Background fighting with robust multivariate techniques

Gagan Mohanty
DHEP, TIFR, Mumbai

The Lecture Outline

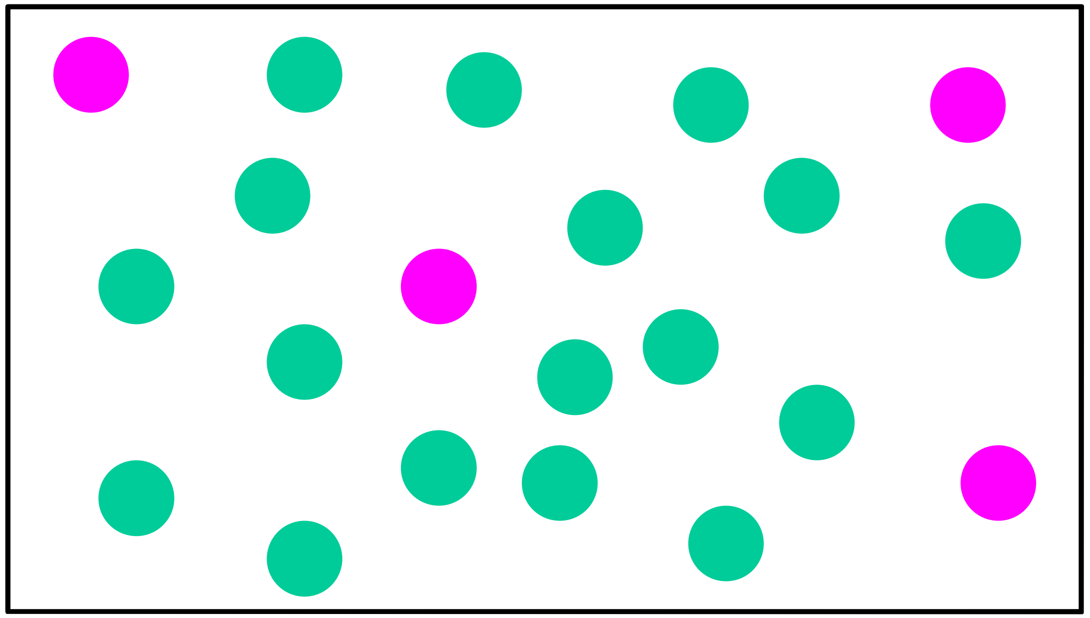
- Background fighting: why and what?

- What do we mean by multivariate analysis technique?
  - ➢ Classic Cut-n-Count Method
  - ➢ Fisher's Linear Discriminant
  - ➢ Multidimensional Likelihood
  - ➢ Artificial Neural Network
  - ➢ Decision Trees (very recent)

➡ You will be provided with the lecture notes

Background fighting – Layman Language

- Almost all the time the number of interesting signal events are very few and overshadowed by huge background

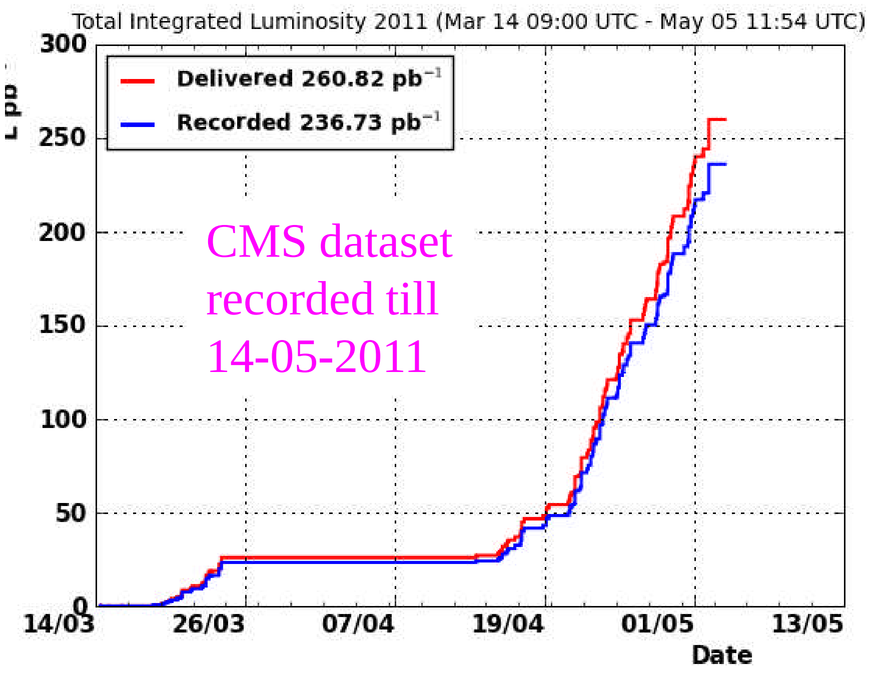


Needle in the haystack

- There is nothing like a 100% signal enriched data ➡ may I dare say, it is difficult to think of an experiment without any background or noise

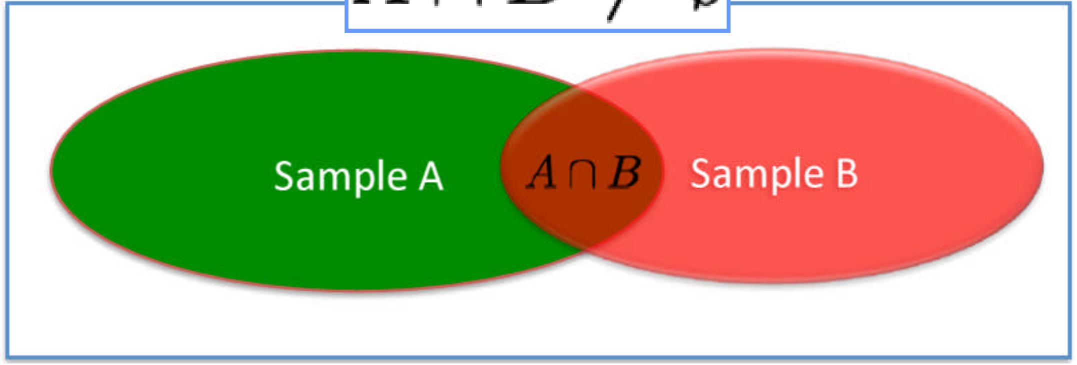
Put it Mathematically

- We start with a data sample of *U* interesting events
  - Each event is described by *n* discriminating variables (or *n* dimensions)
- The data sample contains *m* classes of events: *A*, *B*, … (let's take *m*=2 for simplicity)
- So: $A \subset U$ and $B \subset U$



CMS dataset recorded till 14-05-2011

$$A \cap B \neq \emptyset$$



Right plot has got a profound message, i.e., between *A* and *B* none of the *n* variables is fully nonoverlapping

Closing the Math Chapter

- Consider the event e_i:
  - $e_i \equiv e_i(\mathbf{x}) = e_i(x_1, x_2, x_3, \ldots, x_n)$, ith event of the dataset U
  - How do we determine *A*ness or *B*ness of this event?

