Boosted Decision Trees

Arun Nayak
(Institute of Physics, Bhubaneswar)

Introduction

- Supervised learning is a method where we use the labeled training data (with multiple features) x_i to predict a target variable y_i.

- Model & Parameters:
  - Model ➔ mathematical structure by which the prediction y_i is being made from the input $\mathbf{x}_i$, i.e. $y_i = f(x_i)$
    - e.g. the liner model: $y_i = \Sigma w_{ij} x_j$
    - w_{ij} are the undetermined, to be learnt from data
  - The outcome/prediction (y) can be
    - Quantitative (Regression), e.g. mass distributions, or
    - Qualitative/categorical (classification), e.g. signal & background

Decision Trees

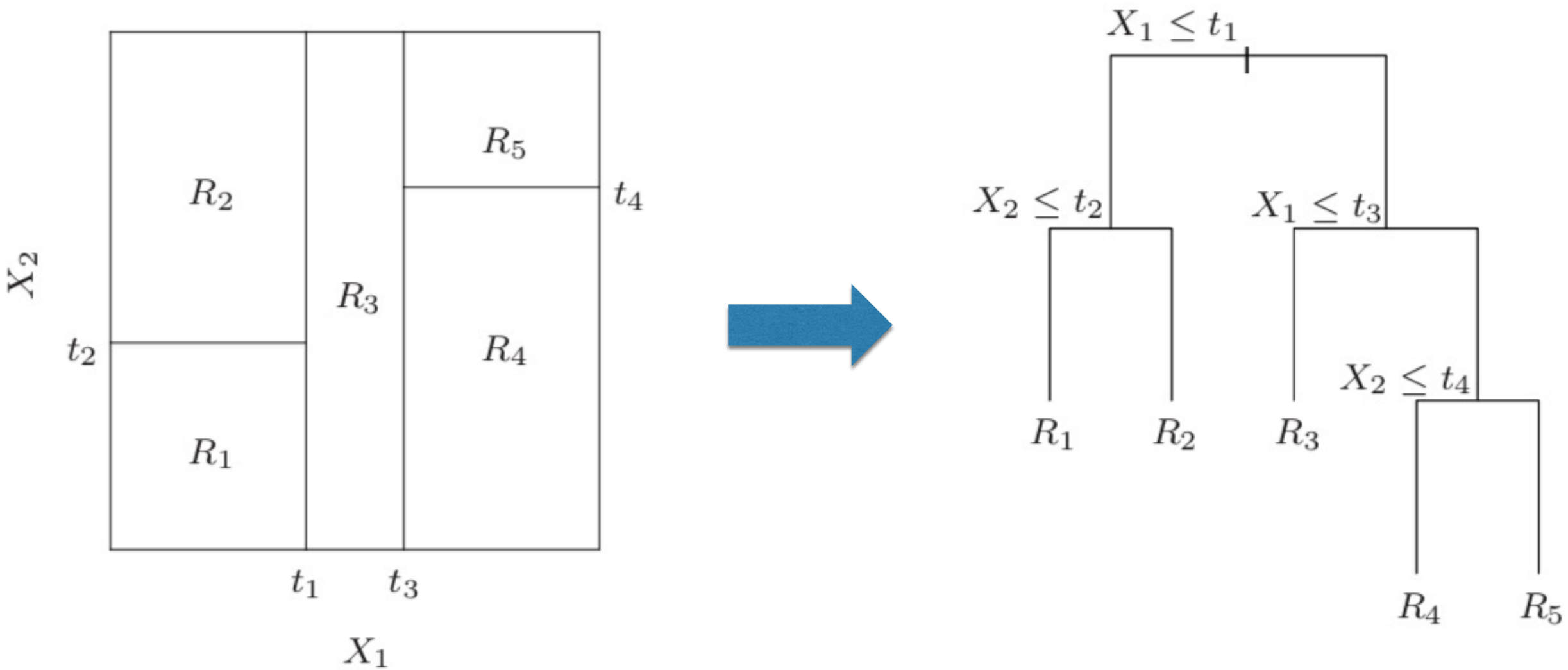
- Decision Trees are a popular (supervised) machine learning technique, commonly used in high energy physics data analysis
  - First developed and formalised by Breiman et al.
    - Proposed the CART algorithm (Classification And Regression Trees)
  - Became very popular in HEP after use in MiniBooNE experiment (arXiv: 0408124)
  - Widely used in Run-1 and Run-2 physics analyses at LHC

