INTRODUCTION TO SUPPORT VECTOR MACHINES

http://www.svms.org/tutorials/Berwick2003.pdf

http://blog.aylien.com/support-vector-machines-for-dummies-a-simple/

https://www.svm-tutorial.com/2014/11/svm-understanding-math-part-1/

https://data-flair.training/blogs/svm-support-vector-machine-tutorial/

https://www.quantstart.com/articles/Support-Vector-Machines-A-Guide-for-Beginners

https://eight2late.wordpress.com/2017/02/07/a-gentle-introduction-to-support-vector-machines-using-r/

SVMs: A New Generation of Learning Algorithms

- Pre 1980:
  - Almost all learning methods learned linear decision surfaces.
  - Linear learning methods have nice theoretical properties
- 1980's
  - Decision trees and NNs allowed efficient learning of nonlinear decision surfaces
  - Little theoretical basis and all suffer from local minima
- 1990's
  - Efficient learning algorithms for non-linear functions based on computational learning theory developed
  - Nice theoretical properties.

Key Ideas

- Two independent developments within last decade
  - Computational learning theory
  - New efficient separability of non-linear functions that use “kernel functions”
- The resulting learning algorithm is an optimization algorithm rather than a greedy search.

Statistical Learning Theory

- Systems can be mathematically described as a system that
  - Receives data (observations) as input and
  - Outputs a function that can be used to predict some features of future data.
- Statistical learning theory models this as a function estimation problem
- Generalization Performance (accuracy in labeling test data) is measured

Motivation for Support Vector Machines

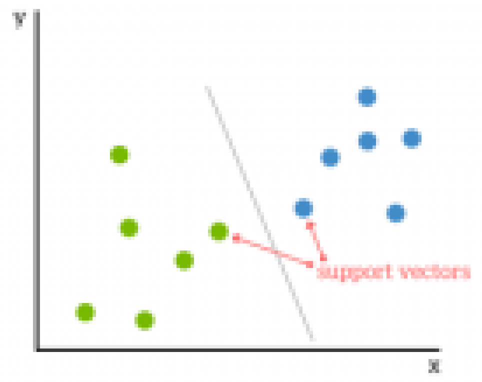
- The problem to be solved is one of the **supervised binary classification**. That is, we wish to categorize new unseen objects into two separate groups based on their properties and a set of known examples, which are already categorized.
- A good example of such a system is classifying a set of new *documents* into positive or negative sentiment groups, based on other documents which have already been classified as positive or negative.
- Similarly, we could classify new emails into spam or non-spam, based on a large corpus of documents that have already been marked as spam or non-spam by humans. SVMs are highly applicable to such situations.

Motivation for Support Vector Machines

- A Support Vector Machine models the situation by creating a *feature space*, which is a finite-dimensional vector space, each dimension of which represents a "feature" of a particular object. In the context of spam or document classification, each "feature" is the prevalence or importance of a particular word.
- The **goal of the SVM** is to train a model that assigns new unseen objects into a particular category.
- It achieves this by creating a linear partition of the feature space into two categories.
- Based on the features in the new unseen objects (e.g. documents/emails), it places an object "above" or "below" the separation plane, leading to a categorization (e.g. spam or non-spam). This makes it an example of a non-probabilistic linear classifier. It is non-probabilistic, because the features in the new objects fully determine its location in feature space and there is no stochastic element involved.

OBJECTIVES

- **Support vector machines (SVM)** are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis.
- It is a **machine learning** approach.
- They analyze the large amount of data to identify patterns from them.
- SVMs are based on the idea of finding a hyperplane that best divides a dataset into two classes, as shown in the image below.



Support Vectors

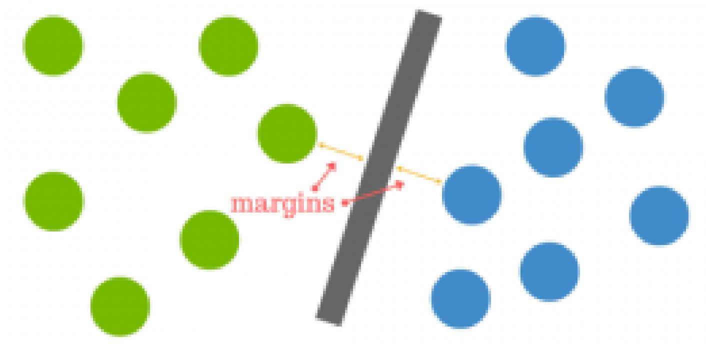
- Support Vectors are simply the co-ordinates of individual observation. Support Vector Machine is a frontier which best segregates the two classes (hyper-plane/ line).
- Support vectors are the data points that lie closest to the decision surface (or hyperplane)
- They are the data points most difficult to classify
- They have direct bearing on the optimum location of the decision surface
- We can show that the optimal hyperplane stems from the function class with the lowest “capacity” (VC dimension).
- Support vectors are the data points nearest to the hyperplane, the points of a data set that, if removed, would alter the position of the dividing hyperplane. Because of this, they can be considered the critical elements of a data set.

What is a hyperplane?

- As a simple example, for a classification task with only two features, you can think of a hyperplane as a line that linearly separates and classifies a set of data.
- Intuitively, the further from the hyperplane our data points lie, the more confident we are that they have been correctly classified. We therefore want our data points to be as far away from the hyperplane as possible, while still being on the correct side of it.
- So when new testing data are added, whatever side of the hyperplane it lands will decide the class that we assign to it.

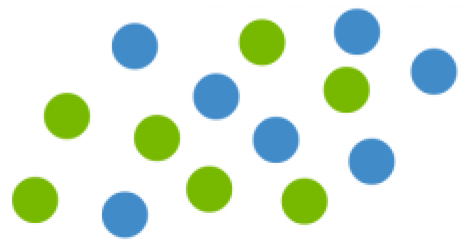
How do we find the right hyperplane?

- How do we best segregate the two classes within the data?
- The distance between the hyperplane and the nearest data point from either set is known as the **margin**. The goal is to choose a hyperplane with the greatest possible margin between the hyperplane and any point within the training set, giving a greater chance of new data being classified correctly. **There will never be any data point inside the margin.**

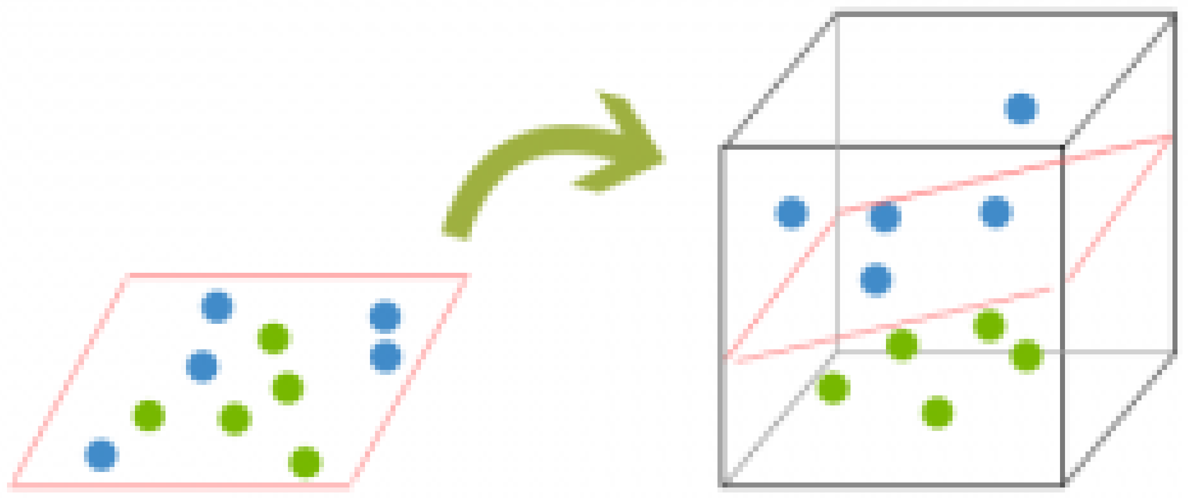


But what happens when there is no clear hyperplane?

- Data are rarely ever as clean as our simple example above. A dataset will often look more like the jumbled balls below which represent a linearly non separable dataset.



- In order to classify a dataset like the one above it's necessary to move away from a 2d view of the data to a 3d view. Explaining this is easiest with another simplified example. Imagine that our two sets of colored balls above are sitting on a sheet and this sheet is lifted suddenly, launching the balls into the air. While the balls are up in the air, you use the sheet to separate them. This 'lifting' of the balls represents the mapping of data into a higher dimension. This is known as **kernelling**.



Because we are now in three dimensions, our hyperplane can no longer be a line. It must now be a plane as shown in the example above. The idea is that the data will continue to be mapped into higher and higher dimensions until a hyperplane can be formed to segregate it.

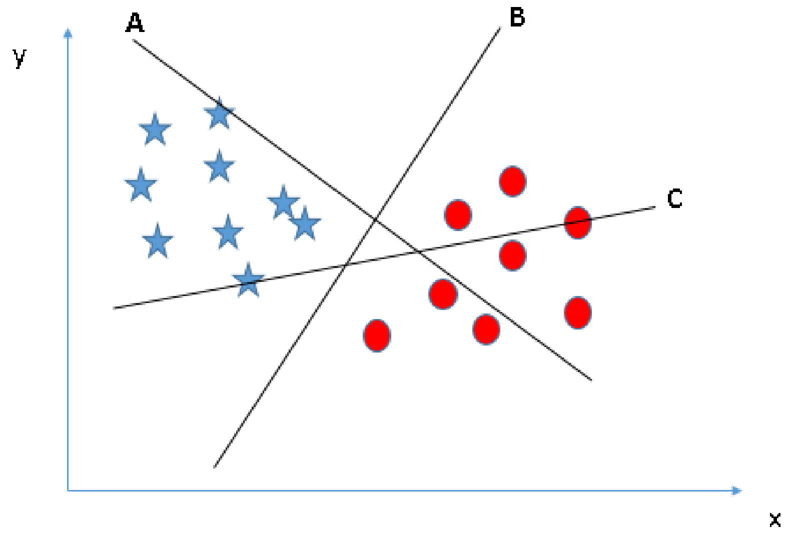
How does it work? How can we identify the right hyper-plane?

- You need to remember a thumb rule to identify the right hyper-plane:

“Select the hyper-plane which segregates the two classes better”.

Identify the right hyperplane (Scenario-1):

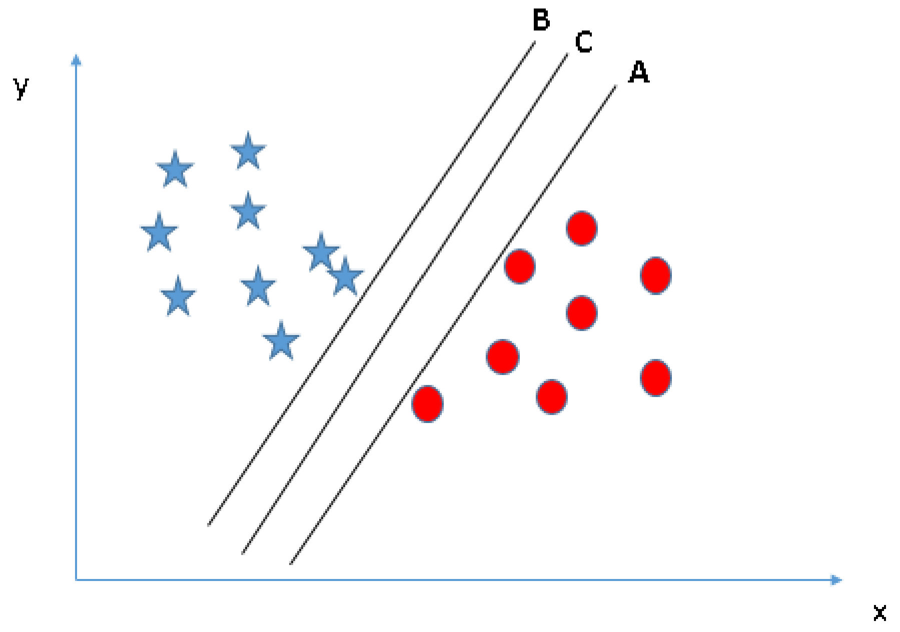
- Here, we have three hyperplanes (A, B and C). Now, identify the right hyperplane to classify star and circle.



- Hyperplane “B” has excellently performed this job.

Identify the right hyperplane (Scenario-2):

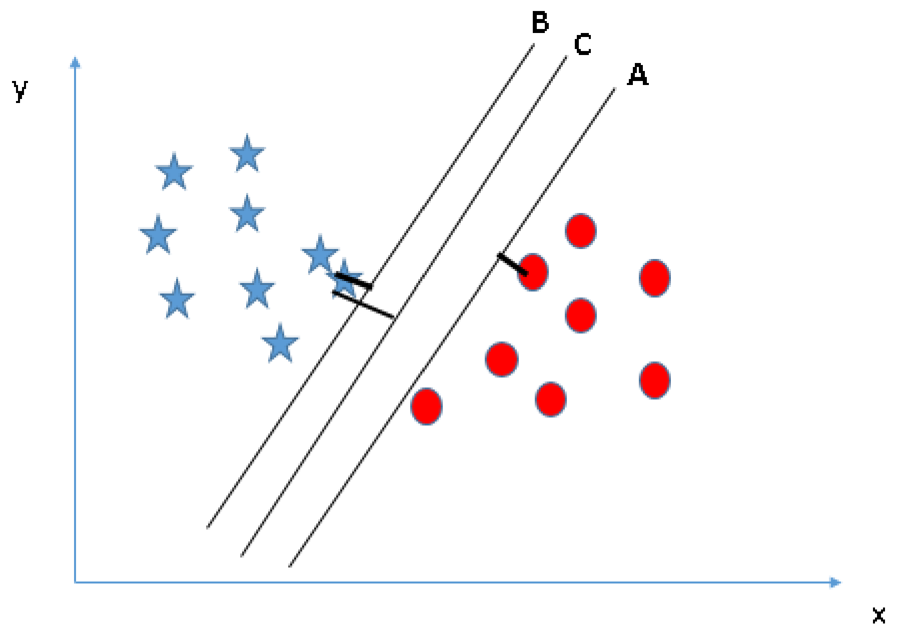
- Here, we have three hyperplanes (A, B and C) and all are segregating the classes well. Now, how can we identify the right hyperplane?



Here, maximizing the distances between nearest data point (either class) and hyperplane will help us to decide the right hyperplane.

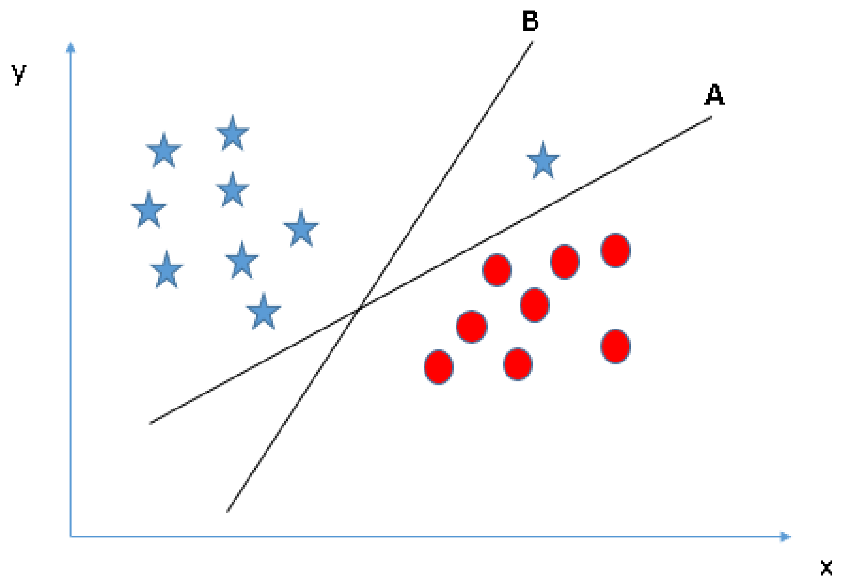
Scenario-2

This distance is called as **Margin**. Let’s look at the below snapshot:



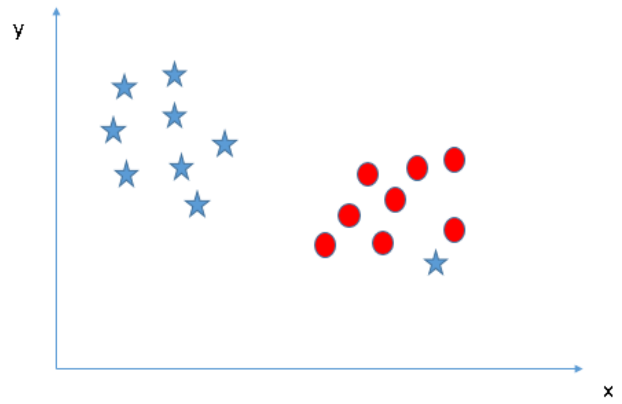
We can see that the margin for hyperplane C is high as compared to both A and B. Hence, we name the right hyperplane as C. Another lightning reason for selecting the hyperplane with higher margin is robustness. If we select a hyperplane having low margin then there is high chance of missclassification.

