to a higher dimension: First of all we're going to explore a method called " Mapping to a higher dimension ". In this case we take our dataset and add an extra dimension into our space that we're dealing with and make our data a linearly separable with "some mapping".
2. Kernel trick: The kernel trick allows us to make our data separable without having to deal with multiple or higher dimensions.

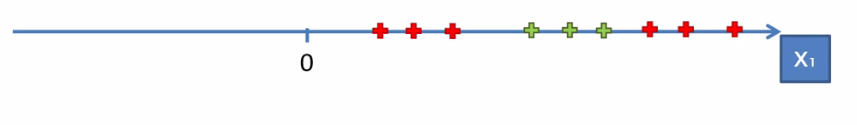
Lastly, we will talk about the different types of kernels that exist.

**3.4.3 Mapping to a higher dimension**

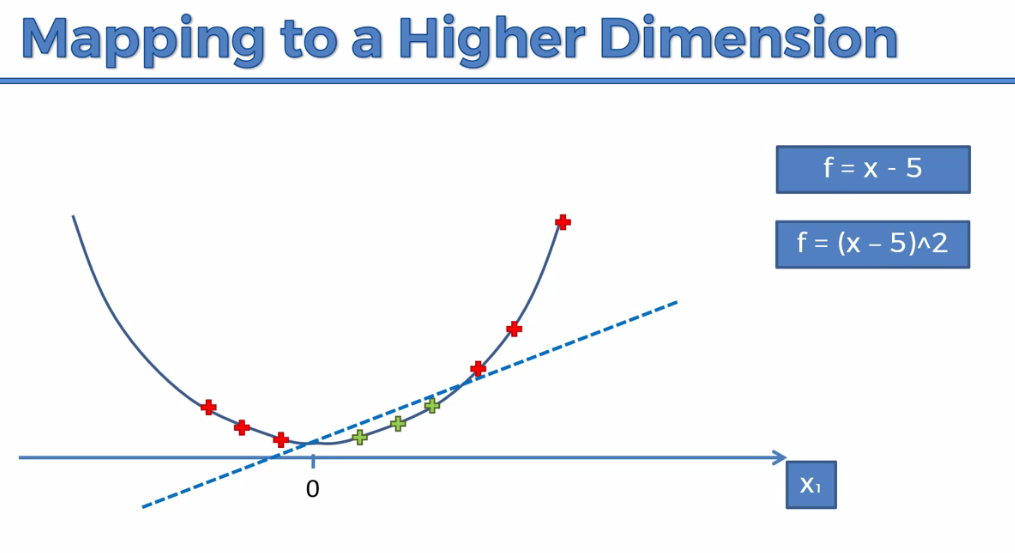
Use a Higher-Dimensional Space to make a separable data-set.

1. We can take our *nonlinearly separable data-set* map it to a higher dimension and get a *linearly separable data-set*.
2. Invoke the *SVM* *algorithm* build a *decision* *boundary* for a dataset and
3. Then *project* all of *that* *back* into our *original* *dimensions*.

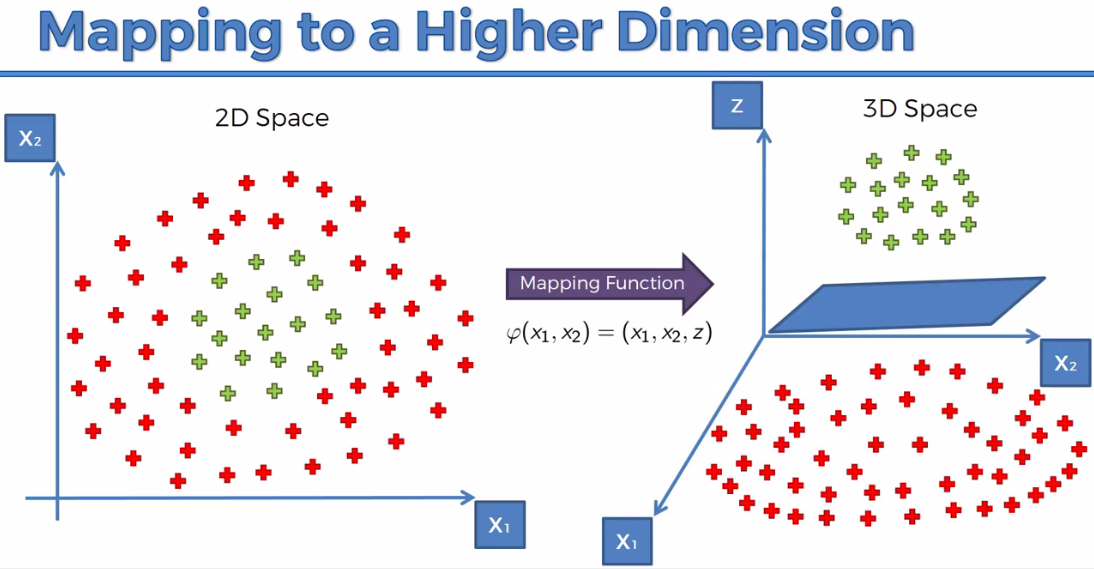
* 1D to 2D: First off we're going to look at a simplified example; we're going to look at a one dimensional dataset, so that we can kind of understand how it would work in multiple dimensions.