- We need some way to assign a probability to the hypothesis that the event e_i is of the class A

$$P(e_i \in A) \leq 1$$

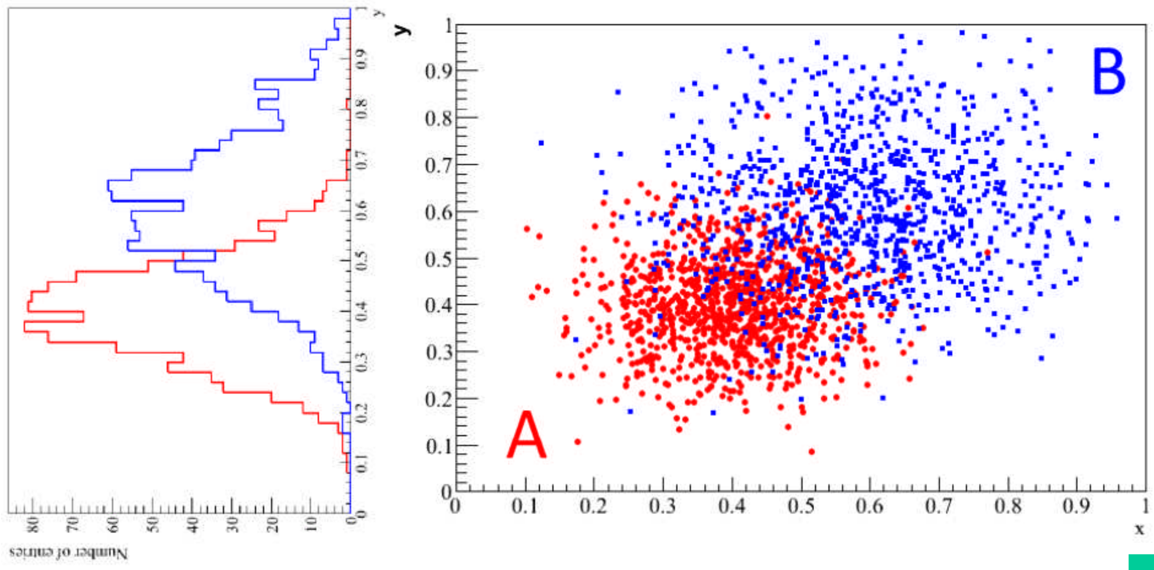
- The compliment of that is the probability that e_i belongs to the class B (a case of two classes)

$$\overline{P(e_i \in A)} = P(e_i \in B) \leq 1$$

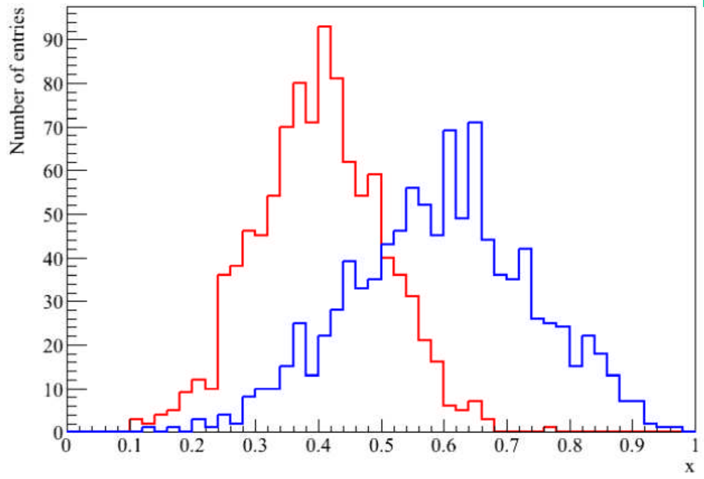
➡ Background fighting is a classification algorithm, where one would like to optimally combine some (or, all) variables $\mathbf{x}$ in order to obtain a best-possible separation between A and B

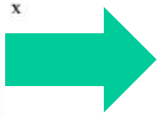
Testing your smartness!!!

Pictorial Sketch of all I have said

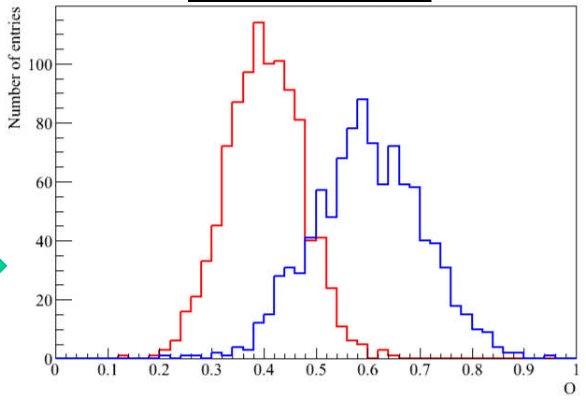


INPUT





OUTPUT

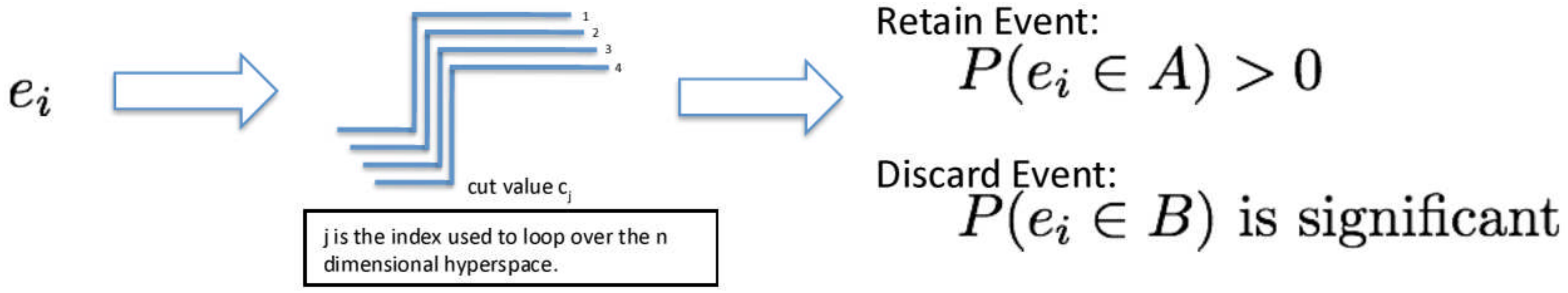


Features of a Good (Optimal) Algorithm

- Separation of classes A and B ➡ reach experimental sensitivity
  - Statistical precision on the measured variables
  - Systematic uncertainties (study some control sample?)
- Understandable and reproducible
  - Can we tell when something has gone wrong?
  - Even better, how easily can we reproduce it?
  - Are there any known pathologies that can impact the problem at hand?
- Ease of use
  - Many readymade toolkits available in the market (TMVA, NeuroBayes, StatPatternRecognition, etc.) ➡ you do not have to code it up yourself!
  - But… that does not mean you need not understand what is going on ➡ It should not be a black-box to you!