Identify the right hyperplane (Scenario-3)

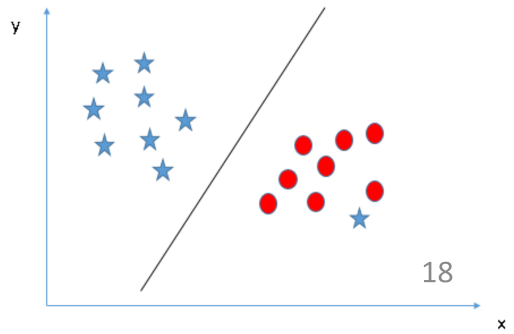


- Some of you may have selected the hyper-plane **B** as it has higher margin compared to **A.** But, here is the catch, SVM selects the hyperplane which classifies the classes accurately prior to maximizing margin. Here, hyperplane B has a classification error and A has classified all correctly. Therefore, the right hyperplane is **A.**

Can we classify two classes (Scenario-4)?

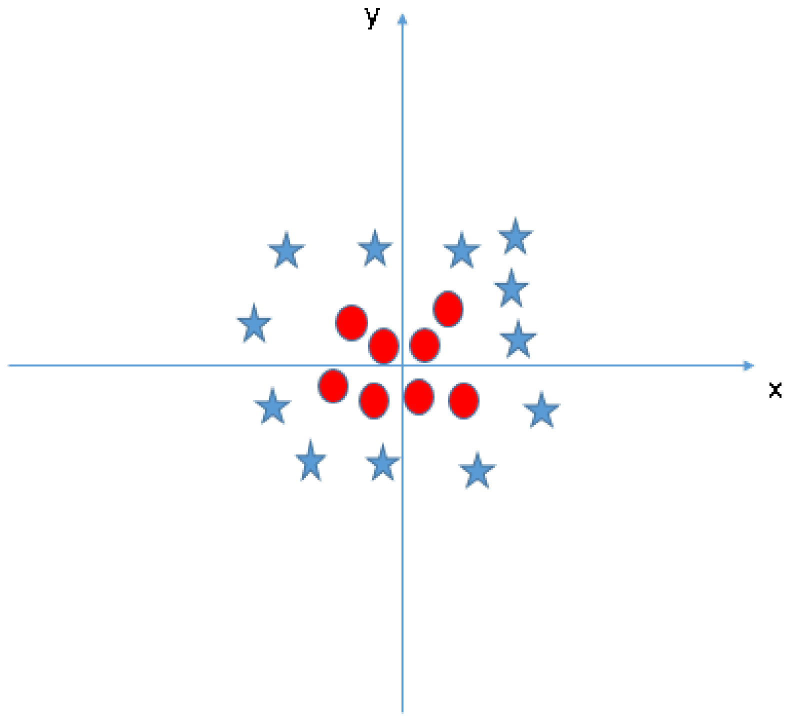


- We are unable to segregate the two classes using a straight line, as one of star lies in the territory of other (circle) class as an outlier.
- One star at other end is like an outlier for star class. SVM has a feature to ignore outliers and find the hyperplane that has maximum margin. Hence, we can say, SVM is robust to outliers.



Find the hyperplane to segregate to classes (Scenario-5)

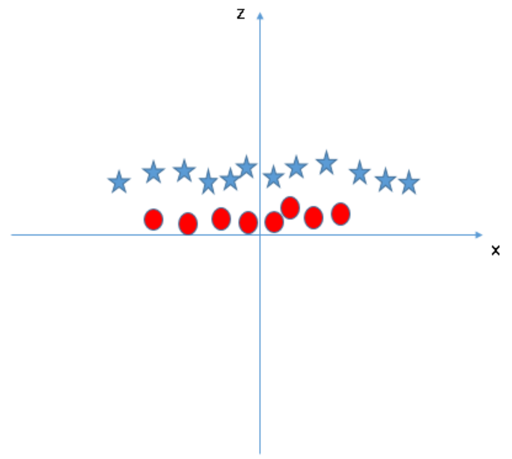
- In the scenario below, we can’t have linear hyperplane between the two classes, so how does SVM classify these two classes? Till now, we have only looked at the linear hyperplane.



SVM can solve this problem. It solves this problem by introducing additional feature. Here, we will add a new feature $z=x^2+y^2$.

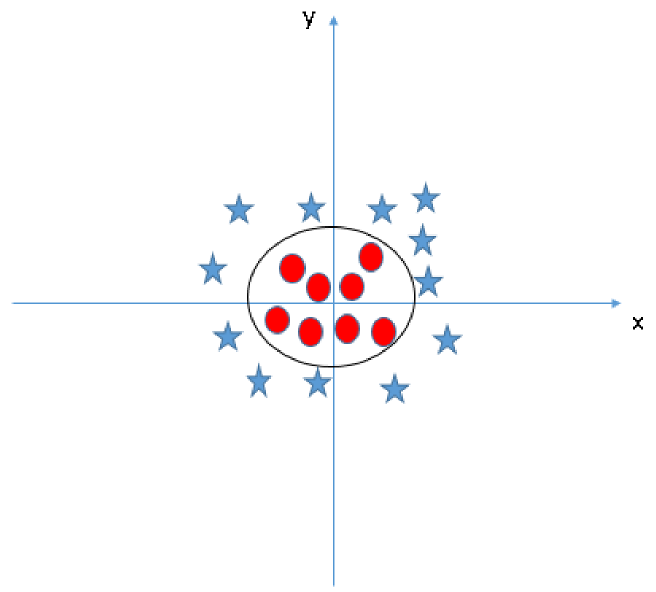
Scenario-5

- Now, let’s plot the data points on axis x and z:



- 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.

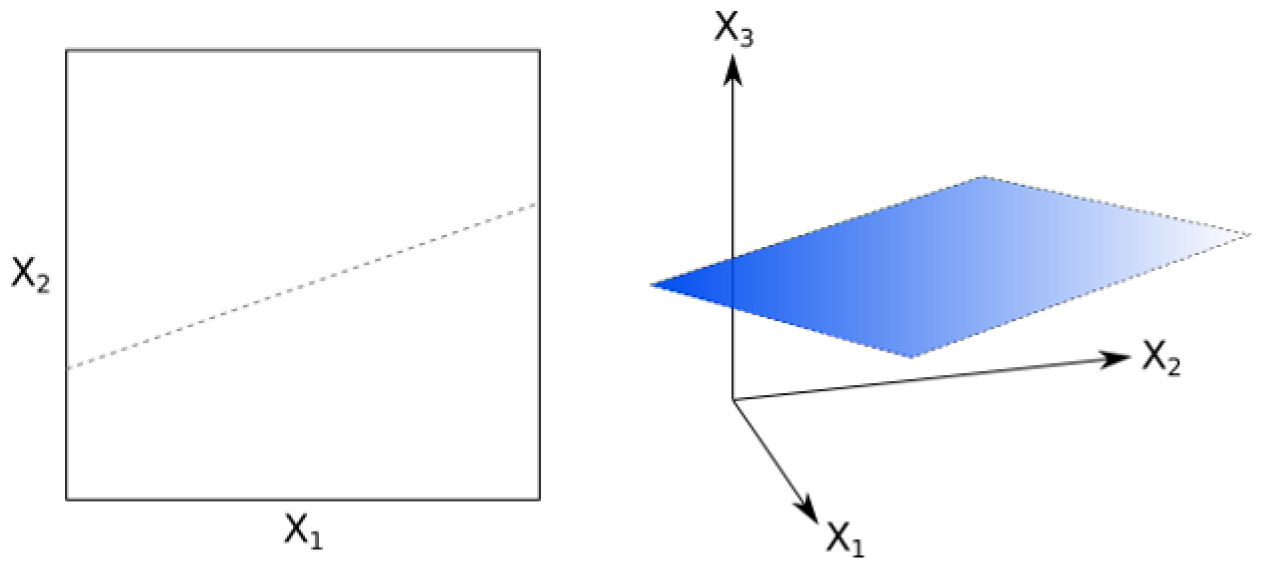
- In SVM, it is easy to have a linear hyperplane between these two classes. But, another burning question which arises is, should we need to add this feature manually to have a hyperplane. No, SVM has a technique called the **kernel trick**. These are functions which takes low dimensional input space 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 hyperplane in original input space it looks like a circle:



Linear Separating Hyperplanes

- The linear separating hyperplane is the key geometric entity that is at the heart of the SVM. Informally, if we have a high-dimensional feature space, then the linear hyperplane is an object one dimension lower than this space that divides the feature space into two regions.
- This linear separating plane need not pass through the origin of our feature space, i.e. it does not need to include the zero vector as an entity within the plane. Such hyperplanes are known as **affine**.
- If we consider a real-valued p-dimensional feature space, known mathematically as $\Re^p$, then our linear separating hyperplane is an affine $p-1$ dimensional space embedded within it.

- For the case of p=2 this hyperplane is simply a one-dimensional straight line, which lives in the larger two-dimensional plane, whereas for p=3 the hyperplane is a two-dimensional plane that lives in the larger three-dimensional feature space.



- If we consider an element of our p-dimensional feature space, i.e. $\vec{x}$=($x_1, \dots, x_p$)$\in\Re^p$, then we can mathematically define an affine hyperplane by the following equation:

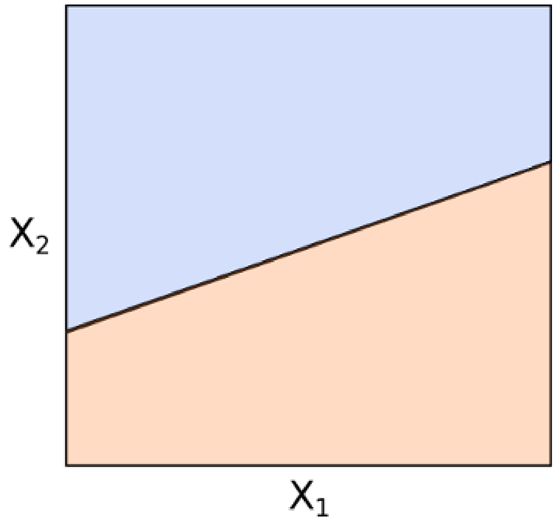
$$b_0 + b_1 x_1 + \cdots + b_p x_p = 0$$

$b_0 \neq 0$ gives us an affine plane (i.e. it does not pass through the origin). We can use a more concise notation for this equation by introducing the summation sign:

$$b_0 + \sum_{i=1}^{p} b_i x_i = 0$$

$$\vec{b}.\vec{x} + b_0 = 0$$

- If an element $\vec{x} \in \Re^p$ satisfies this relation then it lives on the p−1-dimensional hyperplane. This hyperplane splits the p-dimensional feature space into two classification regions.



- Elements $\vec{x}$ above the plane satisfy:

$$\vec{b}.\vec{x} + b_0 > 0$$

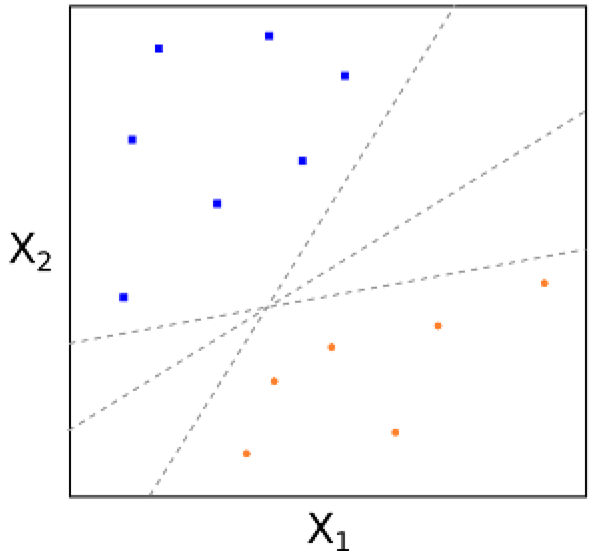
While those below it satisfy:

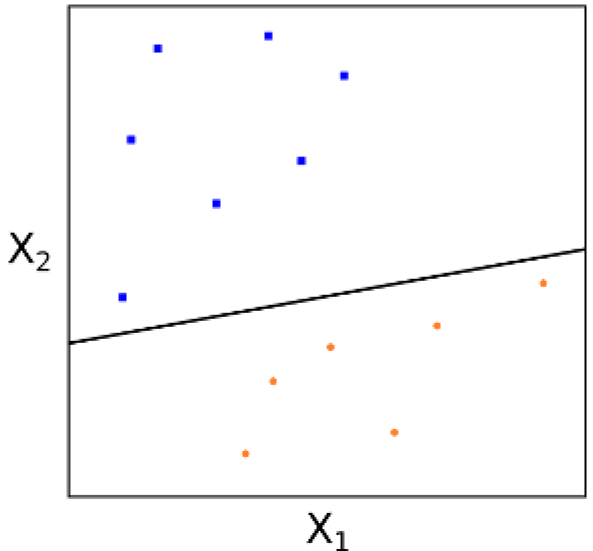
$$\vec{b}.\vec{x} + b_0 < 0$$

Classification

- Continuing with our example of email spam filtering, we can think of our classification problem (say) as being provided with a thousand emails (*n*=1000), each of which is marked spam (+1) or non-spam (−1). In addition, each email has an associated set of keywords (i.e. separating the words on spacing) that provide features. Hence if we take the set of all possible keywords from all of the emails (and remove duplicates), we will be left with p keywords in total.
- If we translate this into a mathematical problem, the standard setup for a supervised classification procedure is to consider a set of *n* training observations, $\vec{x}_i$, each of which is a p-dimensional vector of features. Each training observation has an associated class label, $y_i \in \{-1,1\}$. Hence we can think of *n* pairs of training observations $(\vec{x}_i, y_i)$ representing the features and class labels (keyword lists and spam/non-spam).

- In addition to the training observations we can provide test observations, $\vec{x}^*$=($\vec{x}_1^*$,..., $\vec{x}_p^*$) that are later used to test the performance of the classifiers. In our spam example, these test observations would be new emails that have not yet been seen.
- Our goal is to develop a classifier based on provided training observations that will correctly classify subsequent test observations using only their feature values. This translates into being able to classify an email as spam or non-spam solely based on the keywords contained within it.
- We will initially suppose that it is possible, via a means yet to be determined, to construct a hyperplane that separates training data *perfectly* according to their class labels.





- This translates into a mathematical separating property of:

$$\vec{b}.\vec{x} + b_0 > 0 \text{ if } y_i = 1$$
$$\vec{b}.\vec{x} + b_0 < 0 \text{ if } y_i = -1$$

- This basically states that if each training observation is above or below the separating hyperplane, according to the geometric equation which defines the plane, then its associated class label will be +1 or −1. Thus we have developed a simple classification process. We assign a test observation to a class depending upon which side of the hyperplane it is located on.

- This can be formalized by considering the following function $f(\vec{x}^*)$, with a test observation $\vec{x}^*$=($\vec{x}_1^*$,..., $\vec{x}_p^*$)

$$f(\vec{x}^*) = \vec{b}.\vec{x}^* + b_0$$

- If $f(\vec{x}^*) > 0$, then $y^*= 1$, whereas $f(\vec{x}^*) < 0$, then $y^*= -1$.
- However, this tells us nothing about how we go about finding the b_j components of $\vec{b}$, as well as b_0, which are crucial in helping us determine the equation of the hyperplane separating the two regions.

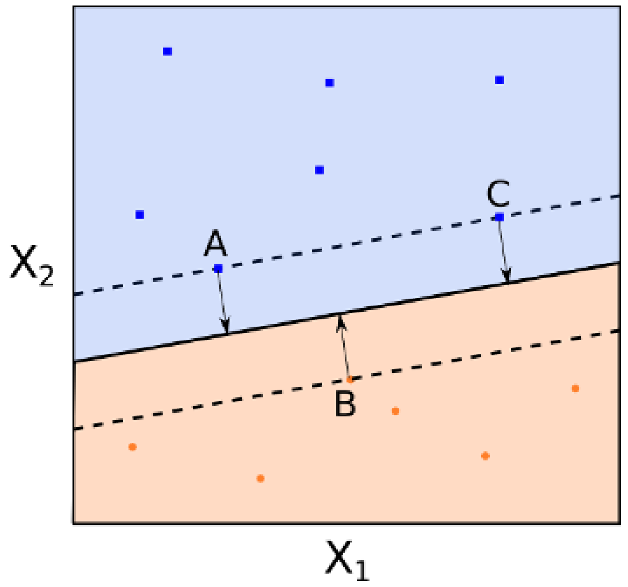
Deriving the Classifier

- Separating hyperplanes are not unique, since it is possible to slightly translate or rotate such a plane without touching any training observations.
- So, not only do we need to know *how* to construct such a plane, but we also need to determine the most *optimal*. This motivates the concept of the **maximal margin hyperplane** (MMH), which is the separating hyperplane that is farthest from any training observations, and is thus "optimal".