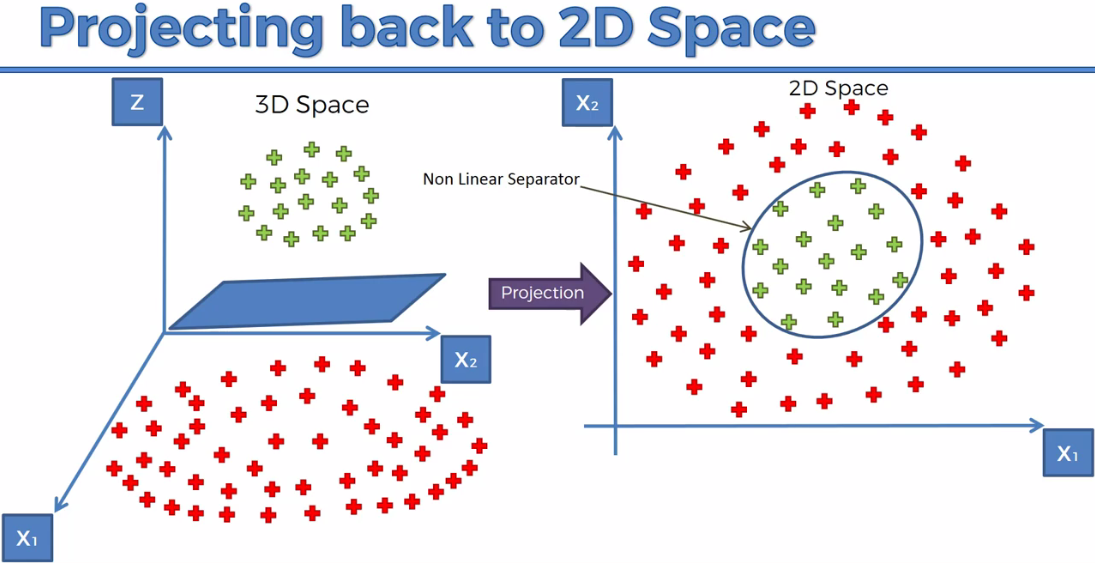
* Here we've got the dimension we've got some nine data points.
* Also, we can see our data is nonlinearity separable. In a single dimensional space a *linear* *separator* would *not* *be* *a line* or would be a *dot*. [In a two dimensional space, a linear separator is a line; in a three dimensional space as a hyper-plane; but in a one dimensional space it's a single dot].
* We are going to create this mapping function on the fly. So let's say that the first green dot is after the point 5. So our first step to build the mapping function it can be any mapping functions that you can build. Lets use the following function:
* We use 2nd-degree function (because it is Convex-curve) and we shift it to point .
* If you take you would get left-side red dots will go into negative. Right-side ones will stay positive. The next step would be to square all to get the curve.
* After getting the curve we will project all the observation points on the curve. Then the points can be linearly seperable.



* We could use a 3rd degree equation or trigonometric function. This depends on how we want o separate the observation points.
* The same thing applies to two dimensional space, moving into three dimensional space. You'd map it into three dimensional space and then somehow it would become a linearly separable dataset.
* In this space the linear separator is no longer a line, it's a hyperplane. So this *hyperplane* separates the two parts of our dataset in the way we want.



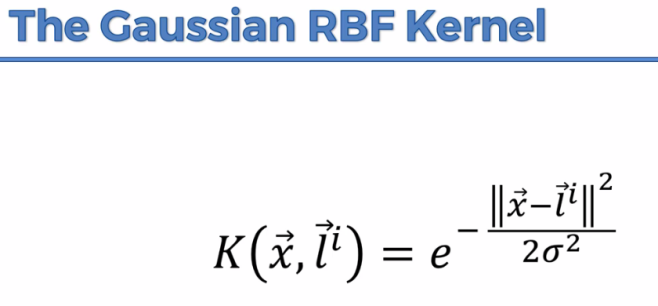
* Into this *higher dimension*, we then apply the *SVM* algorithm to get the separator-hyperplane. Once, we've got this result then we just projected back into our 2D space and we've got this circle (which is a non-linear separator in 2D).