- Basic concept:

  Extend the cut-based analysis by not rejecting events that fail a particular criterion, instead, check if other criteria may help to classify these events correctly

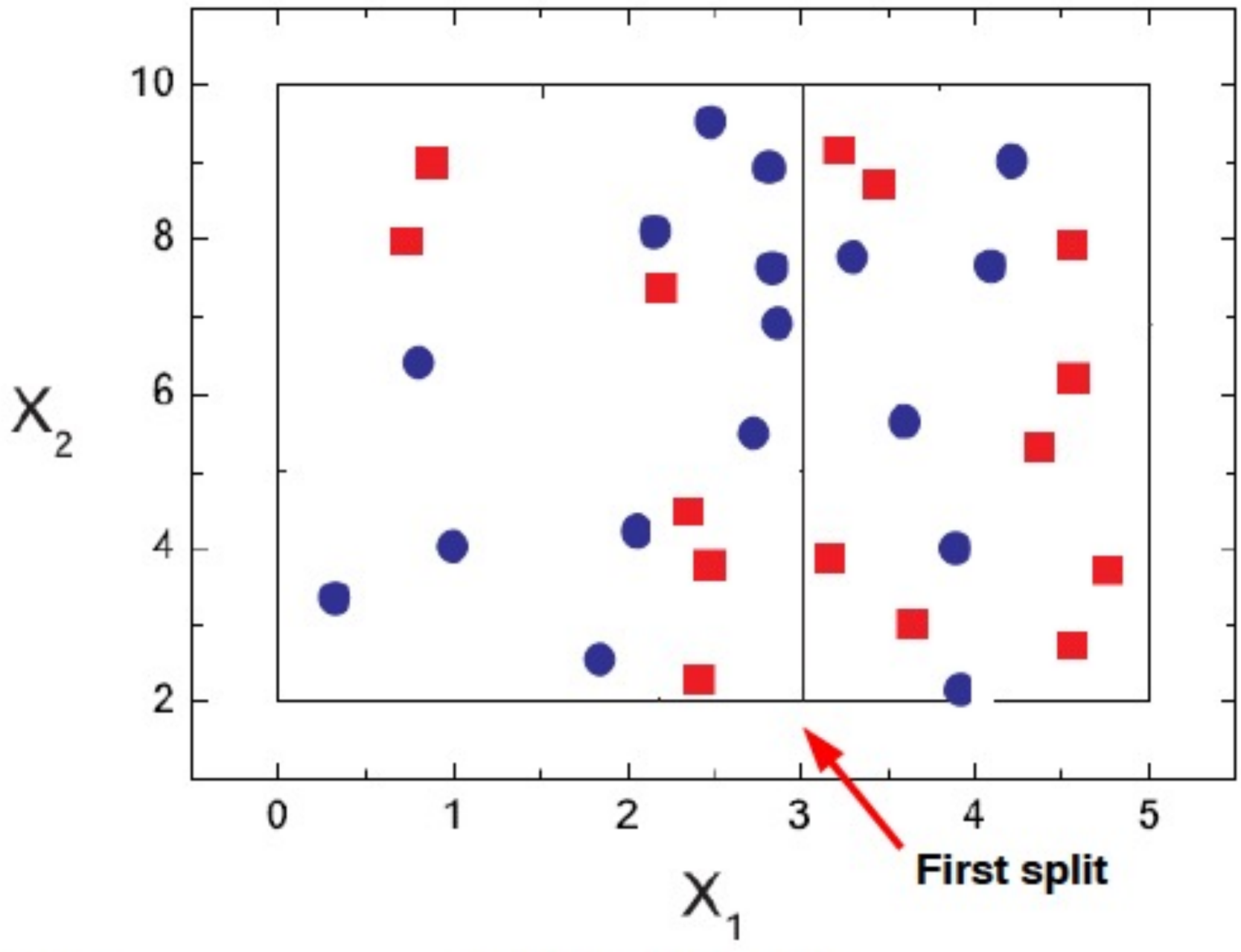
Binary Tree

Perform recursive binary partitions of the feature space into a set of rectangles