- How do we find the maximal margin hyperplane?
- Firstly, we compute the perpendicular distance from each training observation $\vec{x}_i$ for a given separating hyperplane.
- The smallest perpendicular distance to a training observation from the hyperplane is known as the **margin**.
- The MMH is the separating hyperplane where the margin is the largest.
- This guarantees that it is the farthest minimum distance to a training observation.

- The classification procedure is then just simply a case of determining which side a test observation falls on. This can be carried out using the above formula for $f(\vec{x}^*)$.
- Such a classifier is known as a **maximal margin classifier** (MMC).
- Note however that finding the particular values that lead to the MMH is purely based on the training observations. That is, we still need to be aware of how the MMC performs on the test observations.
- We are implicitly making the assumption that a large margin in the training observations will provide a large margin on the test observations, but this may not be the case.
- As always, we must be careful to avoid *overfitting* when the number of feature dimensions is large. Overfitting here means that the MMH is a very good fit for the *training data* but can perform quite poorly when exposed to *testing data*.

- Our goal now becomes finding an algorithm that can produce the b_j values, which will fix the geometry of the hyperplane and hence allow determination of $f(\vec{x}^*)$ for any test observation.



- We can see that the MMH is the mid-line of the widest "block" that we can insert between the two classes such that they are perfectly separated.

- One of the key features of the MMC (and subsequently SVC and SVM) is that the location of the MMH only depends on the support vectors, which are the training observations that lie directly on the margin (but not hyperplane) boundary (see points A, B and C in the figure). This means that the location of the MMH is NOT dependent upon any other training observations.

- Thus it can be immediately seen that a potential drawback of the MMC is that its MMH (and thus its classification performance) can be extremely sensitive to the support vector locations. However, it is also partially this feature that makes the SVM an attractive computational tool, as we only need to store the support vectors in memory once it has been "trained" (i.e. the b_j values are fixed).

Constructing the Maximal Margin Classifier

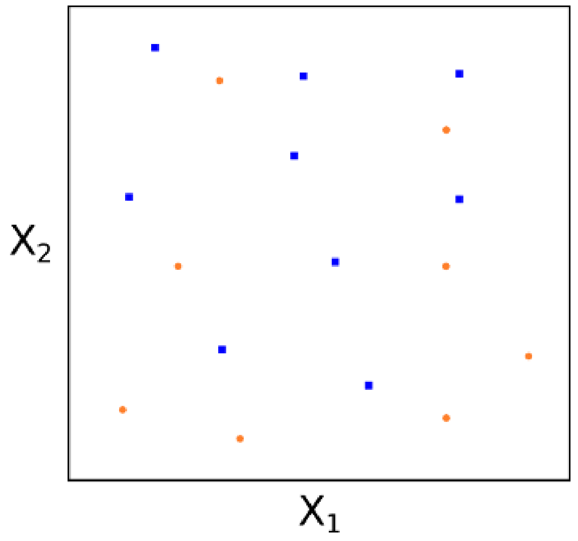
- The procedure for determining a maximal margin hyperplane for a maximal margin classifier is as follows. Given n training observations $\vec{x}_1, \ldots, \vec{x}_n \in \Re^p$ and *n* class labels $y_1, \ldots, y_n \in \{-1,1\}$, the MMH is the solution to the following optimization procedure:
- Maximize $M \in \Re$, by varying $b_1, \ldots, b_p$ such that:

$$\sum_{j=1}^{p} b_j^2 = 1$$

and

$$y_i(\vec{b}.\vec{x} + b_0) \geq M, \forall i = 1, \ldots, n$$

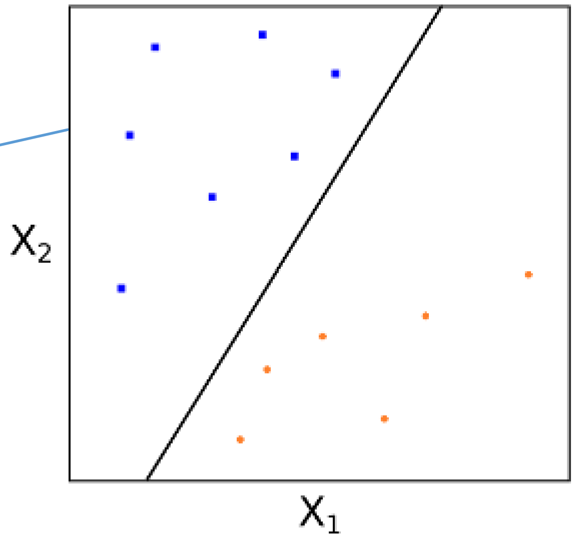
- Despite the complex looking constraints, they actually state that each observation must be on the correct side of the hyperplane and at least a distance M from it. Since the goal of the procedure is to maximize M, this is precisely the condition we need to create the MMC.
- Clearly, the case of perfect separability is an ideal one. Most "real world" datasets will not have such perfect separability via a linear hyperplane. However, if there is no separability then we are unable to construct a MMC by the optimization procedure above. So, how do we create a form of separating hyperplane?

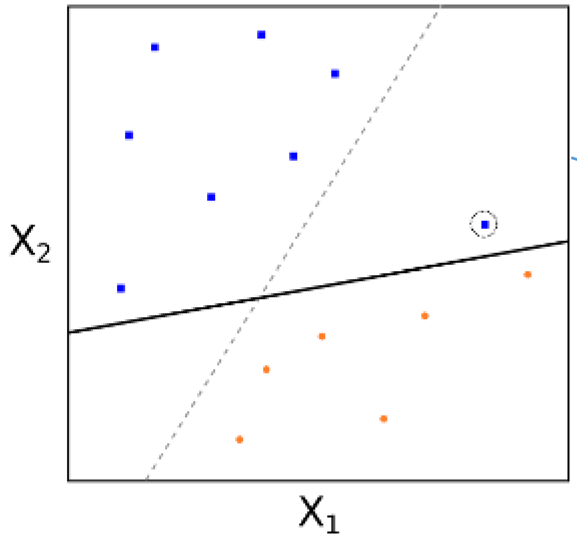


Support Vector Classifiers

- Essentially we have to relax the requirement that a separating hyperplane will perfectly separate every training observation on the correct side of the line (i.e. guarantee that it is associated with its true class label), using what is called a **soft margin**. This motivates the concept of a **support vector classifier**(SVC).
- MMCs can be extremely sensitive to the addition of new training observations.

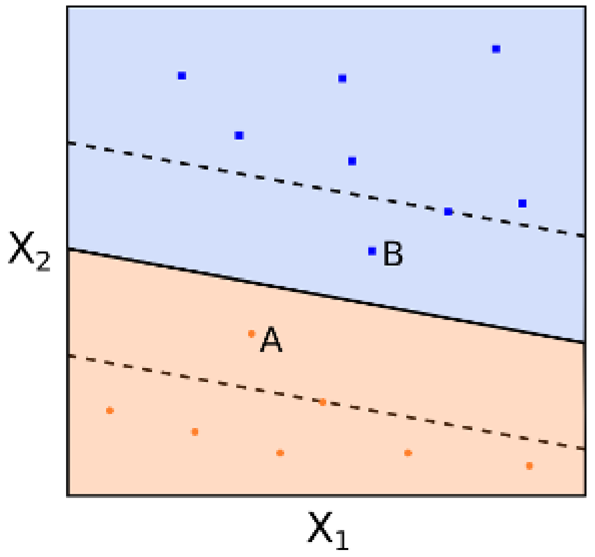
MMH perfectly separating the two classes

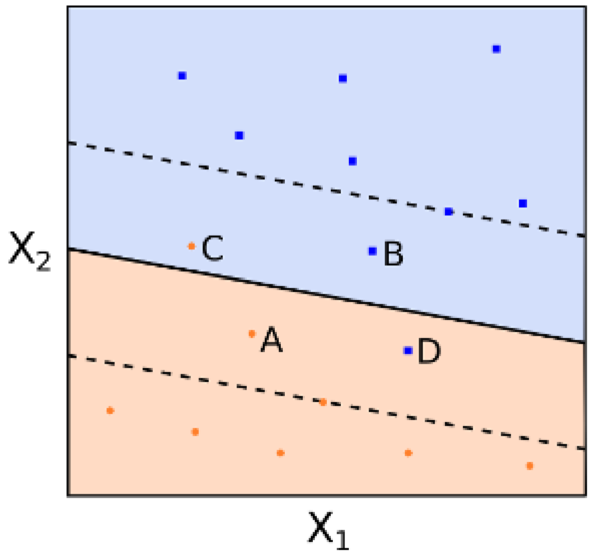




If we add one point to the +1 class, we see that the location of the MMH changes substantially. Hence in this situation the MMH has clearly been over-fit.

- We could consider a classifier based on a separating hyperplane that **doesn't perfectly separate** the two classes, but does have a **greater robustness** to the addition of new individual observations and has a better classification on most of the training observations. This comes at the expense of some misclassification of a few training observations.
- This is how a support vector classifier or soft margin classifier works. A SVC allows some observations to be on the incorrect side of the margin (or hyperplane), hence it provides a "soft" separation. The following figures demonstrate observations being on the wrong side of the margin and the wrong side of the hyperplane respectively:





- As before, an observation is classified depending upon which side of the separating hyperplane it lies on, but some points may be misclassified.
- It is instructive to see how the optimization procedure differs from that described above for the MMC. We need to introduce new parameters, namely *n* ϵ_i values (known as the **slack values**) and a parameter C, known as the budget. We wish to maximize M, across $b_1,\ldots,b_p,\epsilon_1,\ldots,\epsilon_n$ such that:

$$\sum_{j=1}^{p} b_j^2 = 1$$

and

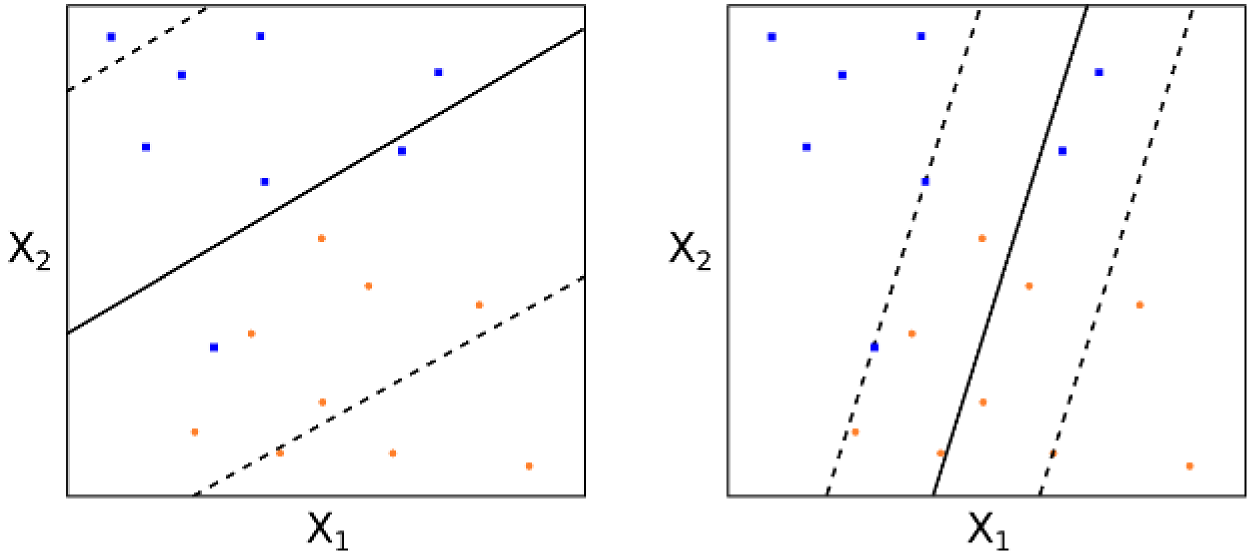
$$y_i(\vec{b}.\vec{x} + b_0) \geq M(1 - \epsilon_i), \forall i = 1, \ldots, n$$

and $\epsilon_i \geq 0,\ \sum_{i=1}^{n} \epsilon_i \leq C.$

where *C*, the budget, is a non-negative "tuning" parameter. *M* still represents the margin and the slack variables ϵ_i allow the individual observations to be on the wrong side of the margin or hyperplane.

- In essence the ϵ_i tell us where the i-th observation is located relative to the margin and hyperplane. For $\epsilon_i=0$ it states that the x_i training observation is on the correct side of the margin. For $\epsilon_i>0$ we have that x_i is on the wrong side of the margin, while for $\epsilon_i>1$ we have that x_i is on the wrong side of the hyperplane.
- C collectively controls how much the individual ϵ_i can be modified to violate the margin. C=0 implies that $\epsilon_i=0, \forall i$ and thus no violation of the margin is possible, in which case (for separable classes) we have the MMC situation.

- For C>0 it means that no more than C observations can violate the hyperplane. As C increases the margin will widen. See figures for two differing values of C:

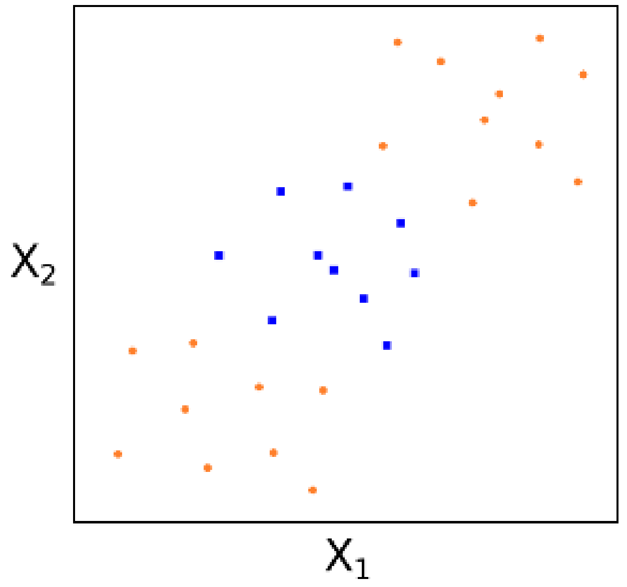


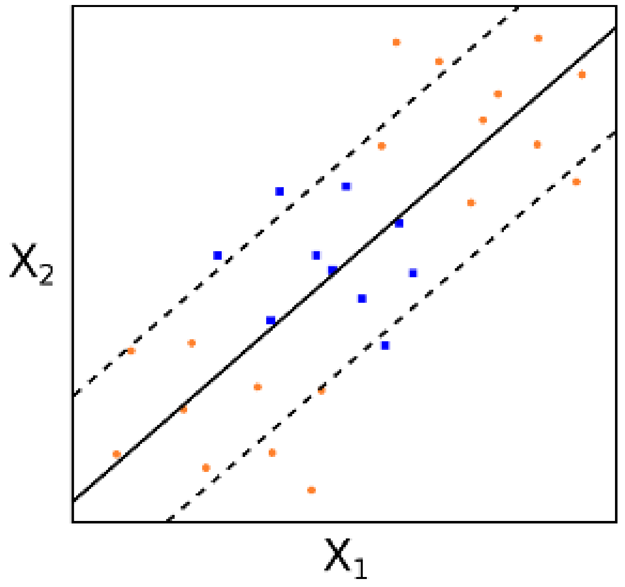
- How do we choose C in practice? Generally this is done via cross-validation. In essence C is the parameter that governs the bias-variance trade-off for the SVC. A small value of C means a low bias, high variance situation. A large value of C means a high bias, low variance situation.

- As before, to classify a new test observation x^* we simply calculate the sign of $f(\vec{x}^*) = \vec{b}.\vec{x}^* + b_0$.
- This is all well and good for classes that are linearly (or nearly linearly) separated. However, what about separation boundaries that are non-linear? How do we deal with those situations? This is where we can extend the concept of support vector classifiers to support vector machines.