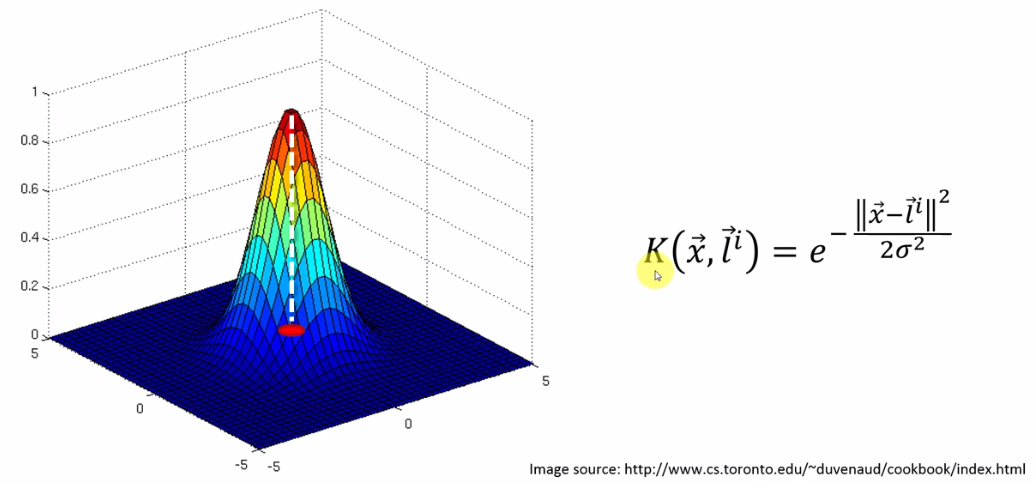
|  |  |
| --- | --- |
| * With this algorithm, the problem is that mapping to a ***higher dimensional space*** can be ***highly*** ***compute*** ***intensive*** so it might require a ***lot*** ***of*** ***computation*** a lot of processing power.   The larger your dataset the more of a problem this can cause and therefore this approach isn't the best because you can imagine like you have a dataset and then mapping it to a higher dimension performing all the calculations there and then coming back to your lower dimension for a computer that can cause a lot of delays it can cause a lot of like processing backlog and issue |  |

**3.4.4 The Kernel Trick**

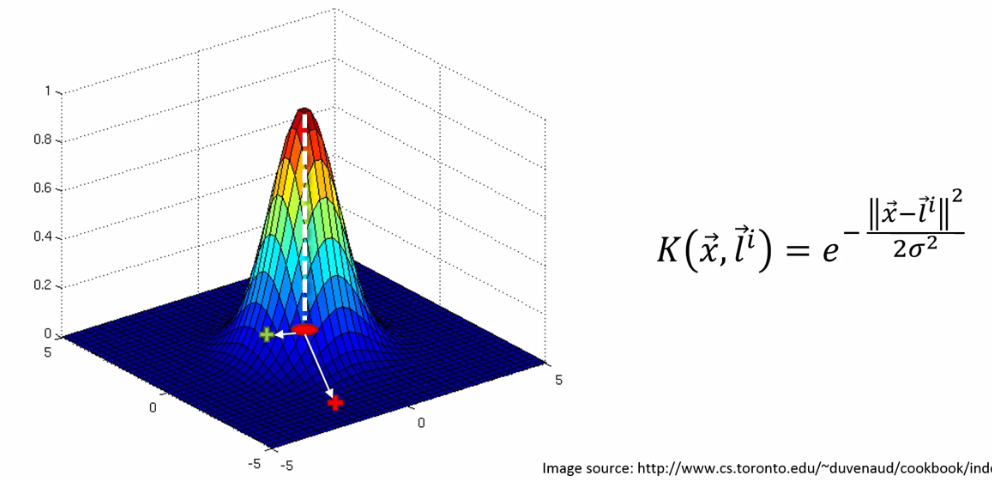
Here we got the ***Gaussian/ Radial Basis Function*** (***Gaussian*** ***RBF*** ***kernel***) kernel



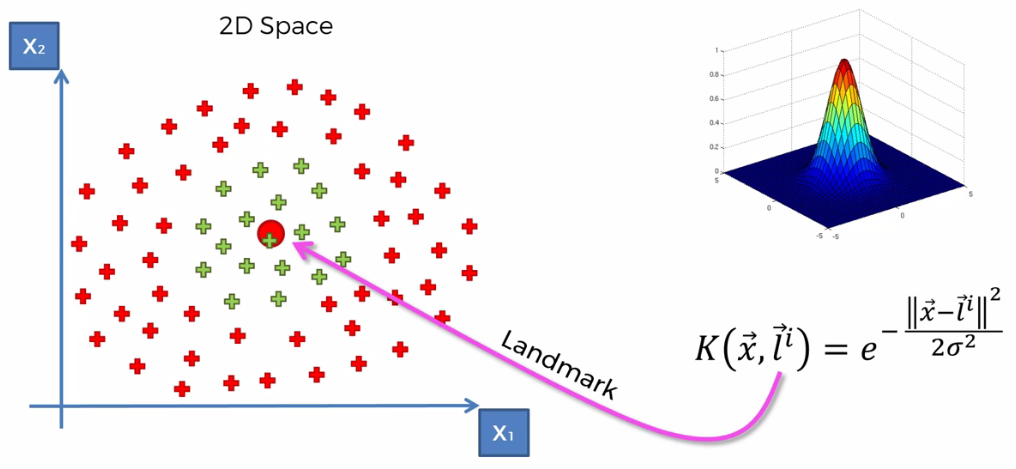
* So stands for kernel and it's the function applied to two vectors , .
* There is a some sort of point in our dataset and called Landmark.
* means there could be several landmark.
* The double vertical lines mean the distance between and the landmark .
* Sigma is act like radius (variance).