Classic Cut-n-Count Method

- Apply an *n*-dimensional step function to an event e_i
  - ➢ Select the event if it satisfies all the criteria
  - ➢ Else, reject it



e_i
1
2
3
4
cut value c_j
j is the index used to loop over the n dimensional hyperspace.
Retain Event:
$P(e_i \in A) > 0$
Discard Event:
$P(e_i \in B)$ is significant

- Determine the cut values using some optimization scheme
  - ➢ Most of the time the signal (statistical) significance is used

$$S = \frac{N_A}{\sqrt{N_A + N_B}}$$

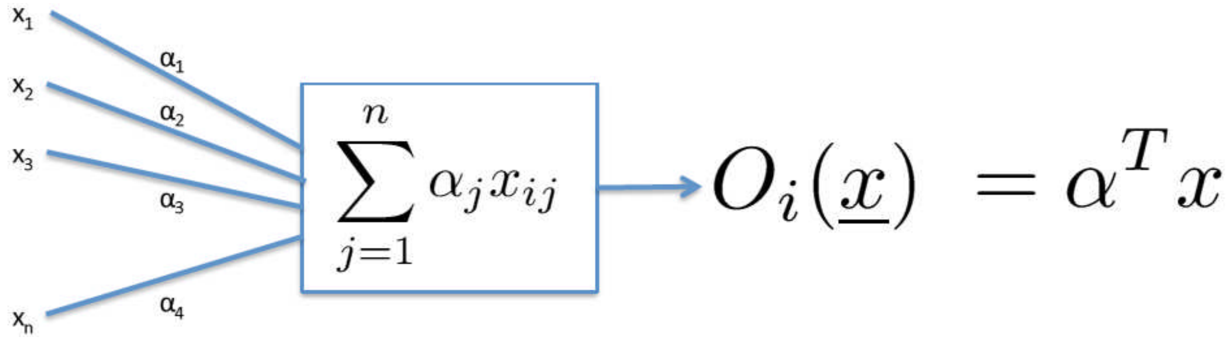
- Pros: easy to use, transparent, systematic well understood
- Cons: Not very efficient ➡ mostly used as a cross-check

Fisher's Linear Discriminant

- Consider the case where we have a data sample *U* which contains one signal class and one background class
  - Our aim is to develop a multivariate discriminating variable $O_i(\mathbf{x})$ for the *i*th event

$$O_i(\underline{x}) = \sum_{j=1}^{n} \alpha_j x_{ij} + \beta$$

  - We will drop β from further discussion as the offset is arbitrary and set for convenience $\Rightarrow$ then pictorially O_i



x_1
x_2
x_3
x_n
α_1
α_2
α_3
α_4
$\sum_{j=1}^{n} \alpha_j x_{ij}$
$O_i(\underline{x}) = \alpha^T x$

  - Again the name of the game is *how to maximize the separation between signal and background* $\Rightarrow$ find out a suitable way to tune the coefficients of **α**

How to Optimize the α’s?

- Before thinking in that direction, let’s see what we know about the data (signal/background)
  - $\mu(\mathbf{x})$, $\sigma(\mathbf{x})$ are known for each variable and for each class
- One can write the Fisher mean and sigma of the corresponding signal and background distributions as:

$$M_{S,B} = \alpha^T \mu_{S,B} \qquad \Sigma^2_{S,B} = \alpha^T \sigma^2_{S,B} \alpha$$

(just the vector sum of the scaled mean and variance using the corresponding weights in $\boldsymbol{\alpha}$)

- To maximize the separation between S and B basically means
  - maximize $|M_S - M_B|$
  - minimize variances $\Sigma^2_{S,B}$
- These requirements can be combined to $\boxed{J(\alpha) = \frac{[M_S - M_B]^2}{\Sigma^2_S + \Sigma^2_B}}$

How to Optimize the α's?

$$\boxed{J(\alpha) = \frac{[M_S - M_B]^2}{\Sigma_S^2 + \Sigma_B^2}}$$

➡ Putting back the respective values of *M* and Σ:

$$[M_S - M_B]^2 = \sum_{i,j=1}^{n} \alpha_i \alpha_j (\mu_S - \mu_B)_i (\mu_S - \mu_B)_j = \alpha^T B \alpha$$

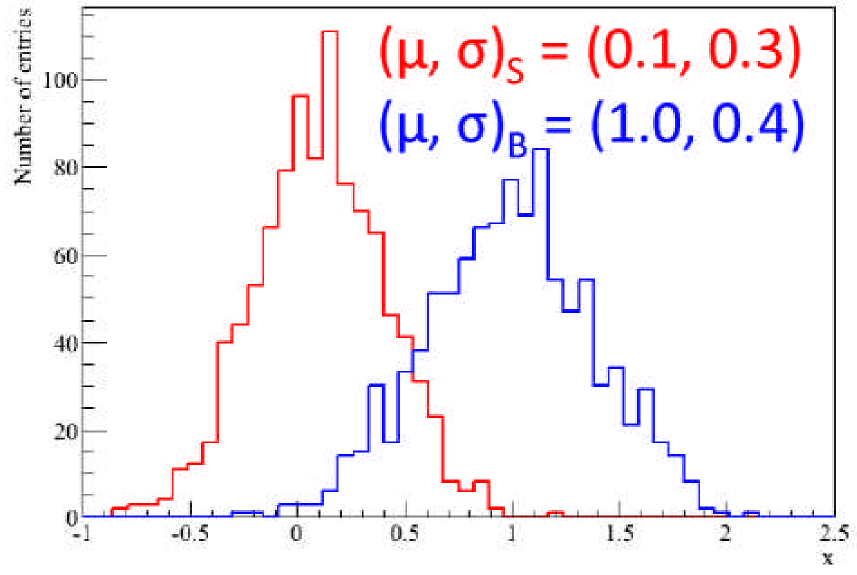
B represents the separation between classes

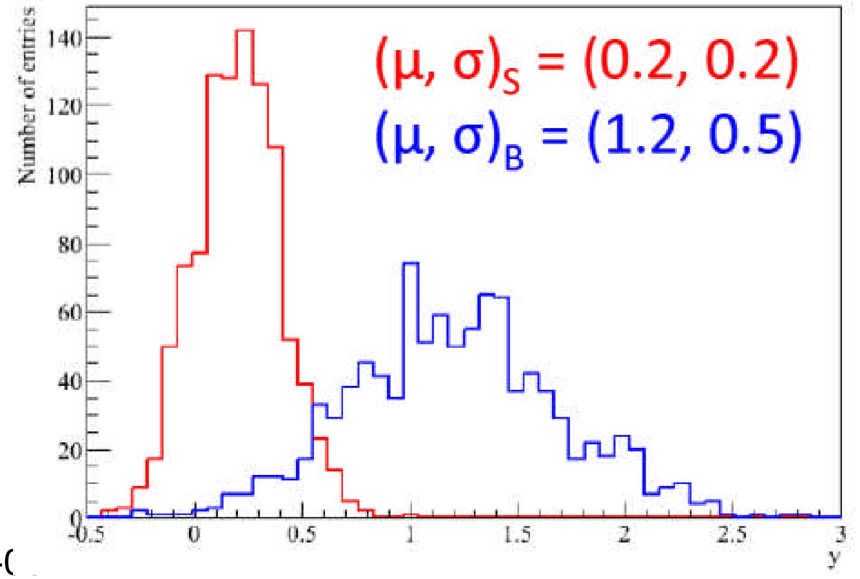
$$\Sigma_S^2 + \Sigma_B^2 = \sum_{i,j=1}^{n} \alpha_i \alpha_j (V_S + V_B)_{ij} = \alpha^T W \alpha$$