Support Vector Machines

- The motivation behind the extension of a SVC is to allow non-linear decision boundaries. This is the domain of the Support Vector Machine (SVM). Consider the following figures. In such a situation a purely linear SVC will have extremely poor performance, simply because the data has no clear linear separation:





- Hence SVCs can be useless in highly non-linear class boundary problems.
- In order to motivate how an SVM works, we can consider a standard "trick" in linear regression, when considering non-linear situations. In particular a set of p features $x_1, \ldots, x_p$ can be transformed, say, into a set of 2p features $x_1, x_1^2, \ldots, x_p, x_p^2$. This allows us to apply a linear technique to a set of non-linear features.
- While the decision boundary is linear in the new 2p-dimensional feature space it is non-linear in the original p-dimensional space. We end up with a decision boundary given by q($\vec{x}$)=0 where q is a quadratic polynomial function of the original features and hence is a non-linear solution.

- This is clearly not restricted to quadratic polynomials. Higher dimensional polynomials, interaction terms and other functional forms, could all be considered. Although the drawback is that it dramatically increases the dimension of the feature space to the point that some algorithms can become untractable.

- The major advantage of SVMs is that they allow a non-linear enlargening of the feature space, while still retaining a significant computational efficiency, using a process known as the "**kernel trick**.

- So what are SVMs? In essence they are an extension of SVCs that results from enlargening the feature space through the use of functions known as kernels. In order to understand kernels, we need to briefly discuss some aspects of the solution to the SVC optimization problem outlined above.

- While calculating the solution to the SVC optimization problem, the algorithm only needs to make use of inner products between the observations and not the observations themselves. Recall that an inner product is defined for two p-dimensional vectors u, v as:

$$\langle u, v \rangle = \sum_{j=1}^{p} u_j v_j$$

Hence for two observations an inner product is defined as:

$$\langle \vec{x}_i, \vec{x}_k \rangle = \sum_{j=1}^{p} x_{ij} x_{kj}$$

- While we won't dwell on the details, it is possible to show that a linear support vector classifier for a particular observation $\vec{x}$ can be represented as a linear combination of inner products:

$$f(\vec{x}) = b_0 + \sum_{i=1}^{n} b_i \langle \vec{x}, \vec{x}_i \rangle$$

- With n α_i coefficients, one for each of the training observations.

- To estimate the b_0 and b_i coefficients we only need to calculate $\binom{n}{2} = \frac{n(n-1)}{2}$ inner products between all pairs of training observations. In fact, we ONLY need to calculate the inner products for the subset of training observations that represent the support vectors. I will call this subset $\mathfrak{I}$.

- This means that:

$$b_i = 0 \; if \; \vec{x}_i \notin \mathfrak{I}.$$

This means we can rewrite the representation formula as:

$$f(\vec{x}) = b_0 + \sum_{i \in \mathfrak{I}} b_i \langle \vec{x}, \vec{x}_i \rangle.$$

This turns out to be a major advantage for computational efficiency.

This now motivates the extension to SVMs. If we consider the inner product $\langle \vec{x}_i, \vec{x}_k \rangle$ and replace it with a more general inner product "**kernel**" function K=$K(\vec{x}_i, \vec{x}_k)$, we can modify the SVC representation to use non-linear kernel functions and thus modify how we calculate "similarity" between two observations.

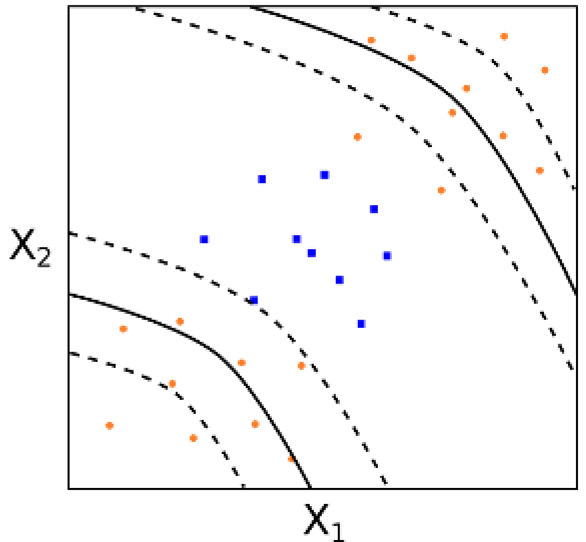
- For instance, to recover the SVC we just take K to be as follows:

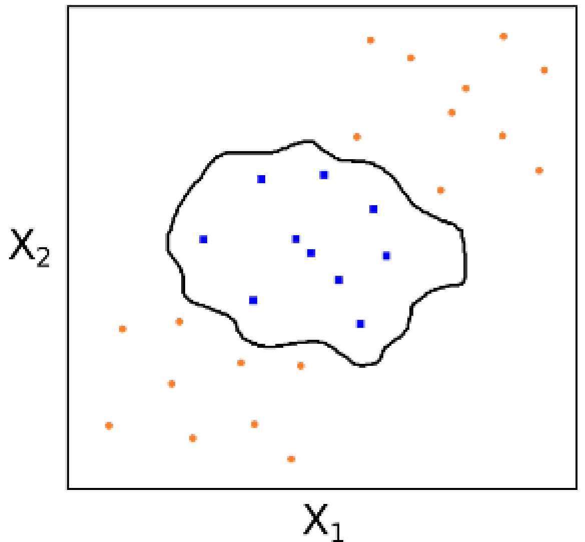
$$K(\vec{x}_i, \vec{x}_k) = \sum_{j=1}^{p} x_{ij}x_{kj}.$$

Since this kernel is linear in its features the SVC is known as the linear SVC. We can also consider polynomial kernels, of degree d:

$$K(\vec{x}_i, \vec{x}_k) = \left(1 + \sum_{j=1}^{p} x_{ij}x_{kj}\right).$$

This provides a significantly more flexible decision boundary and essentially amounts to fitting a SVC in a higher-dimensional feature space involving d-degree polynomials of the features





- Hence, the definition of a support vector machine is a support vector classifier with a non-linear kernel function.
- We can also consider the popular radial kernel

$$K(\vec{x}_i, \vec{x}_k) = exp\left(-\gamma \sum_{j=1}^{p} \left(x_{ij} - x_{kj}\right)^2\right), \gamma > 0.$$

- So how do radial kernels work? They are clearly quite different from polynomial kernels. Essentially if our test observation $\vec{x}^*$ is far from a training observation $\vec{x}_i$ in standard Euclidean distance then the sum $\sum_{j=1}^{p}\left(x_{ij} - x_{kj}\right)^2$ will be large and thus K($\vec{x}^*$,$\vec{x}_i^*$) will be very small. Hence this particular training observation $\vec{x}_i$ will have almost no effect on where the test observation $\vec{x}^*$ is placed, via f($\vec{x}^*$).

- Thus the radial kernel has extremely localized behavior and only nearby training observations to $\vec{x}^*$ will have an impact on its class label.

SVM Kernel Functions

- SVM algorithms use a set of mathematical functions that are defined as the kernel. The function of kernel is to take data as input and transform it into the required form. Different SVM algorithms use different types of kernel functions. These functions can be different types. For example ***linear, nonlinear, polynomial, radial basis function (RBF), and sigmoid.***
- Introduce Kernel functions for sequence data, graphs, text, images, as well as vectors. The most used type of kernel function is **RBF.** Because it has localized and finite response along the entire x-axis.
- The kernel functions return the inner product between two points in a suitable feature space. Thus by defining a notion of similarity, with little computational cost even in very high-dimensional spaces.

Introduction to SVM in R

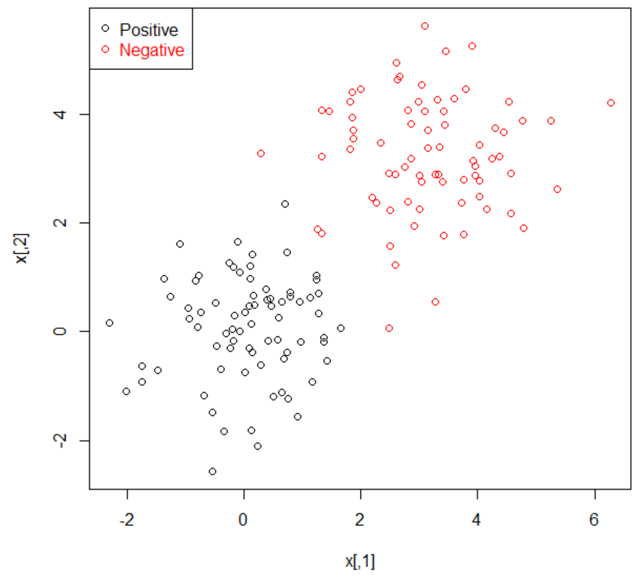
Linear SVM

Here we generate a toy dataset in 2D, and learn how to train and test a SVM.

1.1 Generate toy data First generate a set of positive and negative examples from 2 Gaussians.

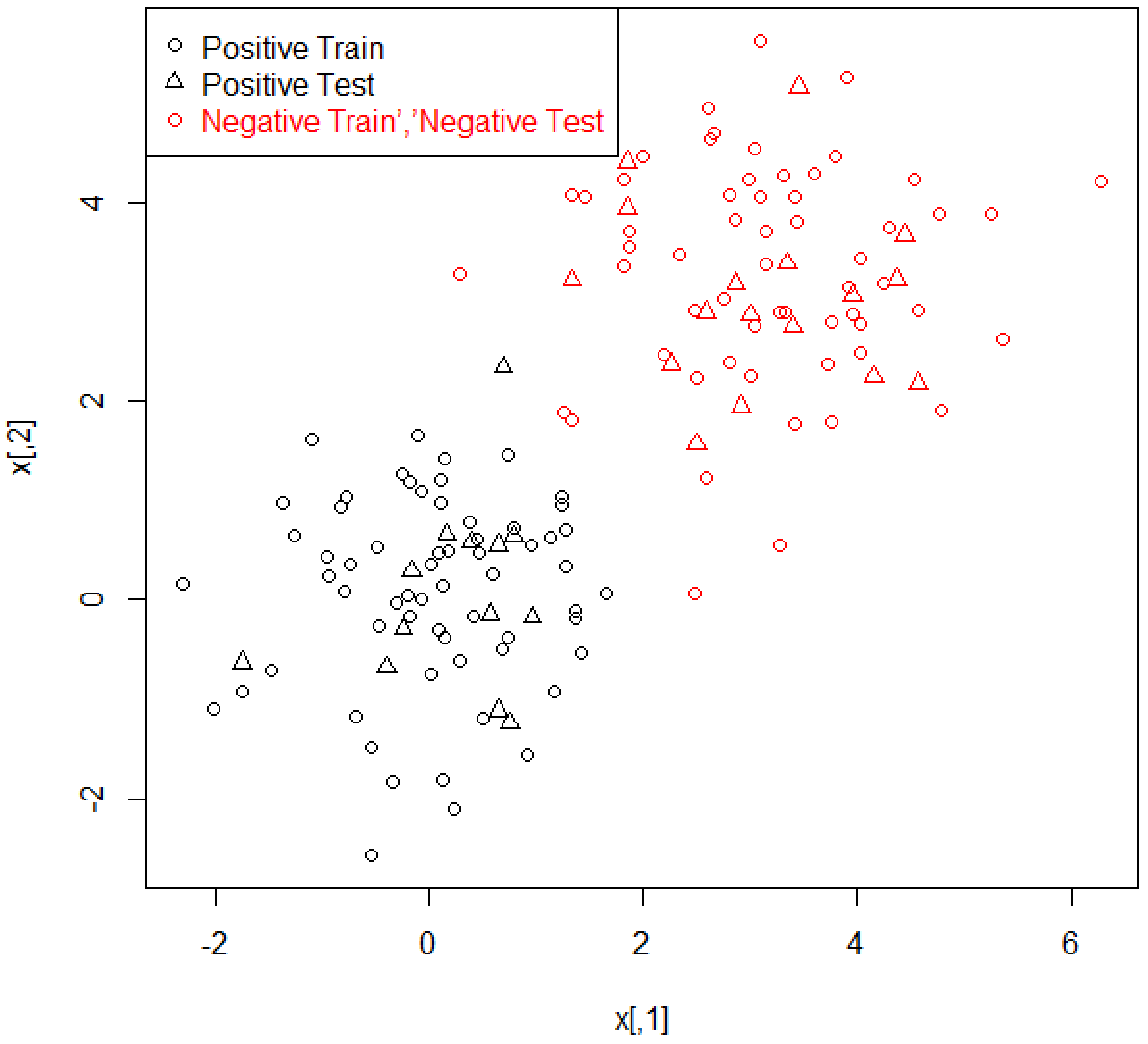
```
n <- 150 # number of data points
p <- 2 # dimension
sigma <- 1 # variance of the distribution
meanpos <- 0 # centre of the distribution of positive examples
meanneg <- 3 # centre of the distribution of negative examples
npos <- round(n/2) # number of positive examples
nneg <- n-npos # number of negative examples
```

```
# Generate the positive and negative examples
xpos <- matrix(rnorm(npos*p,mean=meanpos,sd=sigma),npos,p)
xneg <- matrix(rnorm(nneg*p,mean=meanneg,sd=sigma),npos,p)
x <- rbind(xpos,xneg)
# Generate the labels
y <- matrix(c(rep(1,npos),rep(-1,nneg)))
# Visualize the data
plot(x,col=ifelse(y>0,1,2))
legend("topleft",c('Positive','Negative'),col=seq(2),pch=1,text.col
=seq(2))
```



- Now we split the data into a training set (80%) and a test set (20%):

```
## Prepare a training and a test set ##
ntrain <- round(n*0.8) # number of training examples
tindex <- sample(n,ntrain) # indices of training samples
xtrain <- x[tindex,]
xtest <- x[-tindex,]
ytrain <- y[tindex]
ytest <- y[-tindex]
istrain=rep(0,n)
istrain[tindex]=1
# Visualize
plot(x,col=ifelse(y>0,1,2),pch=ifelse(istrain==1,1,2))
legend("topleft",c('Positive Train','Positive Test','Negative
Train’,’Negative Test'),
col=c(1,1,2,2),pch=c(1,2,1,2),text.col=c(1,1,2,2))
```


Positive Train
Positive Test
Negative Train','Negative Test
x[,2]
x[,1]
-2
0
2
4
6
-2
0
2
4

Train a SVM

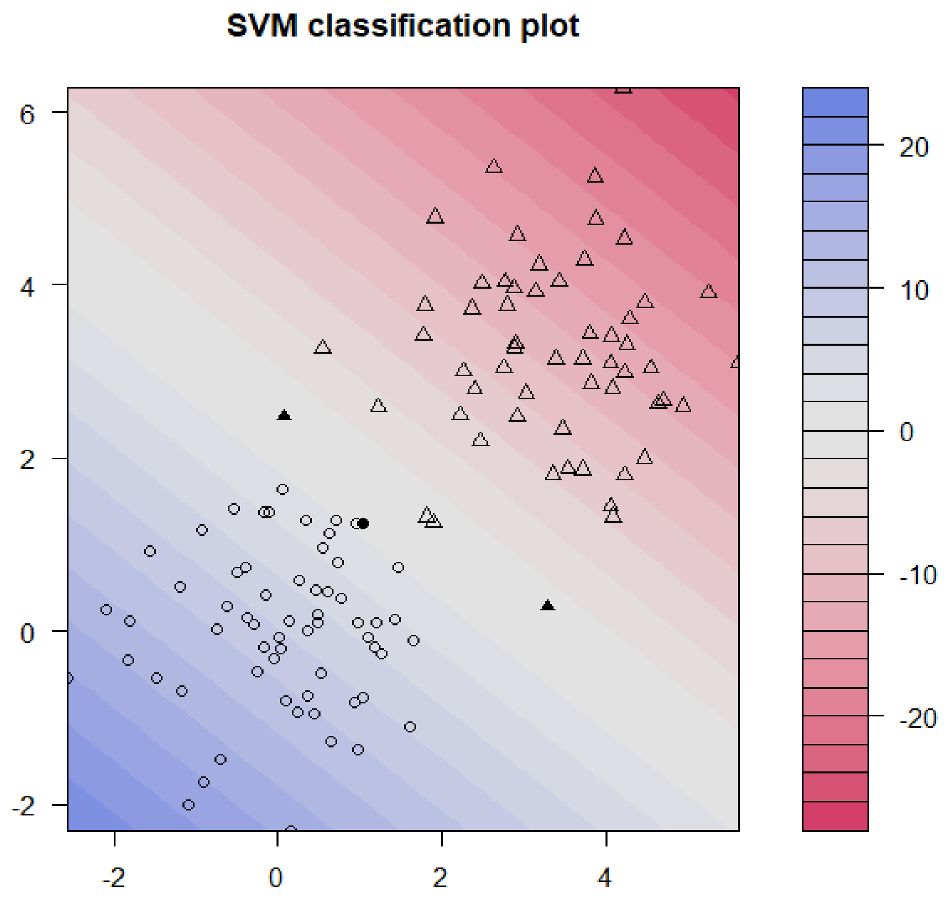
Now we train a linear SVM with parameter C=100 on the training set

```
# load the kernlab package
library(kernlab)
# train the SVM
svp <- ksvm(xtrain,ytrain,type="C-svc", kernel='vanilladot', C=100, scaled=c())
```