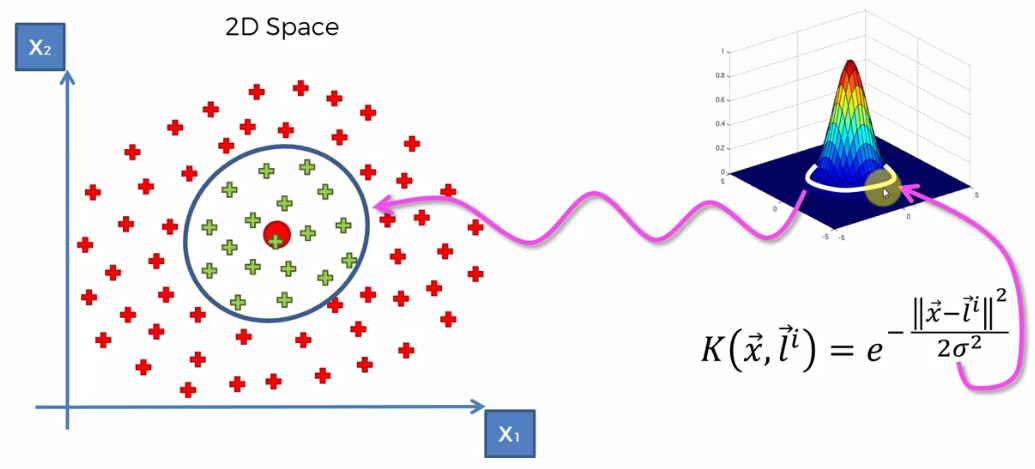
* In the above figure, at the *tip of this observation* is right in the middle, when we project it to xy plane, we get the position of the landmark. It is the point from which we're measuring the distance.