W represents the sum of covariances within classes

- Optimal separation between *S* and *B* can be found by solving the top equation for given signal and background samples
- Essentially we need to solve $\dfrac{\partial J(\alpha)}{\partial \alpha_i} = 0$

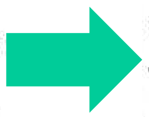
Fisher Discriminant – An Example

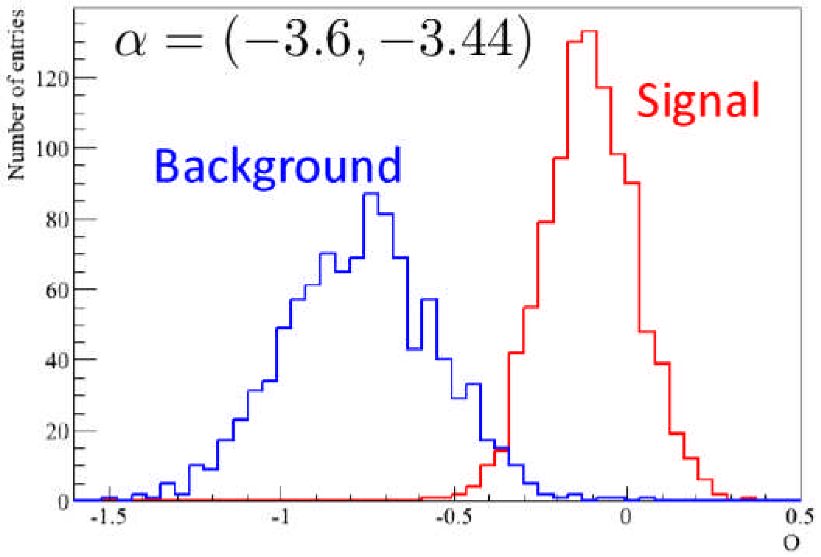




$$(\mu_S - \mu_B) = (-0.9, -1.0)$$

$$W = \begin{pmatrix} 0.25 & 0 \\ 0 & 0.29 \end{pmatrix}$$





Points not to forget about Fisher

- Simpler alternative to the Cut-n-Count method but works pretty well for most of the cases
- Works fine for variables not correlated at all or with linear correlation (see below)

$$O = \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + \ldots + \alpha_n x_n$$

$$\text{Let's say } x_2 \text{ and } x_3 \text{ are linearly correlated}$$

$$\text{which means, } x_2 = m\, x_3 + c$$

$$O = \alpha_1 x_1 + \alpha_2' x_2 + \ldots + \alpha_n x_n$$

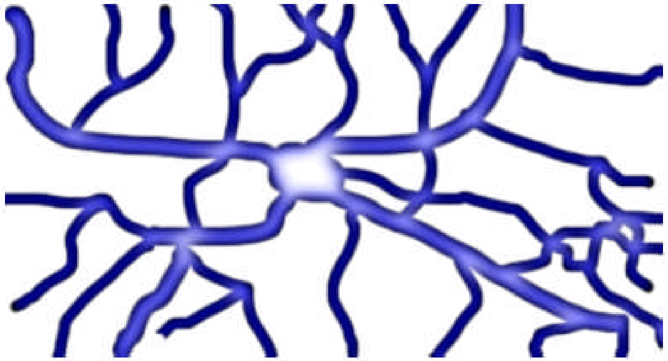
- If a variable is symmetric about the mean value, and both signal and background have a common mean $\Rightarrow$ it will not contribute to Fisher ($\mu_S - \mu_B = 0$)

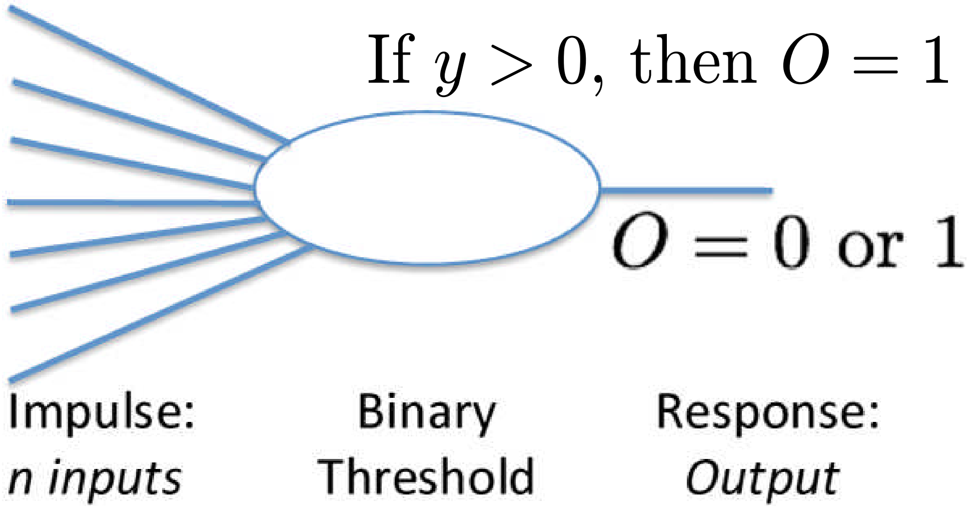
Artificial Neural Network

- Artificial Neural Network (ANN) or simply Neural Network (NN) is a nonlinear algorithm
- Aims to replicate how the human brain works ➡ fundamental building block of a NN is the perceptron ➡ similar to neuron in case of the brain

$$y = f(\underline{w} \cdot \underline{x} + b)$$

If $y > 0$, then $O = 1$



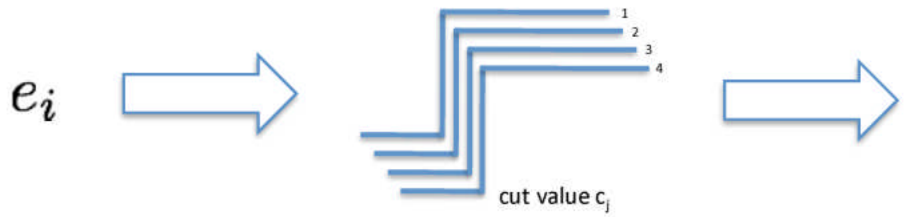


Brain Artificial Neural Network

➢ *y* is the definition of a plane in a *n*-dimensional hyperspace

Basic Unit: Perceptron

- It looks like a familiar concept (recall the logic used in the Cut-n-Count method)