Look and understand what svp contains

```
# General summary
svp
Support Vector Machine object of class "ksvm"

SV type: C-svc  (classification)
 parameter : cost C = 100

Linear (vanilla) kernel function.

Number of Support Vectors : 3

Objective Function Value : -8.7968
Training error : 0
```

```
# Attributes that you can access
attributes(svp)
# For example, the support vectors
alpha(svp)
alphaindex(svp)
b(svp)
# Use the built-in function
to pretty-plot the classifier
plot(svp,data=xtrain)
```

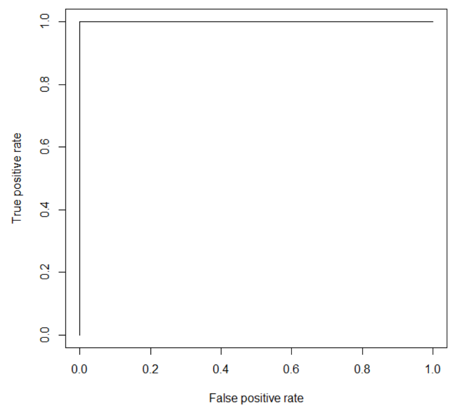


Predict with a SVM

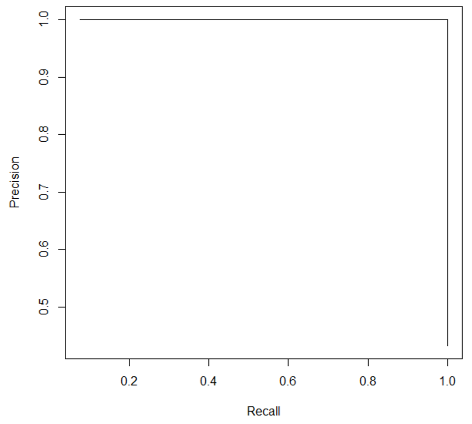
- Now we can use the trained SVM to predict the label of points in the test set, and we analyze the results using variant metrics.

```
# Predict labels on test
ypred = predict(svp,xtest)
table(ytest,ypred)
      ypred
ytest -1  1
   -1 17  0
    1  1 12
# Compute accuracy
sum(ypred==ytest)/length(ytest)
[1] 0.9666667
# Compute at the prediction scores
ypredscore = predict(svp,xtest,type="decision")
```

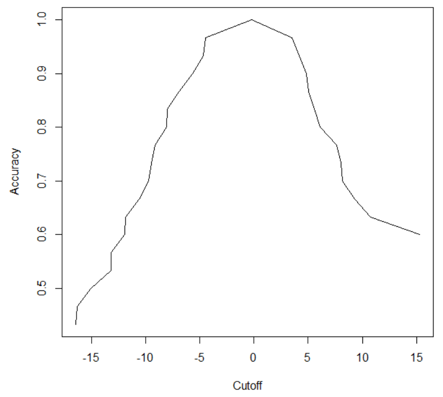
```
# Check that the predicted labels are the signs of the scores
table(ypredscore > 0,ypred)
       ypred
        -1  1
  FALSE 18  0
  TRUE   0 12
# Package to compute ROC curve, precision-recall etc...
library(ROCR)
pred <- prediction(ypredscore,ytest)
# Plot ROC curve
perf <- performance(pred, measure = "tpr", x.measure = "fpr")
plot(perf)
```



```
 # Plot precision/recall curve
perf <- performance(pred, measure = "prec",
x.measure = "rec")
plot(perf)
```



```
# Plot accuracy as function of threshold
perf <- performance(pred, measure = "acc")
plot(perf)
```



Cross-validation

- Instead of fixing a training set and a test set, we can improve the quality of these estimates by running k-fold cross-validation. We split the training set in k groups of approximately the same size, then iteratively train a SVM using k − 1 groups and make prediction on the group which was left aside. When k is equal to the number of training points, we talk of leave-one-out (LOO) cross-validation. To generate a random split of n points in k folds, we can for example create the following function:

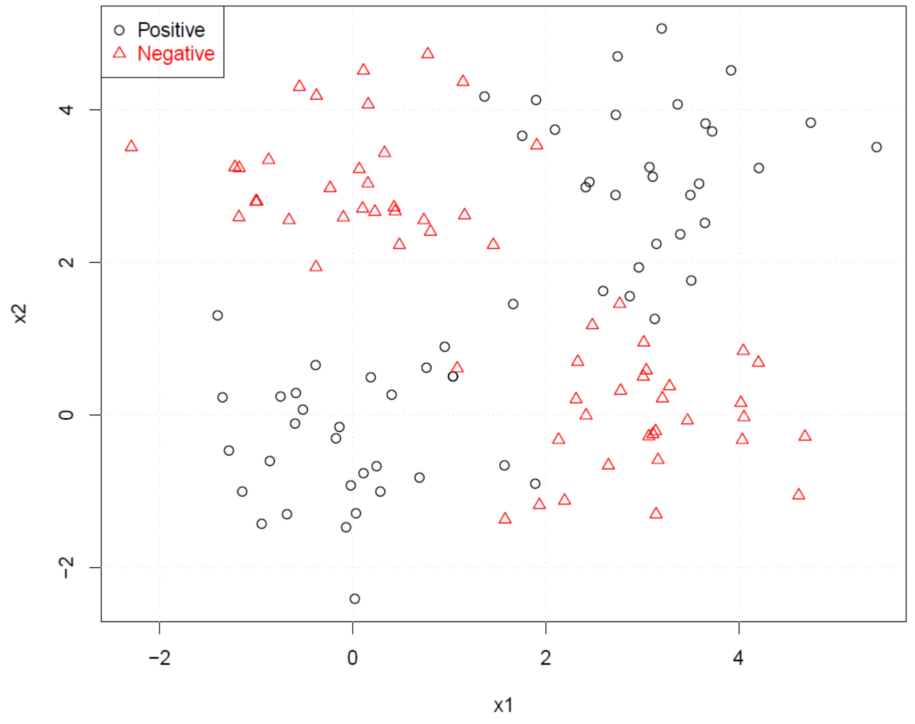
```
cv.folds <- function(n,folds=3)
## randomly split the n samples into folds
{
split(sample(n),rep(1:folds,length=length(y)))
}
```

Effect of C

- The C parameters balances the trade-off between having a large margin and separating the positive and unlabeled on the training set. It is important to choose it well to have good generalization.
- Plot the decision functions of SVM trained on the toy examples for different values of C in the range 2^seq(-10,15). To look at the different plots you can use the function par(ask=T) that will ask you to press a key between successive plots.

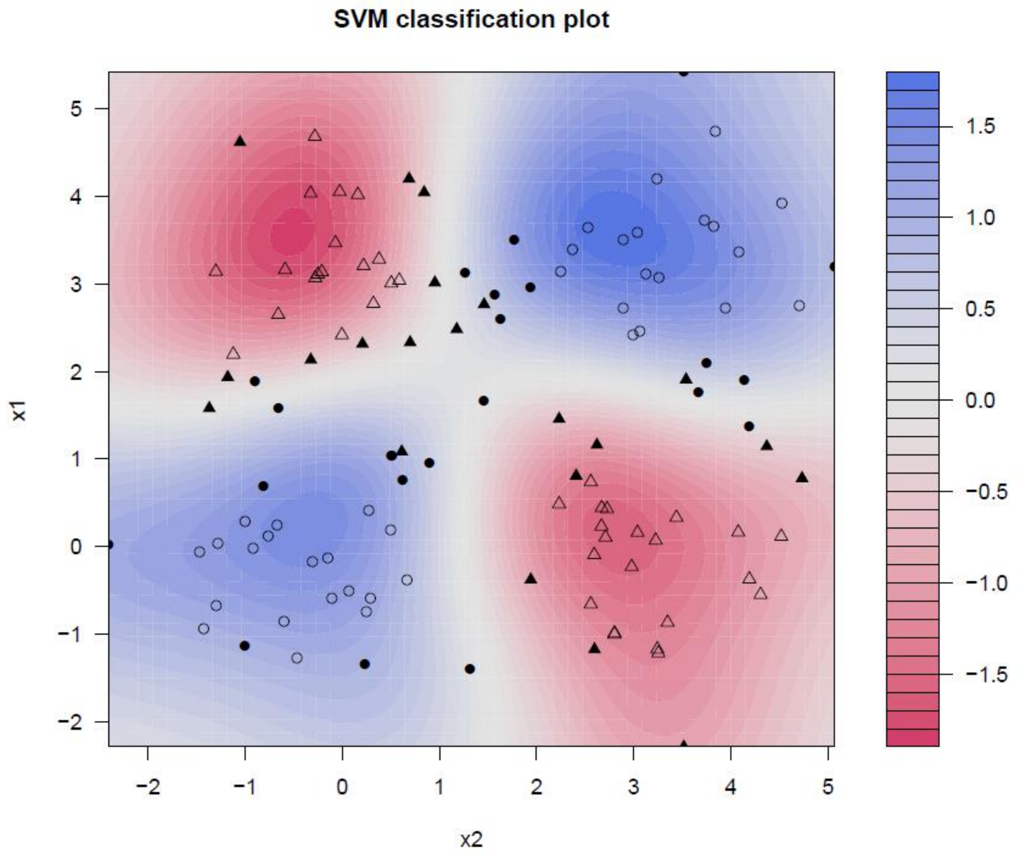
Nonlinear SVM

- Sometimes linear SVM are not enough. For example, generate a toy dataset where positive and negative examples are mixture of two Gaussians which are not linearly separable.



- To solve this problem, we should instead use a nonlinear SVM. This is obtained by simply changing the kernel parameter. For example, to use a Gaussian RBF kernel with = 1 and C = 1:

```
# Train a nonlinear SVM
svp <- ksvm(x,y,type="C-svc",kernel='rbf',kpar=list(sigma=1),C=1)
# Visualize it
plot(svp,data=x)
```



SVM on IRIS Dataset

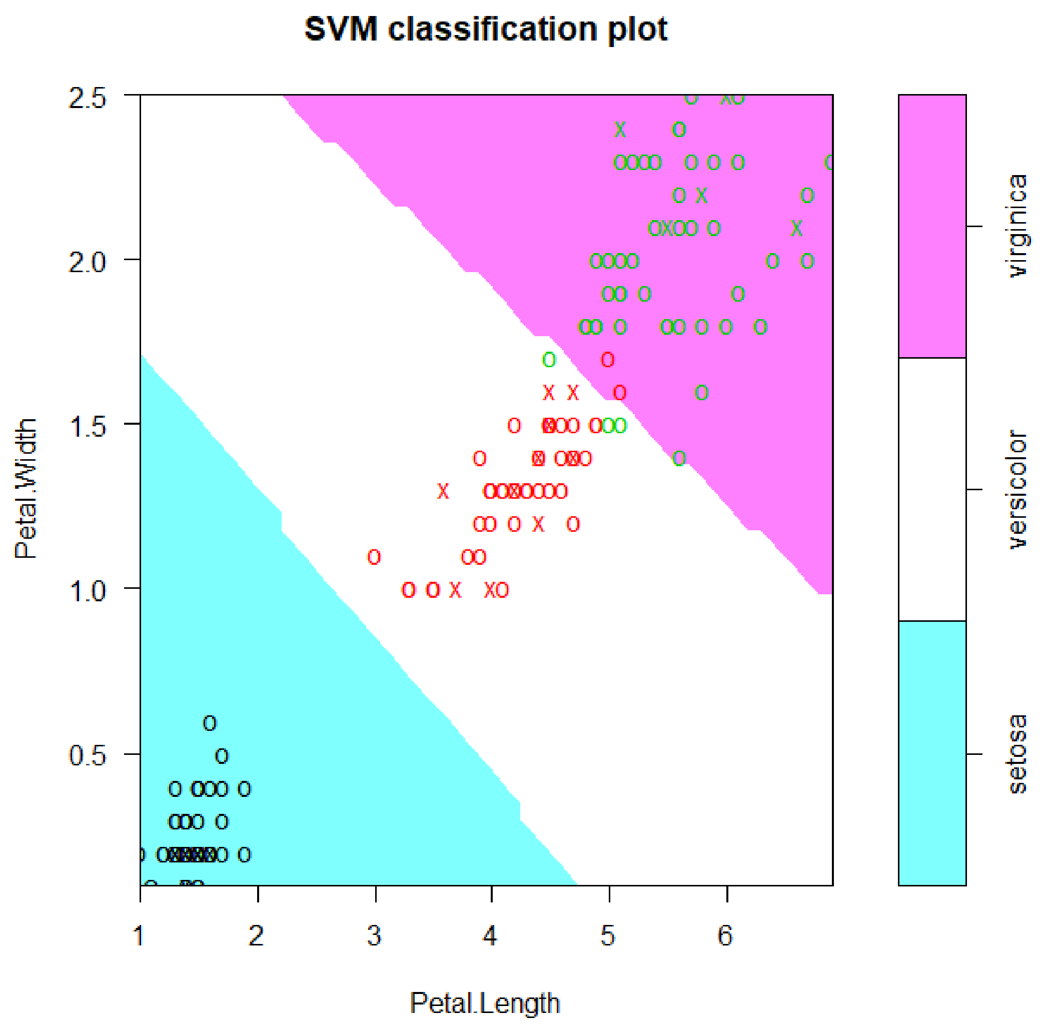
```
library(e1071)
data(iris)
head(iris,5)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
attach(iris)
```

```
#set seed to ensure reproducible results
set.seed(42)
#split into training and test sets
iris[,"train"] <- ifelse(runif(nrow(iris))<0.8,1,0)
#separate training and test sets
trainset <- iris[iris$train==1,]
testset <- iris[iris$train==0,]
#get column index of train flag
trainColNum <- grep("train",names(trainset))
#remove train flag column from train and test sets
trainset <- trainset[,-trainColNum]
testset <- testset[,-trainColNum]
#get column index of predicted variable in dataset
typeColNum <- grep("Species",names(iris))
```

```
#build model – linear kernel and C-classification (soft
margin) with default cost (C=1)

svm_model <- svm(Species~ ., data=trainset, method="C-
classification", kernel="linear")

svm_model

Call:

svm(formula = Species ~ ., data = trainset, method = "C-
classification",
    kernel = "linear")


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  1
      gamma:  0.25

Number of Support Vectors:  24
```

```
#training set predictions
pred_train <-predict(svm_model,trainset)
mean(pred_train==trainset$Species)
[1] 0.9826087
#test set predictions
pred_test <-predict(svm_model,testset)
mean(pred_test==testset$Species)
[1] 0.9142857
```

```
# Visualize it
plot(svm_model, iris, Petal.Width ~ Petal.Length,
     slice = list(Sepal.Width = 3, Sepal.Length = 4))
```



How to tune Parameters of SVM?

- Tuning parameter 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)
```

Kernel

- 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 hyperplane. 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 rbf kernel

```
svm_model_rbf <- svm(Species~ ., data=trainset, method="C-
classification", cost=1, kernel="radial", gamma=0)
> #training set predictions
> pred_train <-predict(svm_model_rbf,trainset)
> mean(pred_train==trainset$Species)
[1] 0.3913043
> #test set predictions
> pred_test <-predict(svm_model_rbf,testset)
> mean(pred_test==testset$Species)
[1] 0.1428571
```

Gamma

- Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’. Higher the value of gamma, will try to exact fit as per training data set i.e. generalization error and cause over-fitting problem.
- Let’s see the difference if we have different gamma values like 0 (previous) or 10.