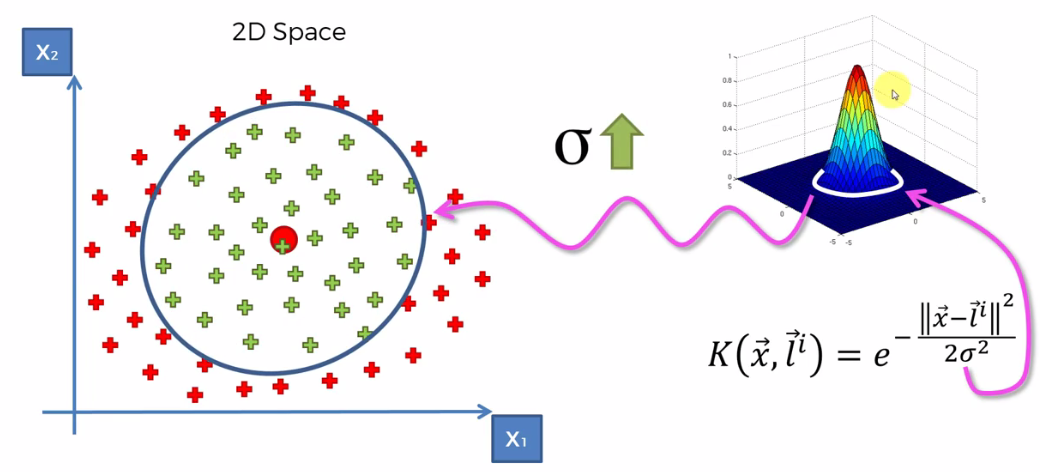


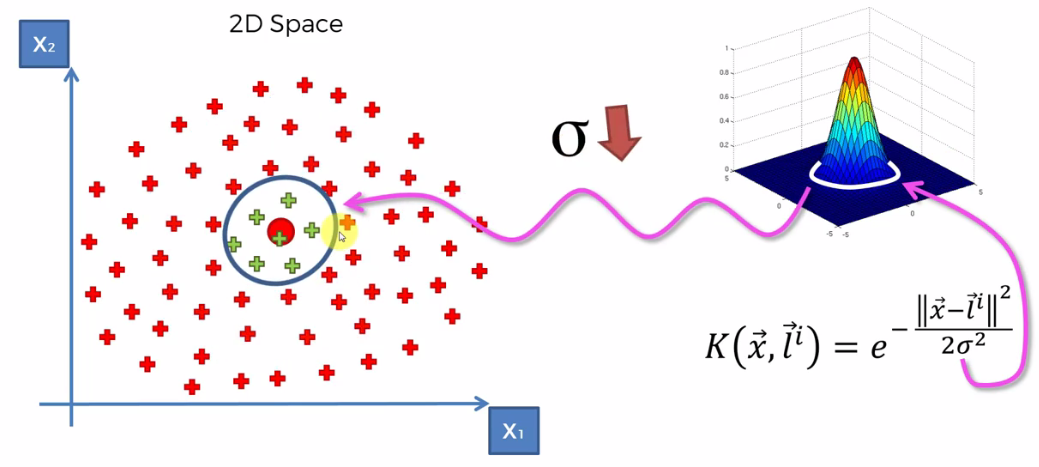
* Let there are two points somewhere on our plane. Green one is near the landmark, red one is far from the landmark.
* *Red* one quite far away from the landmark. So *distance* is *large*, hence is also large. Which makes .
* For the points which are closer to landmark (the green point), distance is small, hence is also small. Which makes . That's the whole methodology on how the machine learning algorithm to find an optimal placement for these landmarks.
* What's in 2D happening? We're going to use this kernel function to separate our data set to build that decision boundary. So let's have a look there is our two dimensional space.
* Let's go back to our plane. Now we're going to take the Landmark and put it somewhere in our Dataset.



* After landmark is placed, the algorithm is set to find the optimal circumference/non-linear boundary using **.** *When increases the boundary spreads, when decrease boundary shrinks.* Kernel function is actually projected here onto our dataset.
* All of the points that are within that circumference and have them like assign them a value of above zero interval. Anything outside the circumference basically all these red points they'll get a value of 0. In 3D space, points that are within that circumference are lie on the landmark, but the points outside the circumference are very close to zero lies in the blue-colored plane.
* Hence, based on this function we can separate the two classes the green from the red just if we pick the right Sigma .