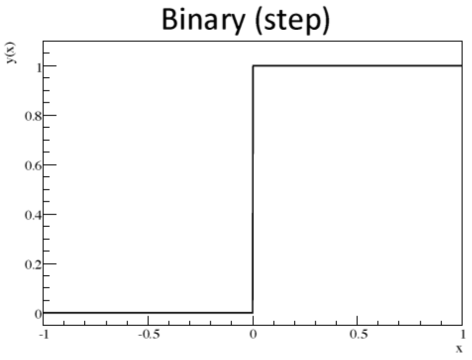


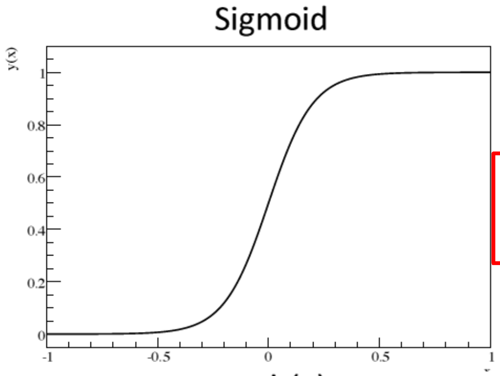
Retain Event:
$$P(e_i \in A) > 0$$
Discard Event:
$$P(e_i \in B) \text{ is significant}$$

- The perceptron is doing pretty much the same thing for given $\underline{w}$ and b, by cutting in the n-dimensional hyperspace
- Recall all the cut information is encoded in c_j in Cut-n-Count
- This tells us an important thing: *A single perceptron won't be of much help than optimally cutting on the data*
- We need to move beyond to the next logical step: Multi-layer Perceptron (MLP)

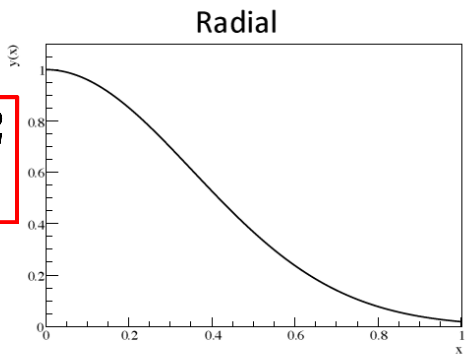
Multi-layer Perceptron – ANN's Heart

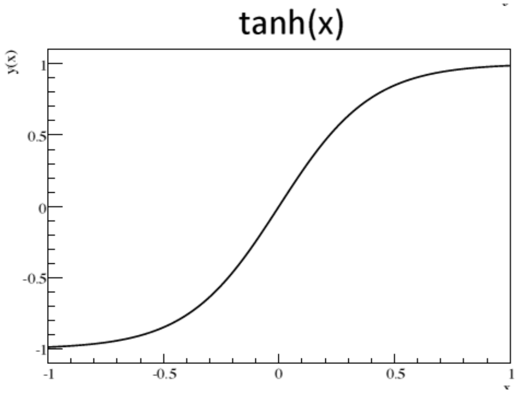
- Combine layers of percetprons in a way so as to obtain a refined separation between classes *A* and *B*
- Modify the output of a perceptron so that it is some function with an output usually between 0 and 1 ➡ activation function
  - Step function can be used
  - Any other suitable function can also be used
  - The sigmoid function is the most popular choice





$$y = \frac{1}{1+e^{\alpha x+\beta}}$$





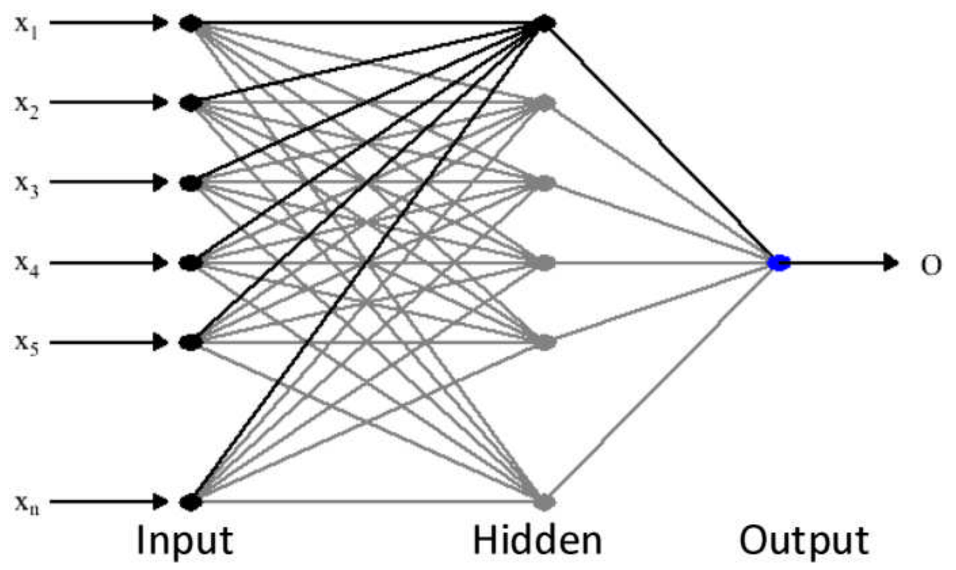
$$y = e^{-\frac{\alpha}{2}x^2}$$

MLP Architecture

- An MLP example with
  - *n* inputs
  - 1 hidden layer with *n* nodes
  - 1 output

Looks good but…

… what are we supposed to do now?



- Decide on the activation function to be used for each node/layer
  - ➡ Let's stick to the sigmoid function
- Determine the weights used to evaluate y_i for each node
  - ➡ Most critical component of ANN
- Check that we have not overtrained our network

How to determine the weights?

$$\boxed{y = f(\underline{w} \cdot \underline{x} + b)}$$

- Start with an initial guess for the weights
  - ➢ Determine how good an estimate this is (use the error in the output classification as a figure-of-merit)
  - ➢ Estimate a new set of weights using the rate of change of errors w.r.t. weights
  - ➢ Re-evaluate the error on the new set of weights
- When the result is *stable* and *good* enough, stop iterating
- At that stage, we have determined the parameters that define the ANN

➡ Is our solution the global minimum?

Global Error Function

- To start with, consider the simple case of a single perceptron
  - Use Class *A* (t=1) and Class *B* (t=0) events from a total data sample *U* as input ➡ supervised learning
  - We want to train our algorithm, so we know the target type t_i for each event e_i
  - Sometimes we can get the classification wrong, which we characterize by an error ε_i:

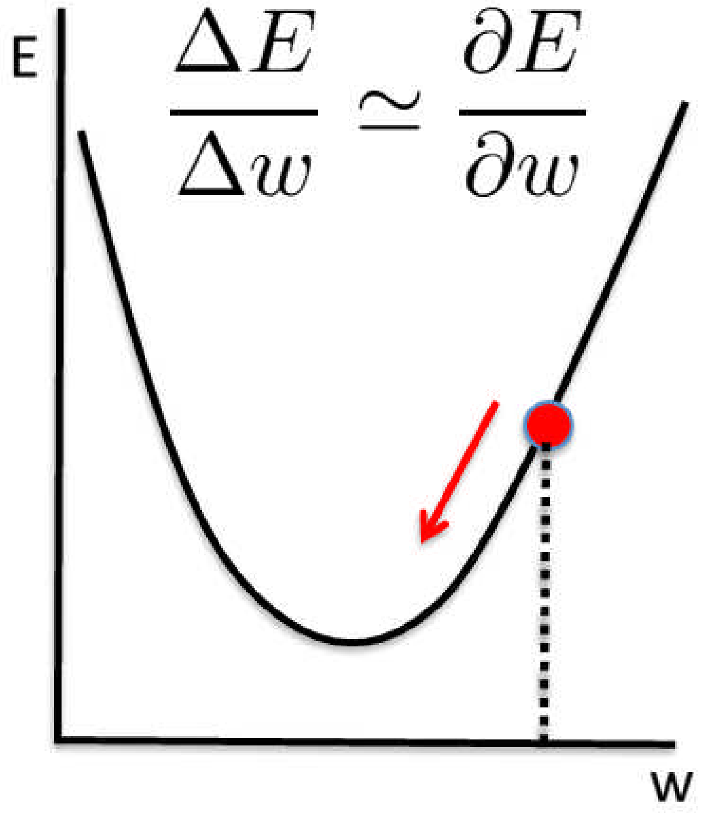
$$\underset{\substack{\nearrow \\ \text{Error}}}{\varepsilon_i} = \frac{1}{2}\left(\underset{\substack{\uparrow \\ \text{Target type}}}{t_i} - \underset{\substack{\nwarrow \\ \text{Perceptron output}}}{y_i}\right)^2$$