```
svm_model_g10 <- svm(Species~ ., data=trainset, method="C-classification", cost=1, kernel="linear", gamma=10)

#training set predictions
pred_train <-predict(svm_model_g10,trainset)
mean(pred_train==trainset$Species)
[1] 0.9826087
#test set predictions
pred_test <-predict(svm_model_g10,testset)
mean(pred_test==testset$Species)
[1] 0.9142857
```

Cost

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

```
> svm_model_c100 <- svm(Species~ ., data=trainset, method="C-classification", cost=100, kernel="linear", gamma=1)
>
> #training set predictions
> pred_train <-predict(svm_model_c100,trainset)
> mean(pred_train==trainset$Species)
[1] 0.9913043
> #test set predictions
> pred_test <-predict(svm_model_c100,testset)
> mean(pred_test==testset$Species)
[1] 0.9714286
```

R Example on Nonlinear SVM

```
#load required library
library(e1071)
library(mlbench)
#load Sonar dataset
data(Sonar)
head(Sonar)
      V1     V2     V3     V4     V5     V6     V7     V8     V9    V10    V11
1 0.0200 0.0371 0.0428 0.0207 0.0954 0.0986 0.1539 0.1601 0.3109 0.2111 0.1609
2 0.0453 0.0523 0.0843 0.0689 0.1183 0.2583 0.2156 0.3481 0.3337 0.2872 0.4918
3 0.0262 0.0582 0.1099 0.1083 0.0974 0.2280 0.2431 0.3771 0.5598 0.6194 0.6333
4 0.0100 0.0171 0.0623 0.0205 0.0205 0.0368 0.1098 0.1276 0.0598 0.1264 0.0881
5 0.0762 0.0666 0.0481 0.0394 0.0590 0.0649 0.1209 0.2467 0.3564 0.4459 0.4152
6 0.0286 0.0453 0.0277 0.0174 0.0384 0.0990 0.1201 0.1833 0.2105 0.3039 0.2988
```

```
#set seed to ensure reproducible results
set.seed(42)
#split into training and test sets
Sonar[,”train”] <- ifelse(runif(nrow(Sonar))<0.8,1,0)
#separate training and test sets
trainset <- Sonar[Sonar$train==1,]
testset <- Sonar[Sonar$train==0,]
#get column index of train flag
trainColNum <- grep("train",names(trainset))
#remove train flag column from train and test sets
trainset <- trainset[,-trainColNum]
testset <- testset[,-trainColNum]
```

```
#get column index of predicted variable in dataset
typeColNum <- grep("Class",names(Sonar))
#build model – linear kernel and C-classification
with default cost (C=1)
svm_model <- svm(Class~ ., data=trainset, method="C-
classification", kernel="linear")
#training set predictions
pred_train <-predict(svm_model,trainset)
mean(pred_train==trainset$Class)
[1] 0.969697
#test set predictions
pred_test <-predict(svm_model,testset)
mean(pred_test==testset$Class)
[1] 0.6046512
```

- The important point to note here is that the performance of the model with the test set is quite dismal compared to the previous case. This simply indicates that the linear kernel is not appropriate here. Let’s take a look at what happens if we use the RBF kernel with default values for the parameters:

```
#build model: radial kernel, default params
svm_model <- svm(Class~ ., data=trainset, method="C-classification",
kernel="radial")
#print params
svm_model$cost
[1] 1
svm_model$gamma
[1] 0.01666667
#training set predictions
pred_train <-predict(svm_model,trainset)
mean(pred_train==trainset$Class)
[1] 0.9878788
#test set predictions
pred_test <-predict(svm_model,testset)
mean(pred_test==testset$Class)
[1] 0.7674419
```

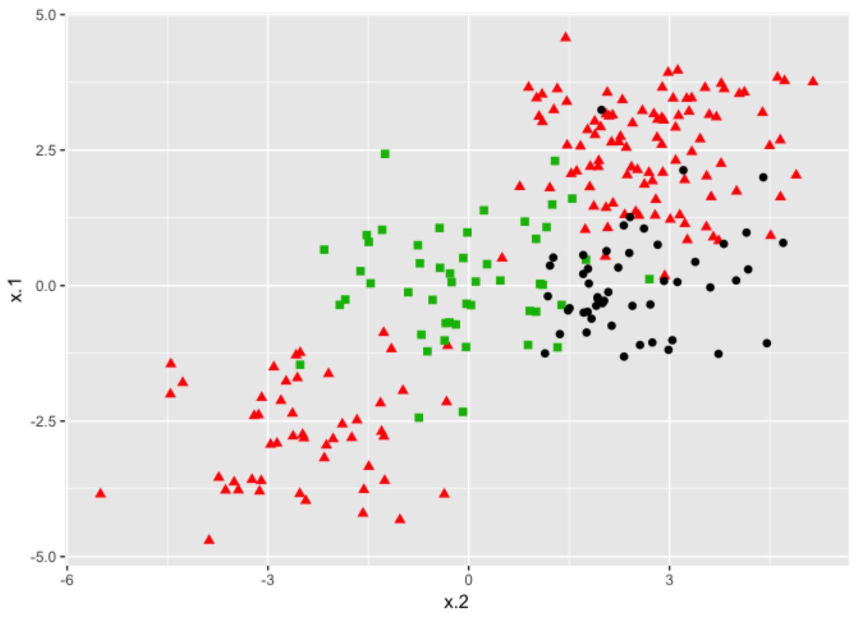
That's a pretty decent improvement from the linear kernel. Let's see if we can do better by doing some parameter tuning. To do this we first invoke *tune.svm* and use the parameters it gives us in the call to *svm*:

```
#find optimal parameters in a specified range

tune_out <- tune.svm(x=trainset[,-typeColNum],y=trainset[,typeColNum],gamma=10^(-
3:3),cost=c(0.01,0.1,1,10,100,1000),kernel="radial")

#print best values of cost and gamma

tune_out$best.parameters$cost

[1] 10

tune_out$best.parameters$gamma

[1] 0.01

#build model

svm_model <- svm(Class~ ., data=trainset, method="C-classification",
kernel="radial",cost=tune_out$best.parameters$cost,gamma=tune_out$best.parameters$gamma)

#training set predictions

pred_train <-predict(svm_model,trainset)

mean(pred_train==trainset$Class)

[1] 1

#test set predictions

pred_test <-predict(svm_model,testset)

mean(pred_test==testset$Class)

[1] 0.8139535
```

SVMs for Multiple Classes

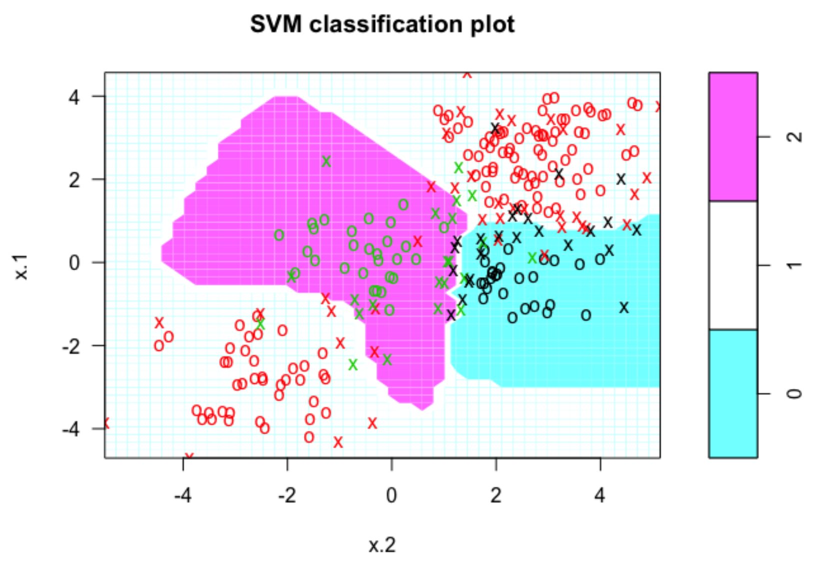
- The procedure does not change for data sets that involve more than two classes of observations. We construct our data set the same way as we have previously, only now specifying three classes instead of two:

```
# construct data set
x <- rbind(x, matrix(rnorm(50*2), ncol = 2))
y <- c(y, rep(0,50))
x[y==0,2] <- x[y==0,2] + 2.5
dat <- data.frame(x=x, y=as.factor(y))
# plot data set
ggplot(data = dat, aes(x = x.2, y = x.1, color = y, shape = y)) +
  geom_point(size = 2) +
  scale_color_manual(values=c("#000000","#FF0000","#00BA00")) +
  theme(legend.position = "none")
```



- We specify a cost and tuning parameter gamma and fit a support vector machine. The results and interpretation are similar to two-class classification.

```
# fit model
svmfit <- svm(y~., data = dat, kernel = "radial", cost =
10, gamma = 1)
# plot results
plot(svmfit, dat)
```



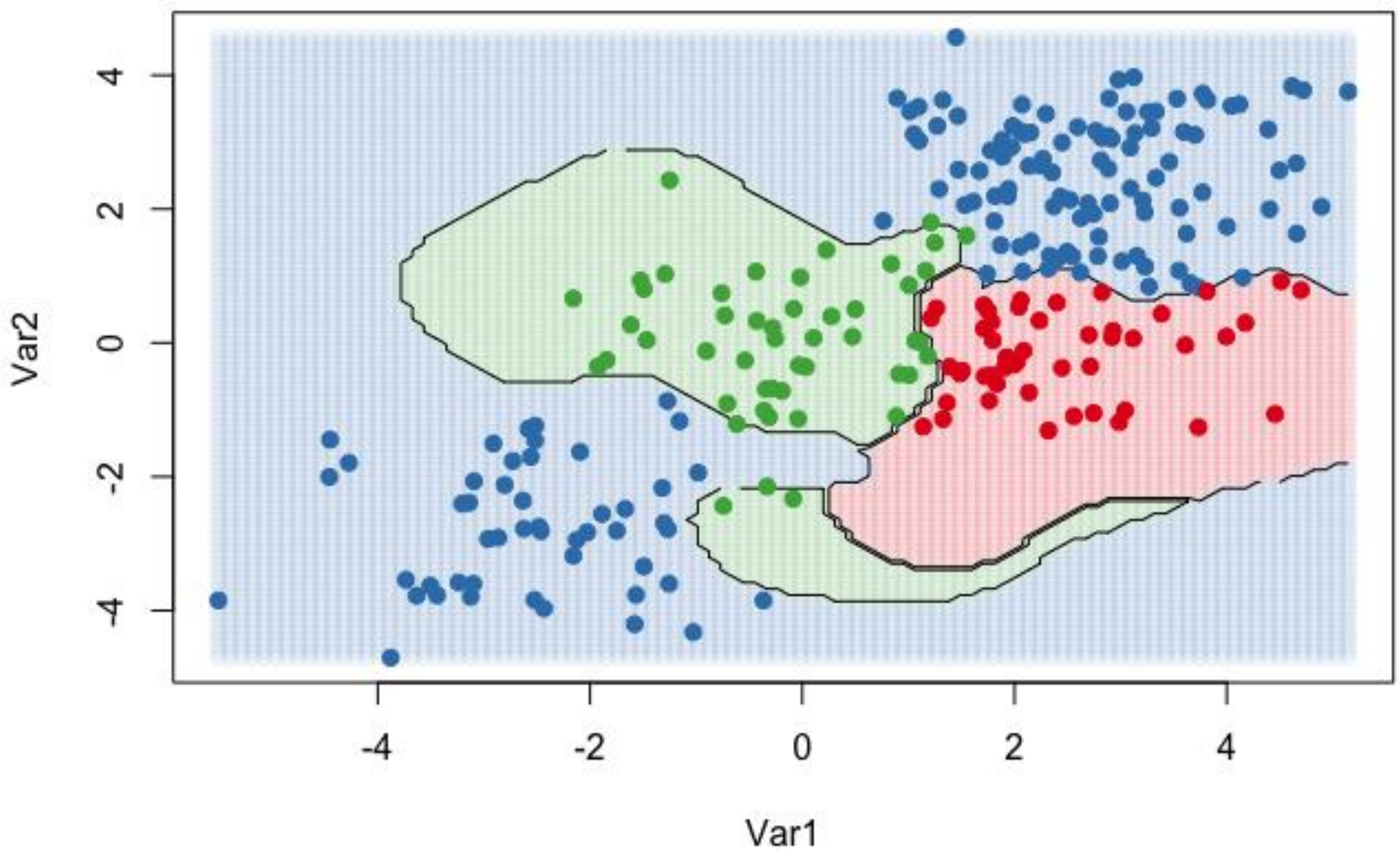
- We can check to see how well our model fit the data by using the predict() command, as follows:

```
#construct table
ypred <- predict(svmfit, dat)
(misclass <- table(predict = ypred, truth = dat$y))
##          truth
## predict    0   1   2
##        0  38   2   4
##        1   8 143   4
##        2   4   5  42
```

As shown in the resulting table, 89% of our training observations were correctly classified. However, since we didn't break our data into training and testing sets, we didn't truly validate our results.

- The kernlab package, on the other hand, can fit more than 2 classes, but cannot plot the results. To visualize the results of the ksvm function, we take the steps listed below to create a grid of points, predict the value of each point, and plot the results:

```
# fit and plot
kernfit <- ksvm(as.matrix(dat[,2:1]),dat$y, type = "C-svc", kernel = 'rbfdot',
                C = 100, scaled = c())
# Create a fine grid of the feature space
x.1 <- seq(from = min(dat$x.1), to = max(dat$x.1), length = 100)
x.2 <- seq(from = min(dat$x.2), to = max(dat$x.2), length = 100)
x.grid <- expand.grid(x.2, x.1)
# Get class predictions over grid
pred <- predict(kernfit, newdata = x.grid)
# Plot the results
cols <- brewer.pal(3, "Set1")
plot(x.grid, pch = 19, col = adjustcolor(cols[pred], alpha.f = 0.05))
classes <- matrix(pred, nrow = 100, ncol = 100)
contour(x = x.2, y = x.1, z = classes, levels = 1:3, labels = "", add = TRUE)
points(dat[, 2:1], pch = 19, col = cols[predict(kernfit)])
```


Var2
4
2
0
-2
-4
-4
-2
0
2
4
Var1

Support Vector Machine - Regression (SVR)

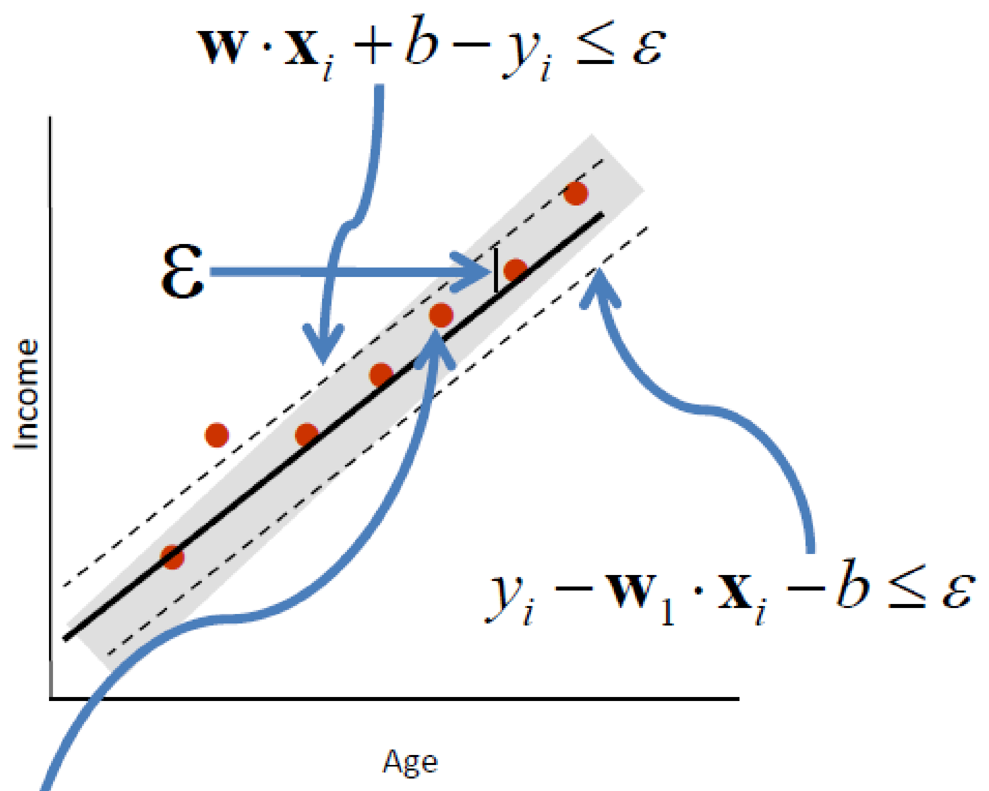
- Support Vector Machine can also be used as a regression method, maintaining all the main features that characterize the algorithm (maximal margin).
- The Support Vector Regression (SVR) uses the same principles as the SVM for classification, with only a few minor differences.
- First of all, because output is a real number it becomes very difficult to predict the information at hand, which has infinite possibilities.
- In the case of regression, a margin of tolerance (epsilon) is set in approximation to the SVM which would have already requested from the problem. But besides this fact, there is also a more complicated reason, the algorithm is more complicated therefore to be taken in consideration. However, the main idea is always the same: to minimize error, individualizing the hyperplane which maximizes the margin, keeping in mind that part of the error is tolerated.

Support Vector Regression
(https://cs.adelaide.edu.au/~chhshen/teaching/ML_SVR.pdf)

- Find a function, f(x), with at most ε-deviation from the target y
- The problem can be written as a convex optimization problem

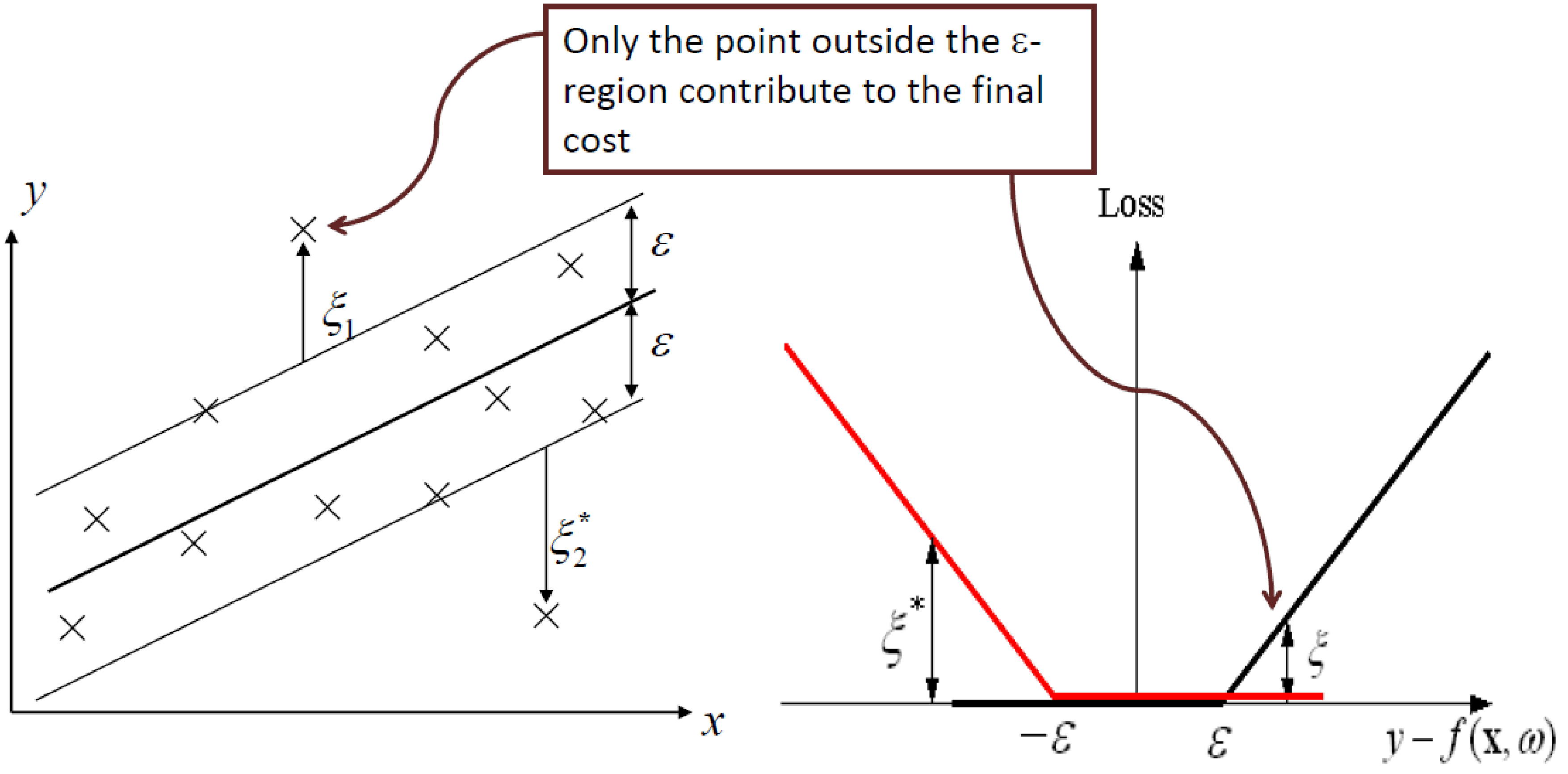
$$min \frac{1}{2} ||w||^2$$
$$such\ that\ y_i - \boldsymbol{w}_1.\boldsymbol{x}_i - b \leq \varepsilon;$$
$$\boldsymbol{w}_1.\boldsymbol{x}_i + b - y_i \leq \varepsilon$$

- C: trade off the complexity
- What if the problem is not feasible?
- We can introduce slack variables (similar to soft margin loss function).


$\mathbf{w}\cdot\mathbf{x}_i + b - y_i \le \varepsilon$
ε
Income
$y_i - \mathbf{w}_1\cdot\mathbf{x}_i - b \le \varepsilon$
Age

We do not care about errors as long as they are less than ε

Assume linear parameterization $f(\mathbf{x},\omega)=\mathbf{w}\cdot\mathbf{x}+b$

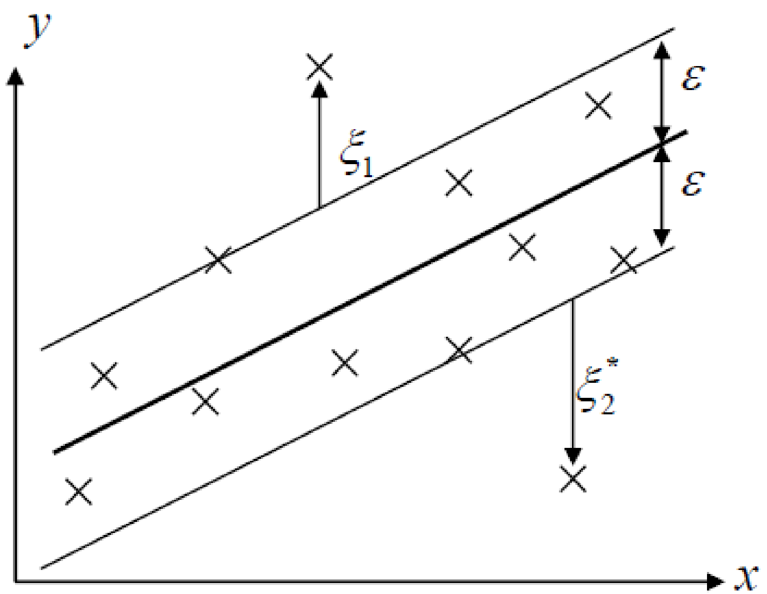


$$L_\varepsilon(y,f(\mathbf{x},\omega))=\max\left(|\,y-f(\mathbf{x},\omega)\,|-\varepsilon,0\right)$$

Soft Margin

Given training data

$$(\mathbf{x}_i, y_i) \quad i = 1, \ldots, m$$



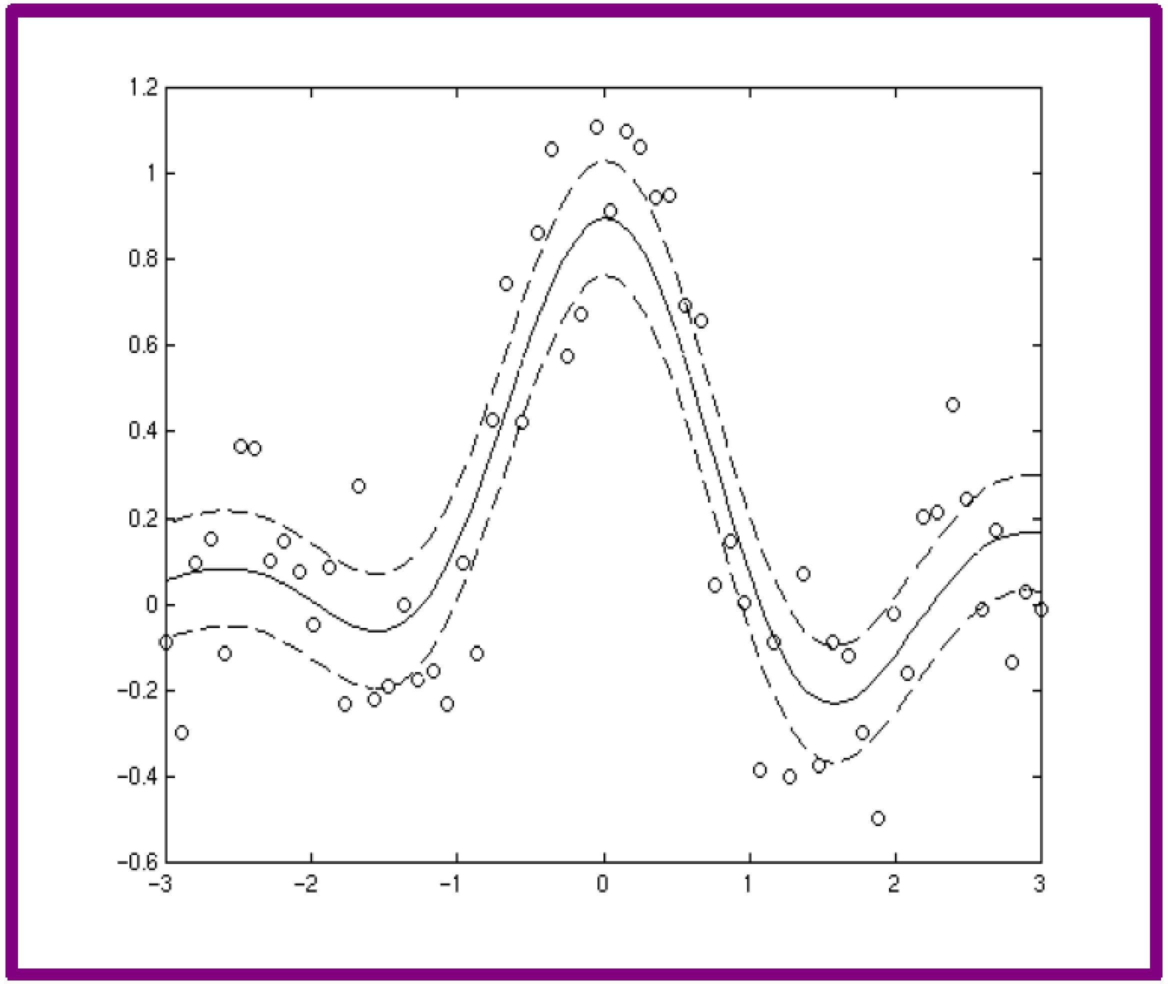
Minimize

$$\frac{1}{2} \| \mathbf{w} \|^2 + C \sum_{i=1}^{m} (\xi_i + \xi_i^*)$$

Under constraints

$$\begin{cases} y_i - (\mathbf{w} \cdot \mathbf{x}_i) - b \leq \varepsilon + \xi_i \\ (\mathbf{w} \cdot \mathbf{x}_i) + b - y_i \leq \varepsilon + \xi_i^* \\ \xi_i \;, \xi_i^* \geq 0, i = 1, \ldots, m \end{cases}$$

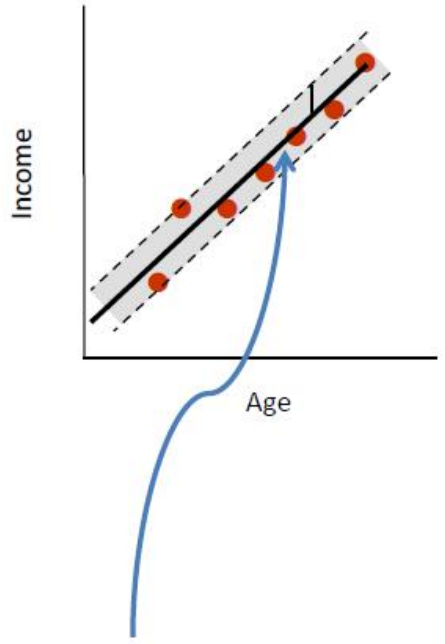
How about a non-linear case?



Linear versus Non-linear SVR

- **Linear case**

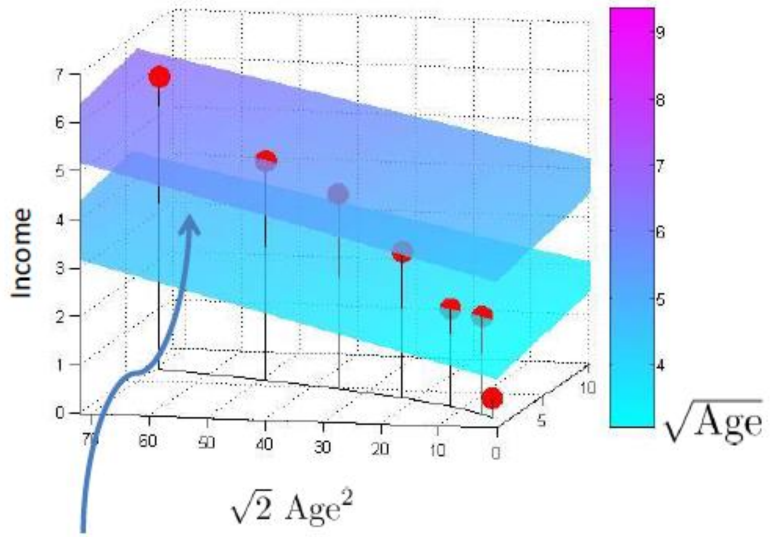
$$f : age \rightarrow income$$



$$y_i = \mathbf{w}_1 \cdot \mathbf{x}_i + b$$

- **Non-linear case**
  - Map data into a higher dimensional space, e.g.,

$$f : (\sqrt{age}, \sqrt{2}age^2) \rightarrow income$$



$$y_i = \mathbf{w}_1 \sqrt{\mathbf{x}_i} + \mathbf{w}_2 \sqrt{2}\mathbf{x}_i^2 + b$$

Dual Problem

- Primal

$$\min \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m}(\xi_i + \xi_i^*)$$

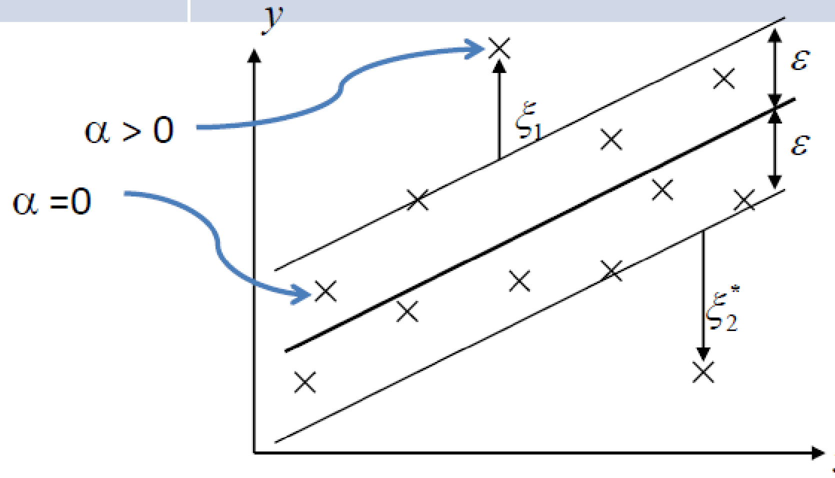
$$s.t.\begin{cases} y_i - (\mathbf{w}\cdot\mathbf{x}_i) - b \le \varepsilon + \xi_i \\ (\mathbf{w}\cdot\mathbf{x}_i) + b - y_i \le \varepsilon + \xi_i^* \\ \xi_i, \xi_i^* \ge 0, i = 1,\dots,m \end{cases}$$

- Dual

$$\max\begin{cases} \frac{1}{2}\sum_{i,j=1}^{m}(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)\langle x_i, x_j\rangle \\ -\varepsilon\sum_{i=1}^{m}(\alpha_i + \alpha_i^*) + \sum_{i=1}^{m} y_i(\alpha_i - \alpha_i^*) \end{cases}$$

$$s.t.\sum_{i=1}^{m}(\alpha_i - \alpha_i^*) = 0;\ \ 0 \le \alpha_i, \alpha_i^* \le C$$

Primal variables: w for each feature dim	**Dual variables: α, α* for each data point**
Complexity: the dim of the input space	Complexity: Number of support vectors



Kernel Trick

- Linear: $\langle x, y \rangle$
- Non-linear: $\langle \varphi(x), \varphi(y) \rangle = K(x, y)$
- Note: No need to compute the mapping function, $\varphi(.)$, explicitly. Instead, we use the kernel function.
- **Commonly used kernels:**

- Polynomial kernels: $K(x, y) = (x^T y + 1)^d$

- Radial basis function (RBF) kernels: $K(x, y) = \exp(-\frac{1}{2\sigma^2} \| x - y \|^2)$

Dual problem for non-linear case

- Primal

$$\min \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m}(\xi_i+\xi_i^*)$$

$$s.t.\begin{cases} y_i-(\mathbf{w}\cdot\varphi(\mathbf{x}_i))-b\le\varepsilon+\xi_i \\ (\mathbf{w}\cdot\varphi(\mathbf{x}_i))+b-y_i\le\varepsilon+\xi_i^* \\ \xi_i\ ,\xi_i^*\ge 0, i=1,\ldots,m \end{cases}$$

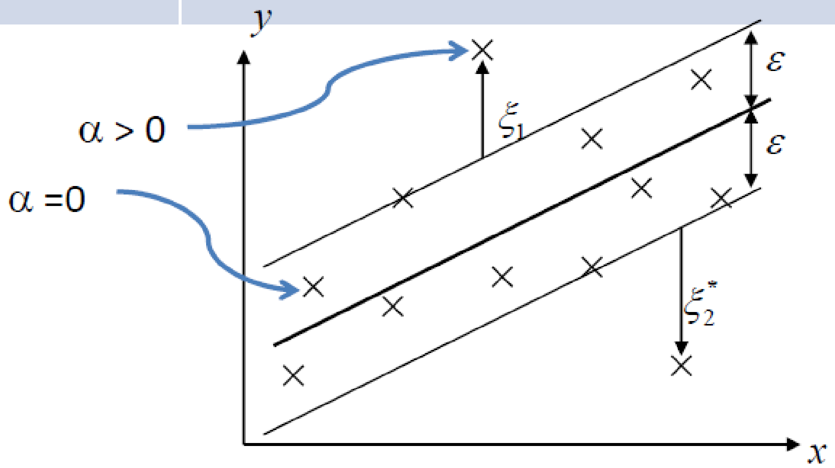
- Dual

K(xi, xj)

$$\max\begin{cases} \frac{1}{2}\sum_{i,j=1}^{m}(\alpha_i-\alpha_i^*)(\alpha_j-\alpha_j^*)\langle\varphi(\mathbf{x}_i),\varphi(\mathbf{x}_j)\rangle \\ -\varepsilon\sum_{i=1}^{m}(\alpha_i+\alpha_i^*)+\sum_{i=1}^{m}y_i(\alpha_i-\alpha_i^*) \end{cases}$$

$$s.t.\sum_{i=1}^{m}(\alpha_i-\alpha_i^*)=0;\ \ 0\le\alpha_i,\alpha_i^*\le C$$

Primal variables: w for each feature dim	Dual variables: α, α* for each data point
Complexity: the dim of the input space	Complexity: Number of support vectors



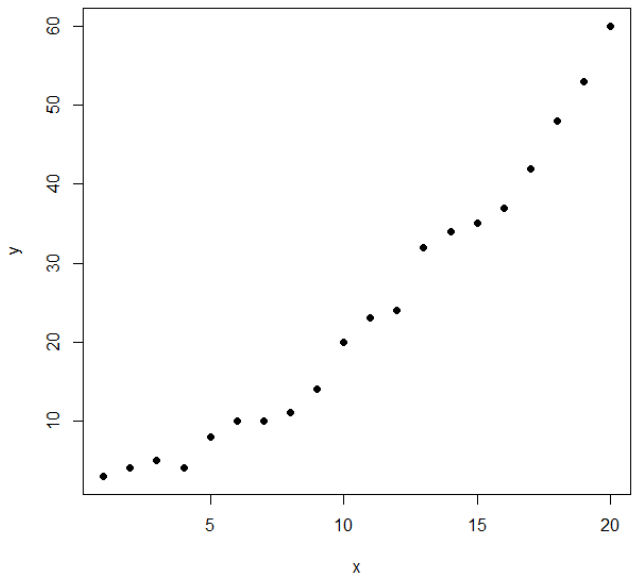
R Application

- To use SVM in R, Let’s create a random data with two features x and y. We took all the values of x as just a sequence from 1 to 20 and the corresponding values of y as derived using the formula y(t)=y(t-1) + r(-1:9) where r(a,b) generates a random integer between a and b. We took y(1) as 3. The following code in R illustrates a set of sample generated values:

```
x=c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)
y=c(3,4,5,4,8,10,10,11,14,20,23,24,32,34,35,37,42,48,53,60)

#Create a data frame of the data
train=data.frame(x,y)
```

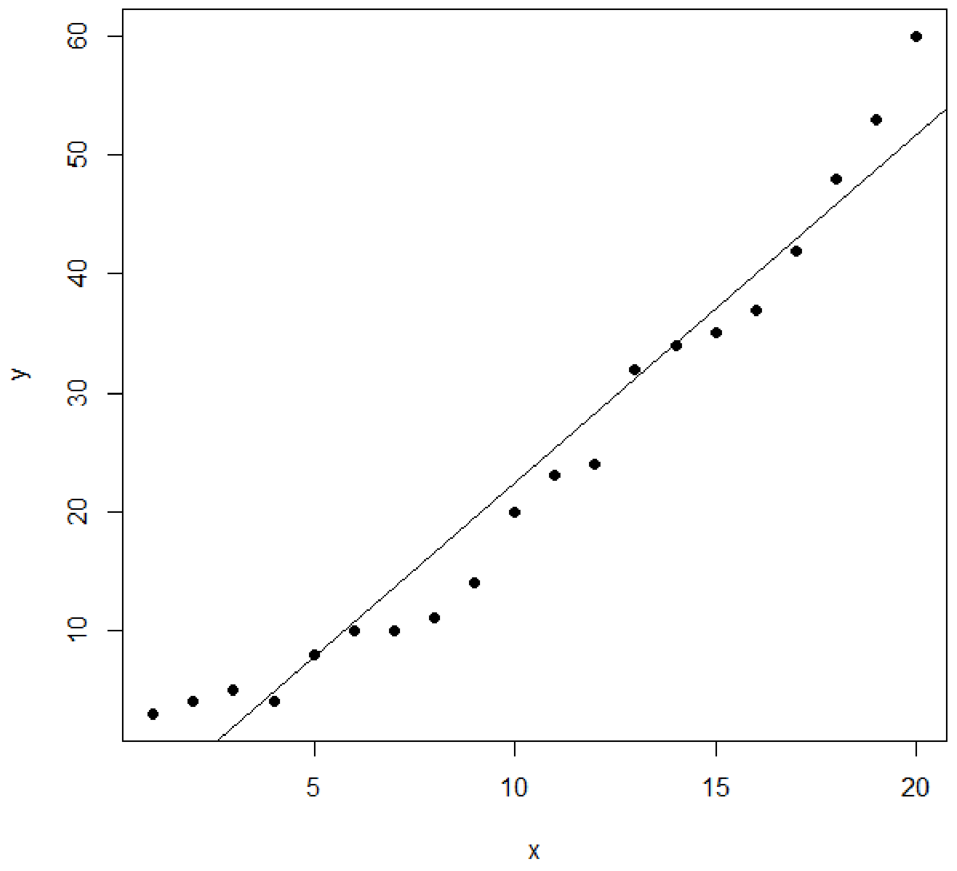
Let’s see how our data looks like. For this we use the plot function.

```
#Plot the dataset
plot(train,pch=16)
```



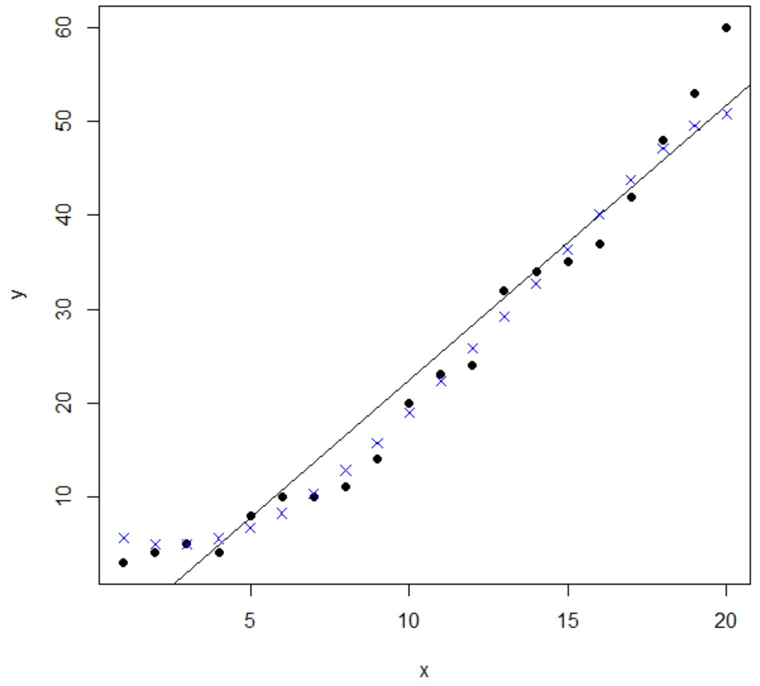
Seems to be a fairly good data. Looking at the plot, it also seems like a linear regression should also be a good fit to the data. We’ll use both the models and compare them. First, the code for linear regression:

```
#Linear regression
 model <- lm(y ~ x, train)
 #Plot the model using abline
 abline(model)
```



To use SVM in R, we have a package e1071.

```
#SVM
 library(e1071)
 #Fit a model. The function syntax is very similar to lm
function
 model_svm <- svm(y ~ x , train)
 #Use the predictions on the data
 pred <- predict(model_svm, train)
 #Plot the predictions and the plot
to see our model fit
 points(train$x, pred, col = "blue", pch=4)
```



- The points follow the actual values much more closely than the abline. Can we verify that? Let’s calculate the rmse errors for both the models:

```
#Linear model has a residuals part which we can extract and
directly calculate rmse

error <- model$residuals

lm_error <- sqrt(mean(error^2))

[1] 3.832974

#For svm, we have to manually calculate the difference
between actual values (train$y) with our predictions (pred)

error_2 <- train$y - pred

svm_error <- sqrt(mean(error_2^2))

[1] 2.696281
```

- In this case, the rmse for linear model is ~3.83 whereas our svm model has a lower error of ~2.7. A straightforward implementation of SVM has an accuracy higher than the linear regression model.
- We can further improve our SVM model and tune it so that the error is even lower. We will now go deeper into the SVM function and the tune function.
- We can specify the values for the cost parameter and epsilon which is 0.1 by default. A simple way is to try for each value of epsilon between 0 and 1 (we will take steps of 0.01) and similarly try for cost function from 4 to 2^9 (we will take exponential steps of 2 here). We am taking 101 values of epsilon and 8 values of cost function. We will thus be testing 808 models and see which ones performs best. The code may take a short while to run all the models and find the best version. The corresponding code will be

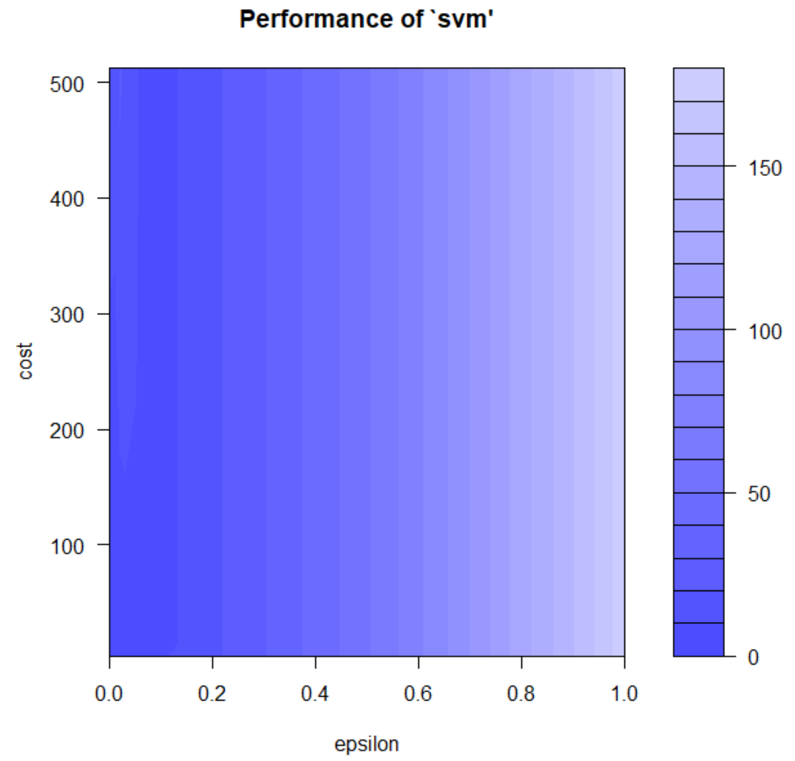
```
svm_tune <- tune(svm, y ~ x, data = train,
            ranges = list(epsilon = seq(0,1,0.01), cost=2^(2:9)))
print(svm_tune)

Parameter tuning of 'svm':

- sampling method: 10-fold cross validation

- best parameters:
 epsilon cost
    0.09  256

- best performance: 2.600606
#This best performance denotes the MSE. The corresponding RMSE is
1.612639 which is square root of MSE
```

An advantage of tuning in R is that it lets us extract the best function directly. We don’t have to do anything and just extract the best function from the svm_tune list. We can now see the improvement in our model by calculating its RMSE error using the following code

```
#The best model
best_mod <- svm_tune$best.model
best_mod_pred <- predict(best_mod, train)

error_best_mod <- train$y - best_mod_pred

# this value can be different on your computer
# because the tune method randomly shuffles the data
best_mod_RMSE <- sqrt(mean(error_best_mod^2))
[1] 1.290738
```

- This tuning method is known as grid search. R runs all various models with all the possible values of epsilon and cost function in the specified range and gives us the model which has the lowest error. We can also plot our tuning model to see the performance of all the models together

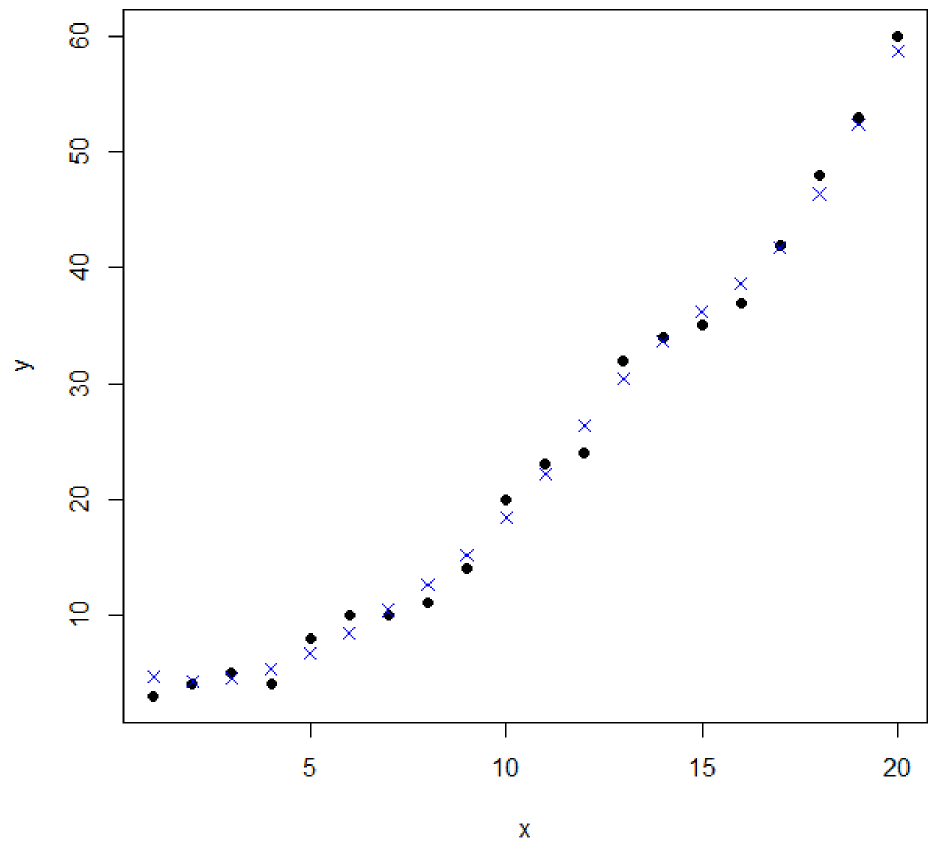
```
plot(svm_tune)
```



This plot shows the performance of various models using color coding. Darker regions imply better accuracy. The use of this plot is to determine the possible range where we can narrow down our search to and try further tuning if required. For instance, this plot shows that we can run tuning for epsilon in the new range of 0 to 0.2 and while we are at it, we can move in even lower steps (say 0.002) but going further may lead to overfitting so we can stop at this point. From this model, we have improved on our initial error of 2.69 and come as close as 1.29 which is about half of our original error in SVM.

- We have come very far in our model accuracy. Let's see how the best model looks like when plotted.

```
plot(train,pch=16)
points(train$x, best_mod_pred, col = "blue", pch=4)
```



y
60
50
40
30
20
10
5
10
15
20
x