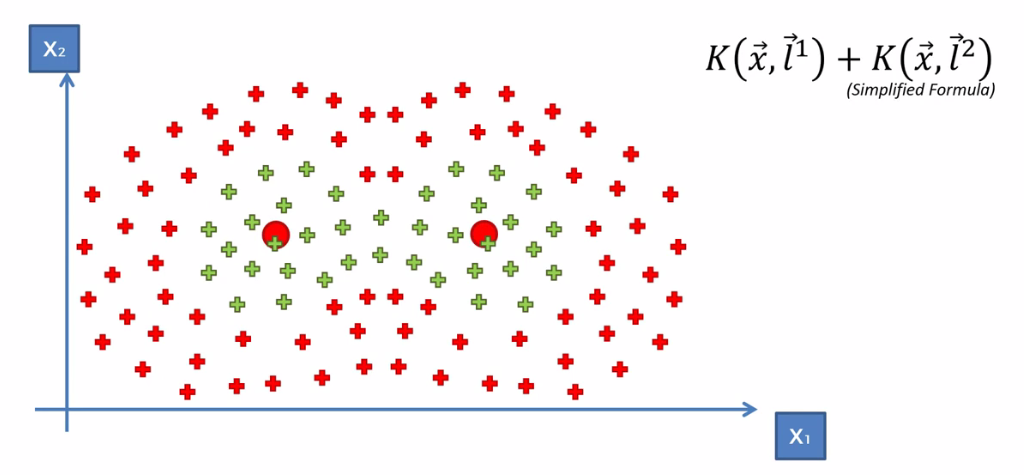




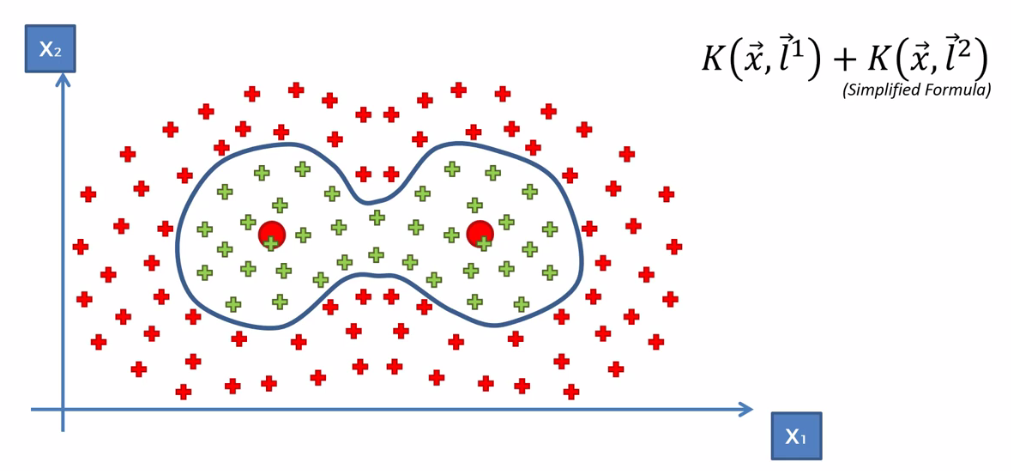
* So basically by finding the right Sigma you can set up the *correct kernel function* to assign *0* values to all of the points that you *don't want* in your classification and values *above zero* to the point that *you do want* in your classification. And that will allow you to *separate* the *two* *classes*. Allow you to classify each from one. That in essence is a kernel trick
* Hence, we have created a decision boundary actually going into a higher dimensional space without having to project us back to 2D space.
* Here *higher dimensional space is used* but we're still *doing the computations in the low dimensional space*.
* Yes we have this visual representation that involves a higher dimensional space but at the same time if you look at the part we were just calculating this formula in 2D.

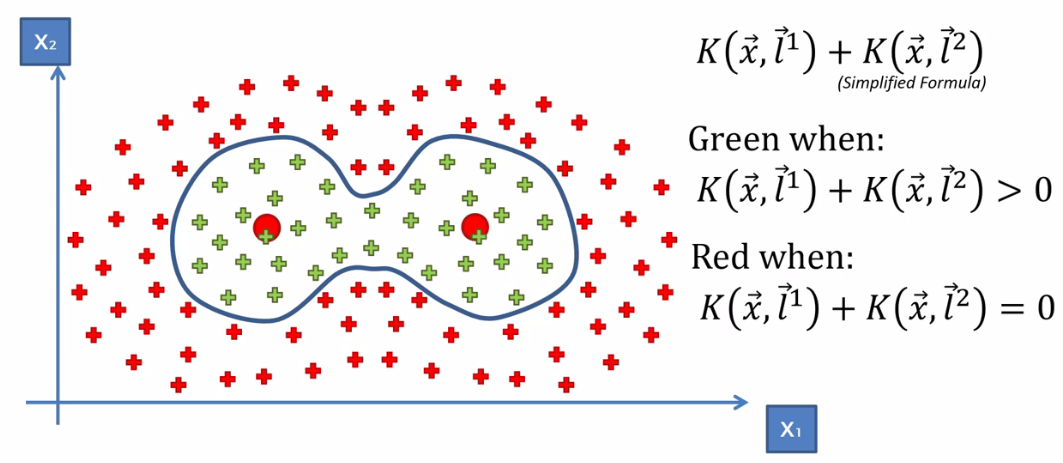
**3.4.5 The Kernel Trick with Multiple landmark**

Say all of a sudden you can't adjust your decision boundary and it's non-linear and moreover you find yourself being able to solve much harder much more complex problems like this for example.



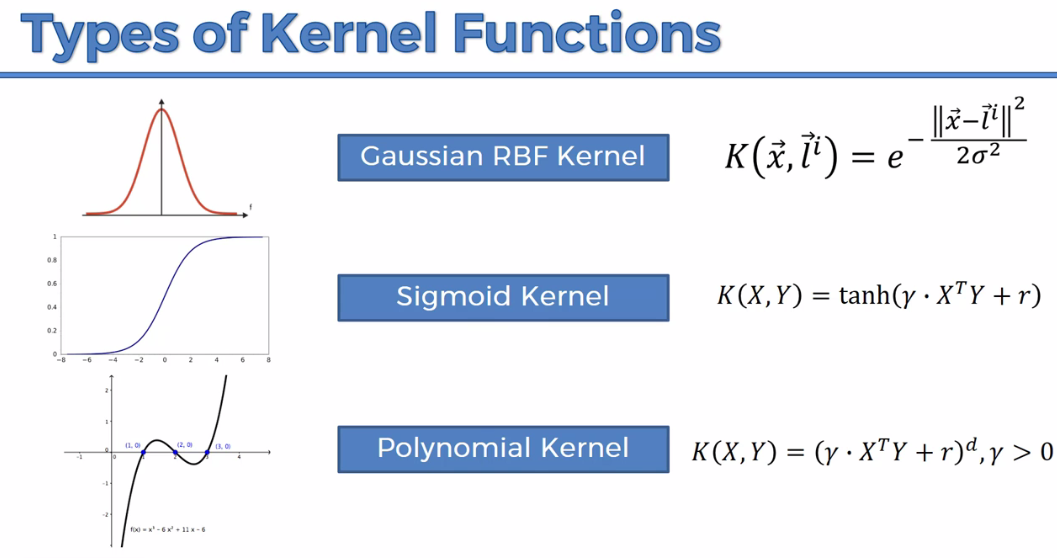
* So here is a very simplified formula:
* If you take *two kernel function* and you just *add* them up (in reality you need some coefficients). The points *far* from *one* *landmark* but *near* to *another* *landmark* then it has non-zero value. The points which are far from *both* *landmark* will get a 0 value. That's it.
* And the formula here would be the *point is assigned to the* ***green******class***, when this *equation is greater than zero* and the *point is assigned to* ***read******class*** when this *equation is equal to zero*.