- So for the whole data sample *U*, containing *N* events, the total error *E*:

$$E = \sum_{i=1}^{N} \epsilon_i = \sum_{i=1}^{N} \frac{1}{2}(t_i - y_i)^2$$

Minimizing the Error

- Now that we have defined an error, we can:
  - Guess a set of weights
  - Evaluate the total error using those weights
- We need to estimate a new set of weights



$$\frac{\Delta E}{\Delta w} \simeq \frac{\partial E}{\partial w}$$

- Want to try with a new value, which is at a small distance from the initial one

$$w_0 \rightarrow w_0 + \Delta w$$

- At the same time, we wish to move closer to the minimum, so let

$$\Delta w = -\alpha \frac{\partial E}{\partial w}$$

- Here α (learning rate) is a small positive parameter making

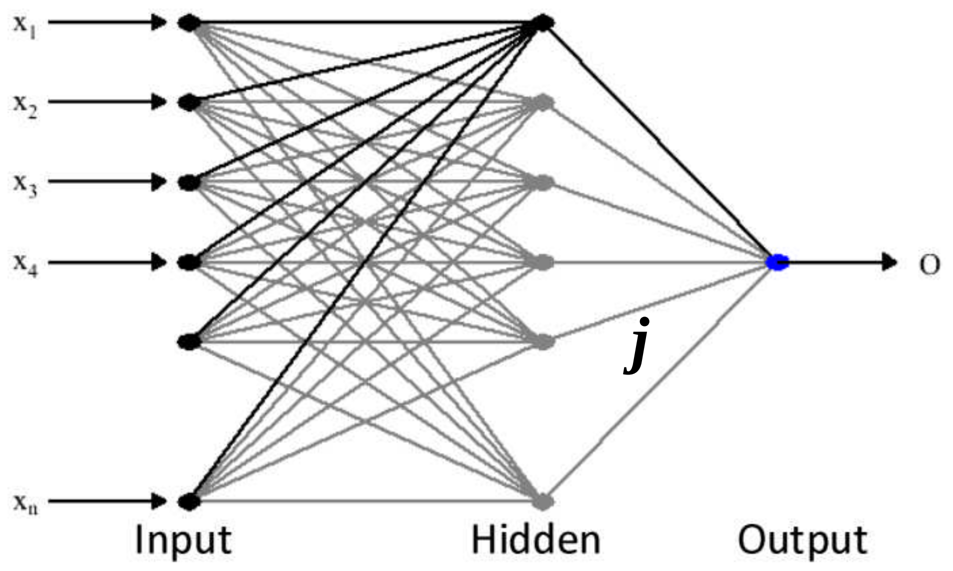
$$\Delta E = -\alpha \left(\frac{\partial E}{\partial w}\right)^2$$

is always positive

➡ This is called gradient descent on an error or delta rule

Complexity of the Problem

- Need to determine weights for each of the nodes, *e.g.*, see the left plot
- A complex job involving many nodes ➡ many weights, akin to a multi-dimensional fit with a lot of free parameters



- We use the method of *Back Propagation* (of error) to accomplish this complicated task

➡See the error contribution of a node *j* to the output layer

$$\Delta t = t_j - y_j$$

➡Check how was the situation at the previous hidden layer

$$\Delta t = \sum_{k\,nodes} (t_j - y_j) w_{jk}$$

Train the MLP (as you want it to perform)

- In order to train the network we need two samples of data (at a time):
  - Sample 1 containing M entries of class A events
  - Sample 2 containing M entries of class B events

You don't have to use equal numbers of events for both classes, however not doing so will affect the convergence of your network. You are advised to keep to using equal number of events in samples A and B.

- How do we know when training has finished?

➡ Compare the error and its gradient against some anticipated threshold

➡ Validate the performance against some test sample

Need for a Validation Sample

- Provides a statistically independent reference point
  - ➢ If the training and the validation (or, testing) samples perform the same with a set of weights, then we can have faith in the MLP configuration when applying it to the real data
- Solution should be more robust than using all data for training
  - ➢ Possibility of overtraining can be easily traced out
- How much data we keep aside for testing?

➡ Experience says approximately similar size as the training sample is a good choice

- How much data in total (training + testing) we need for an MLP?

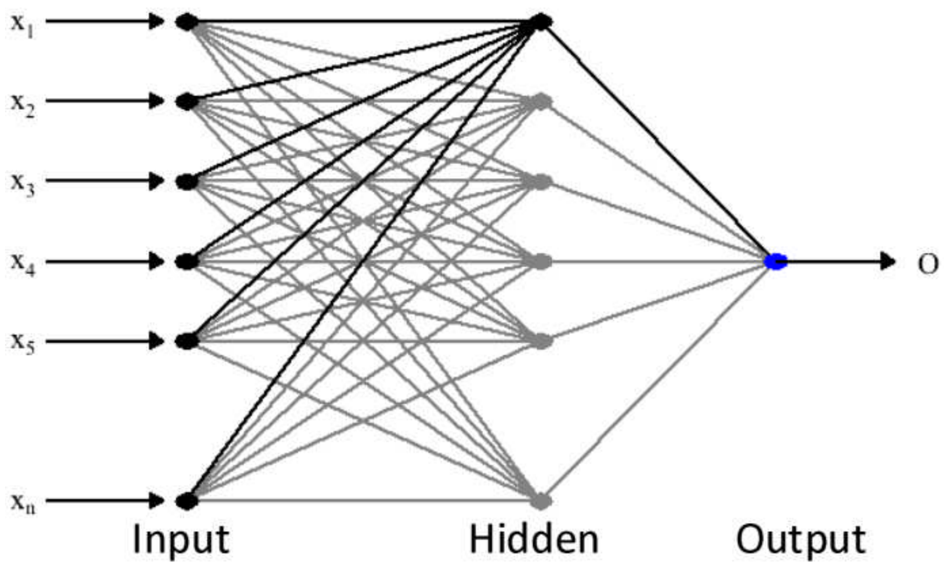
$$M > O\left(\frac{W}{\varepsilon}\right)$$

M = data size for training
W = total number of weights
ε = error threshold
N = number of nodes

➡ For more complex network $M > O\left[\frac{W}{\varepsilon}\log(N/\varepsilon)\right]$

How much data one needs for an MLP?