**3.4.6 Different types of Kernel Functions**

you need to know about the kernel SVM is that the radial basis function which also called the Gaussian function is *not the only Kernel function* that is used in this method. So let's have a look at a couple.



|  |  |
| --- | --- |
| J:\My_images\feturespace_5.png | J:\My_images\feturespace_4.png |
| J:\My_images\feturespace_3.png | J:\My_images\feturespace_2.png |

Some more kernel fuunctions

|  |  |
| --- | --- |
| P:\Kernel\Machine Learning Kernel Functions_files\image-5_002.png | P:\Kernel\Machine Learning Kernel Functions_files\image-4.png |
| P:\Kernel\Machine Learning Kernel Functions_files\image-3.png | P:\Kernel\Machine Learning Kernel Functions_files\image-2.png |
| P:\Kernel\Machine Learning Kernel Functions_files\image-1.png | P:\Kernel\Machine Learning Kernel Functions_files\image-8_002.png |

**3.4.7 Python implementation of the kernel- SVM**

All code remain same as previous SVM-Algorithm. We just need to change the "Kernel": **kernel="rbf"**

#*Kernel="rbf" instead of kernel="linear"*

clsFier = **SVC**(kernel="rbf", random\_state=0)

Practiced version

#*Library*

**import** pandas **as** pd

**import** matplotlib**.**pyplot **as** pLt

**import** numpy **as** np

#*Data Extract*

dataSet = pd**.read\_csv**("Social\_Network\_Ads.csv")

X = dataSet**.**iloc[:, [2,3]]**.**values

y = dataSet**.**iloc[:, 4]**.**values

        #*Feature-Scaling after Data Split*

#*Data Split*

**from** sklearn**.**model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size= 0.25, random\_state = 0)

#*Feature-Scaling*

**from** sklearn**.**preprocessing **import** StandardScaler

st\_x= **StandardScaler**()

X\_train= st\_x**.fit\_transform**(X\_train)

X\_test= st\_x**.transform**(X\_test)

#*Fit train set to Kernel-SVM classifier*

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

#*Since data-points are non-seperable linearly, use "rbf" : Gaussian kernel, gives better result.*

#*Kernel="rbf" instead of kernel="linear"*

clsFier = **SVC**(kernel="rbf", random\_state=0)

clsFier**.fit**(X\_train, y\_train) #*fit the dataset*

#*Predict*

y\_prd = clsFier**.predict**(X\_test)

#*Making the confusion matrix use the function "confusion\_matrix"*

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

cm = **confusion\_matrix**(y\_true= y\_test, y\_pred= y\_prd)

#*parameters of cm: y\_true: Real values, y\_pred: Predicted value*

#*Visualising the Training set results*

**from** matplotlib**.**colors **import** ListedColormap

X\_set, y\_set = X\_train, y\_train

X1, X2 = np**.meshgrid**(np**.arange**(start = X\_set[:, 0]**.min**() - 1, stop = X\_set[:, 0]**.max**() + 1, step = 0.01),

                     np**.arange**(start = X\_set[:, 1]**.min**() - 1, stop = X\_set[:, 1]**.max**() + 1, step = 0.01))

pLt**.contourf**(X1, X2, clsFier**.predict**(np**.array**([X1**.ravel**(), X2**.ravel**()])**.**T)**.reshape**(X1**.**shape),

             alpha = 0.5, cmap = **ListedColormap**(('red', 'green')))

pLt**.xlim**(X1**.min**(), X1**.max**())

pLt**.ylim**(X2**.min**(), X2**.max**())

**for** i, j **in** **enumerate**(np**.unique**(y\_set)):

    pLt**.scatter**(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1],

                c = **ListedColormap**(('red', 'green'))(i), label = j)

pLt**.title**('Kernel-SVM (Training set)')

pLt**.xlabel**('Age')

pLt**.ylabel**('Estimated Salary')

pLt**.legend**()

pLt**.show**()