- Example:
  - Below is an MLP with 1 hidden layer of 6 nodes, there are 6 input variables and 1 output node
  - Total number of weight parameters, W = (6+1)×6 = 42
  - Say, the misclassification (error) level we want to achieve is 1%



➡ The training data size should be at least 42/0.01 = 4200, so the total data should be twice of that, which is 8400

Points not to forget about NN

- Just don’t assume MLP is a magic tool for you
  - ➢ Need to check if your problem is that complex which a simpler algorithm cannot suitably handle
  - ➢ Check all the input variables you want to use as inputs to MLP ➡ don’t add any unuseful variable ➡ adding noise to your network
  - ➢ What sort of correlations exist between various input variables, e.g., if there are no nonlinear correlations, a Fisher discriminant may be sufficient
- If all of the above makes you think that MLP is the way to go about, you should
  - ➢ Ensure there is a large data samples at your avail
  - ➢ Go step by step ➡ don’t put everything together at a time ➡ judiciously add the input variables in increment ➡ check if the improvement in performance makes any sense
  - ➢ Pay enough attention to the validation step ➡ it would be even better to test out the performance with some control data sample in addition

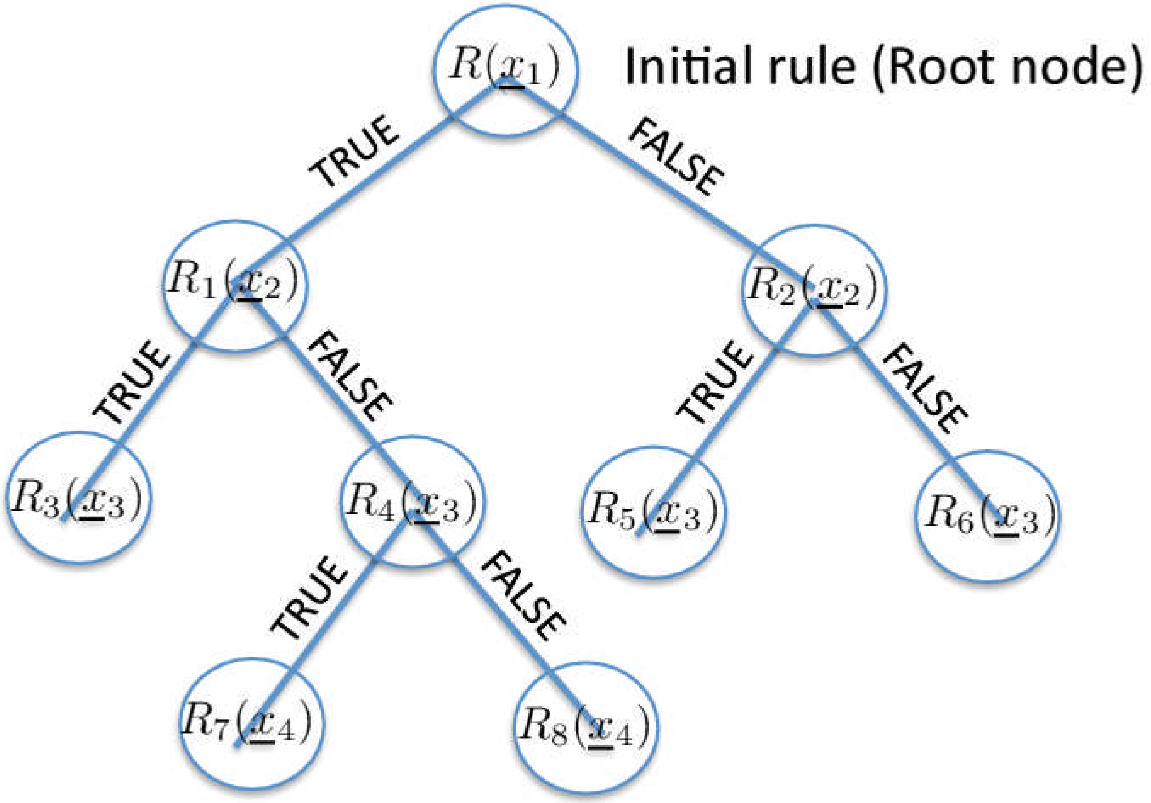
Bottom-line: *Think twice before you plunge*

What is a (Binary) Decision Tree?

- Apply the initial rule to all data:
  - ➢ Divide them into two classes with a binary output

$$R(\underline{x}_1) = \underline{x} > \underline{x}_i \text{ TRUE}$$
$$= \underline{x} < \underline{x}_i \text{ FALSE}$$

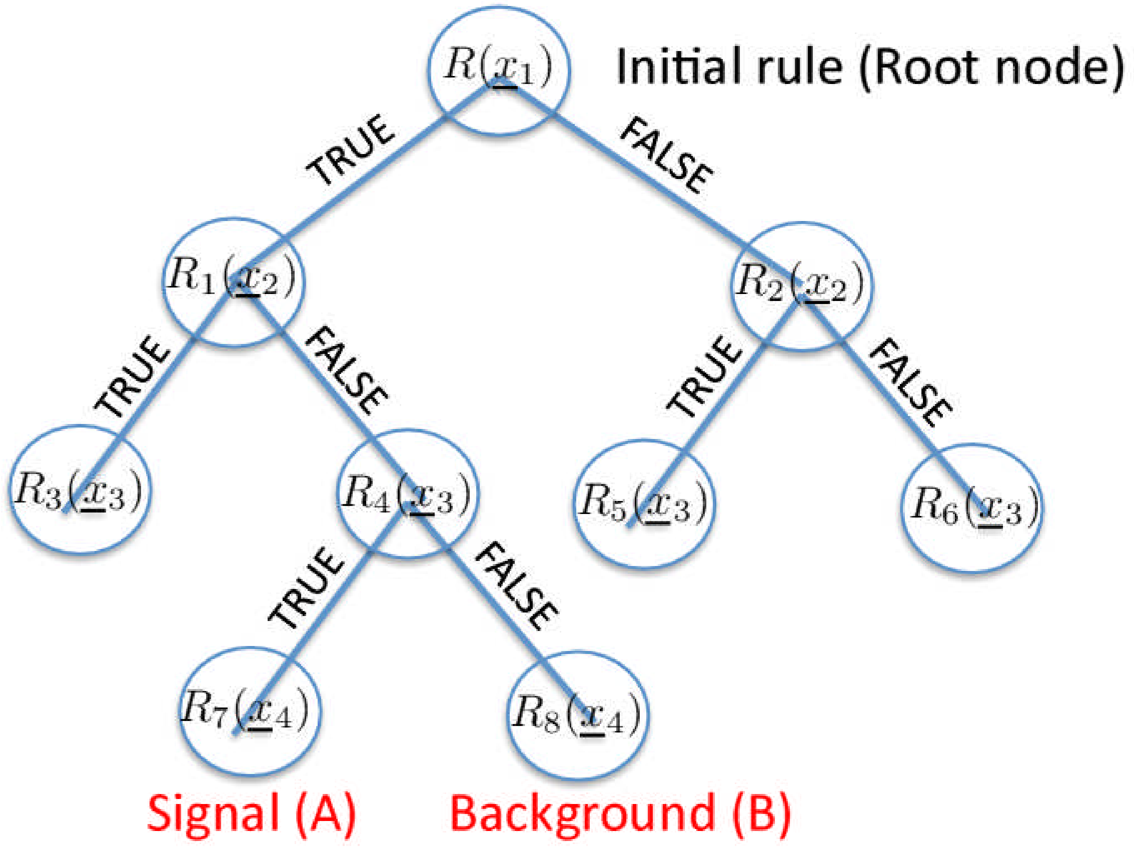
  - ➢ At each successive layer, we divide the data further into signal (class *A*) or background (class *B*)



- The classification for a set of cut values will have an error
  - ➢ Just as with a NN, one can vary the cut values in order to minimize the error ➡ train the decision tree

What is really happening?