#*Visualising the Test set results*

**from** matplotlib**.**colors **import** ListedColormap

X\_set, y\_set = X\_test, y\_test

X1, X2 = np**.meshgrid**(np**.arange**(start = X\_set[:, 0]**.min**() - 1, stop = X\_set[:, 0]**.max**() + 1, step = 0.01),

                     np**.arange**(start = X\_set[:, 1]**.min**() - 1, stop = X\_set[:, 1]**.max**() + 1, step = 0.01))

pLt**.contourf**(X1, X2, clsFier**.predict**(np**.array**([X1**.ravel**(), X2**.ravel**()])**.**T)**.reshape**(X1**.**shape),

             alpha = 0.5, cmap = **ListedColormap**(('red', 'green')))

pLt**.xlim**(X1**.min**(), X1**.max**())

pLt**.ylim**(X2**.min**(), X2**.max**())

**for** i, j **in** **enumerate**(np**.unique**(y\_set)):

    pLt**.scatter**(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1],

                c = **ListedColormap**(('red', 'green'))(i), label = j)

pLt**.title**('Kernel-SVM (Test set)')

pLt**.xlabel**('Age')

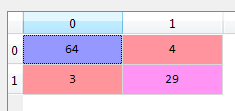
pLt**.ylabel**('Estimated Salary')

pLt**.legend**()

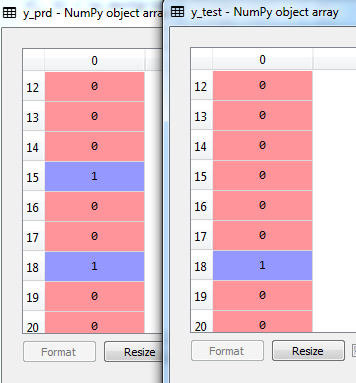
pLt**.show**()

#*python prctc\_\_kernel\_SVM.py*

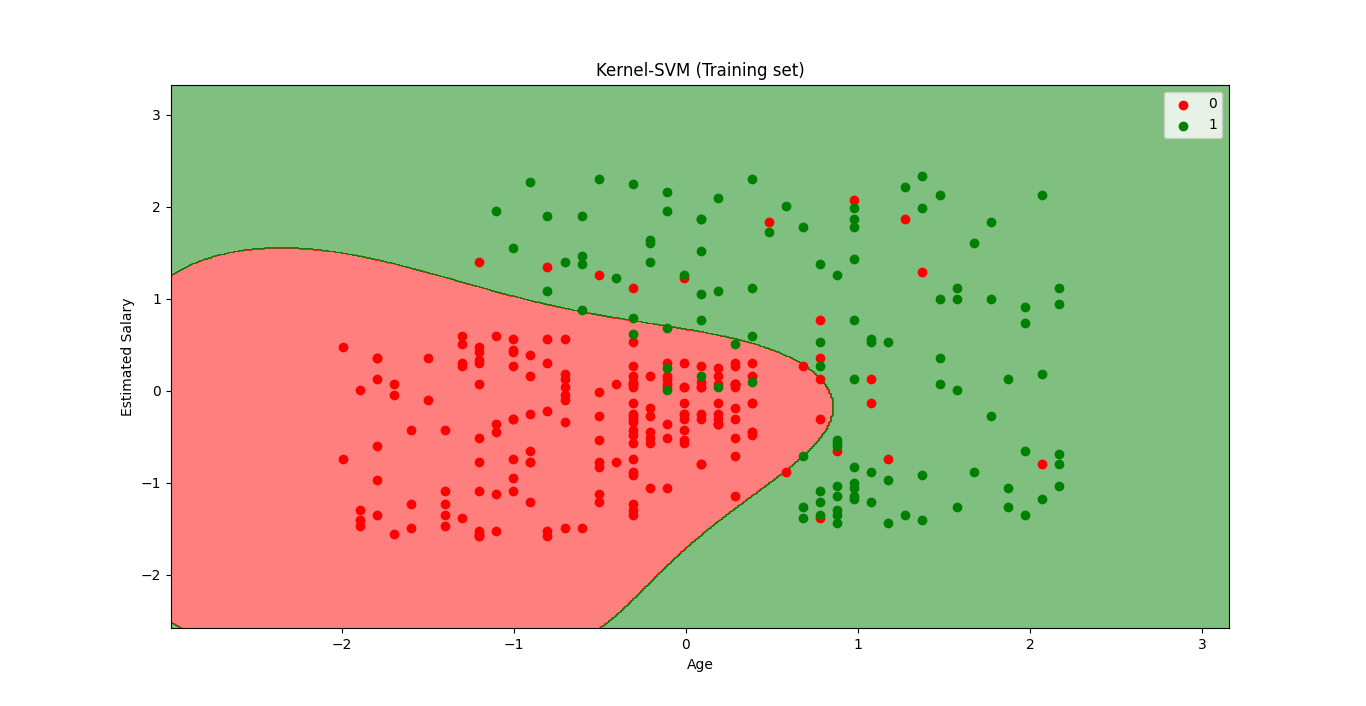
* Confusion matrix:



* Prediction:



Train data Visualization



Test data Visualization