- Each node uses a subset of input variables that give the best separation between two classes

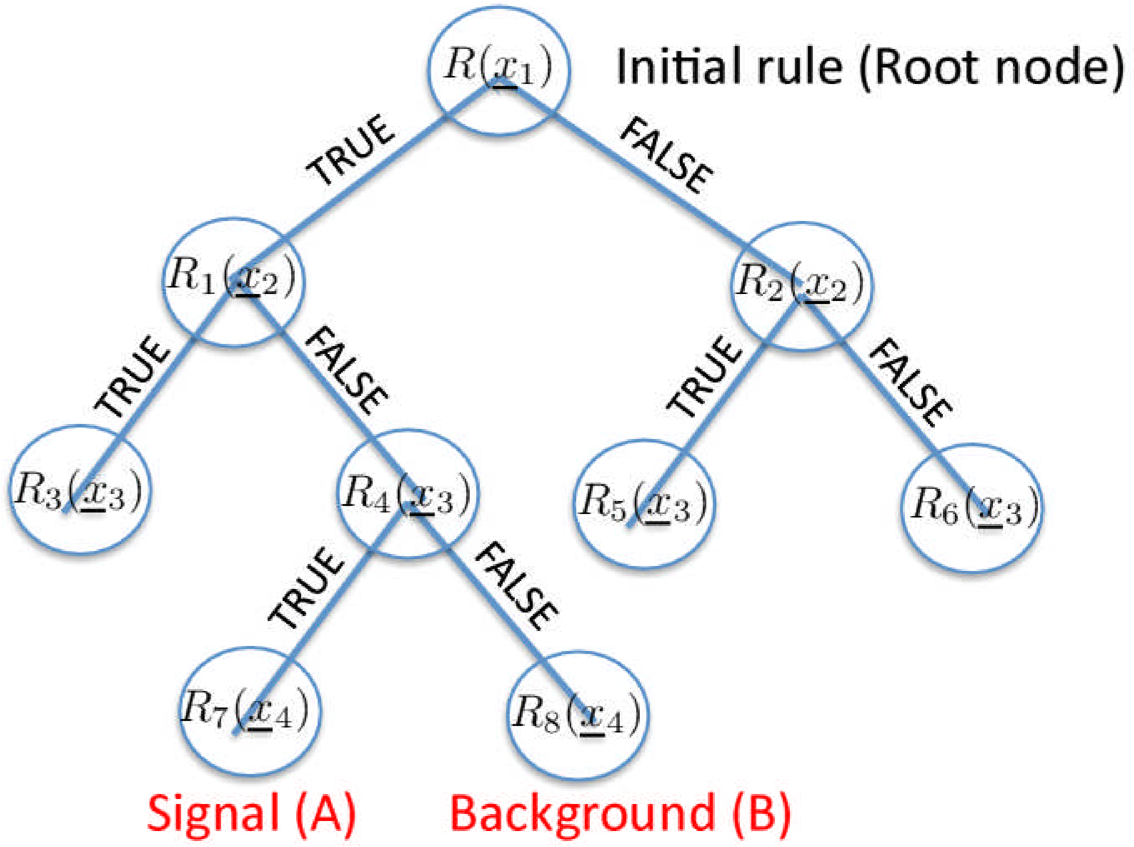


➡ Some variables may be used more often by more than one node

➡ Other (noise) variables may never be used

Looking little bit closer

- Bottom of the tree just looks like a sub-sample of events subjected to a cut based analysis



➡ There are many bottom levels to the tree

➡ …many signal/background regions defined by the algorithm

Is it a carbon-copy of C-n-C?

- If the binary decision tree just mimicks Cut-n-Count, why to bother about it?

- Pros:
  - ➢ More flexibility in the algorithm when trying to separate two classes of events ➡ able to obtain a better separation than a simple cut-based approach
  - ➢ Easy to understand and interpret in contrast to a NN

- Cons:
  - ➢ Potential instability with respect to statistical fluctuations in the training sample

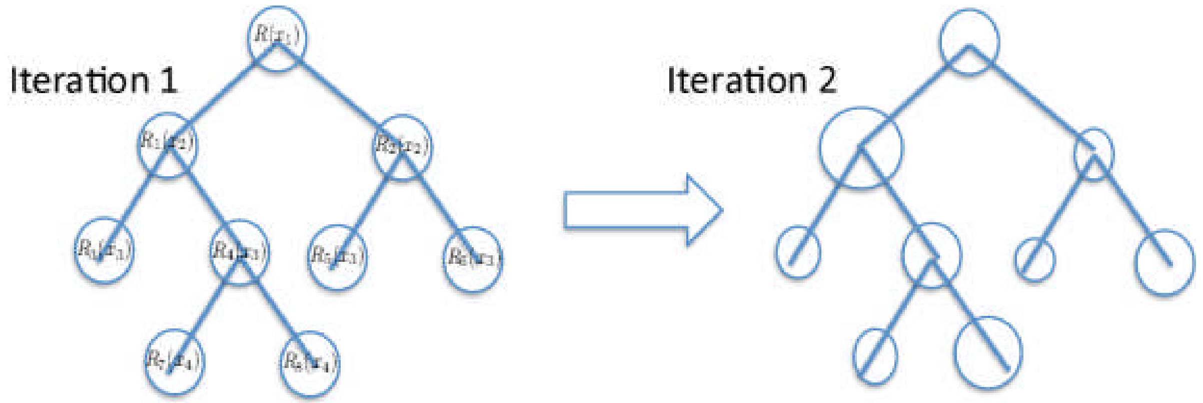
➡ Could you think beyond the binary version of the tree?

Boosted Decision Trees

- At each stage in training there may be some misclassification of events (error rate) ➡ let’s try to minimize that
  - Assign a greater event weight α to misclassified events in the next iteration

$$\alpha = \frac{1-\varepsilon}{\varepsilon}, \text{ where } \varepsilon \text{ is the error rate}$$

  - Reweight the whole sample so that the sum of event weights remains the same, continue to iterate until the tree is stabilized



➡ By reweighting misclassified events by α the aim is to reduce the error rate of the trained tree

Let’s now take a tour of TMVA