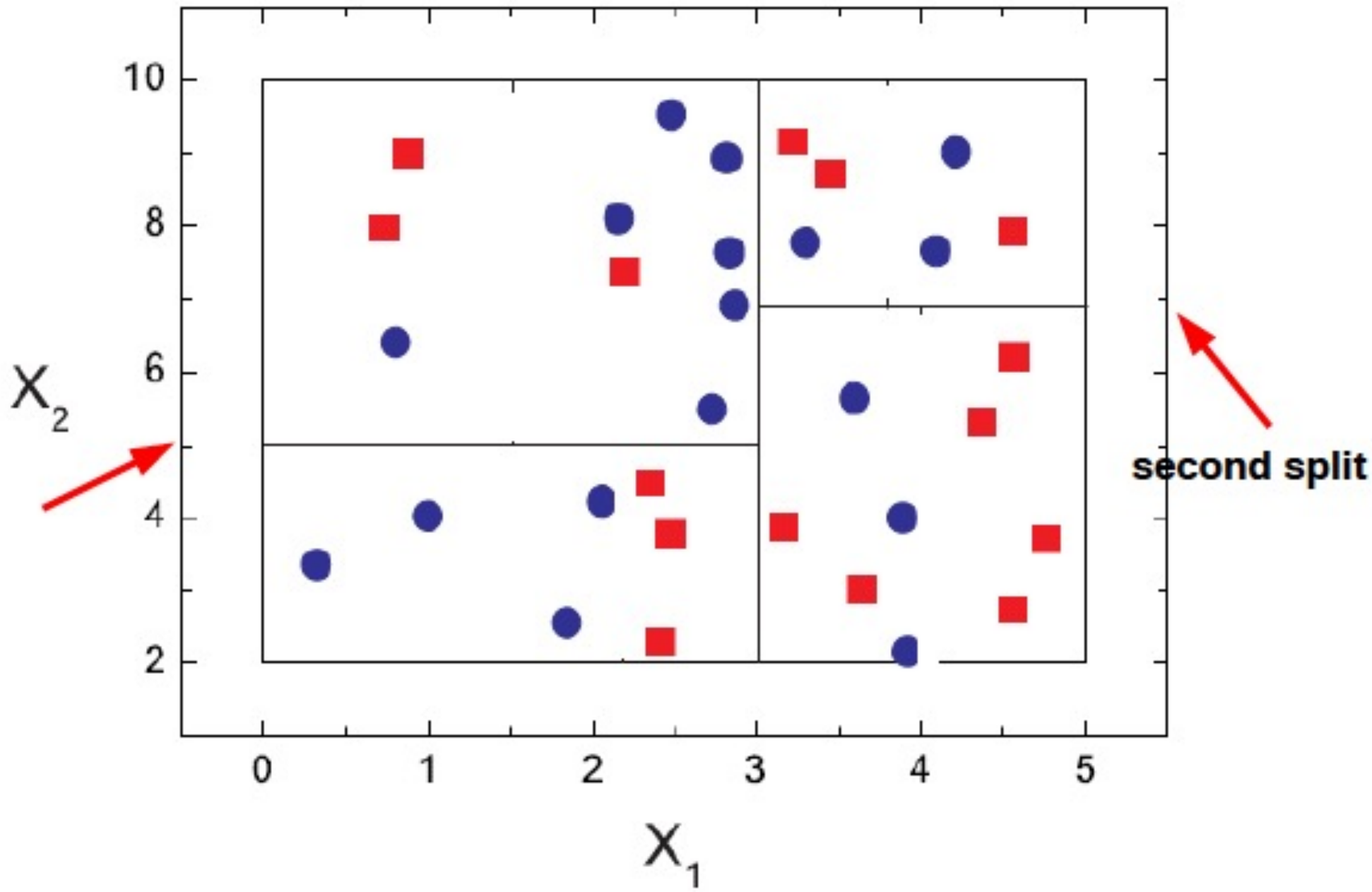


Ref: Figure 9.2,T. Hastie et al.

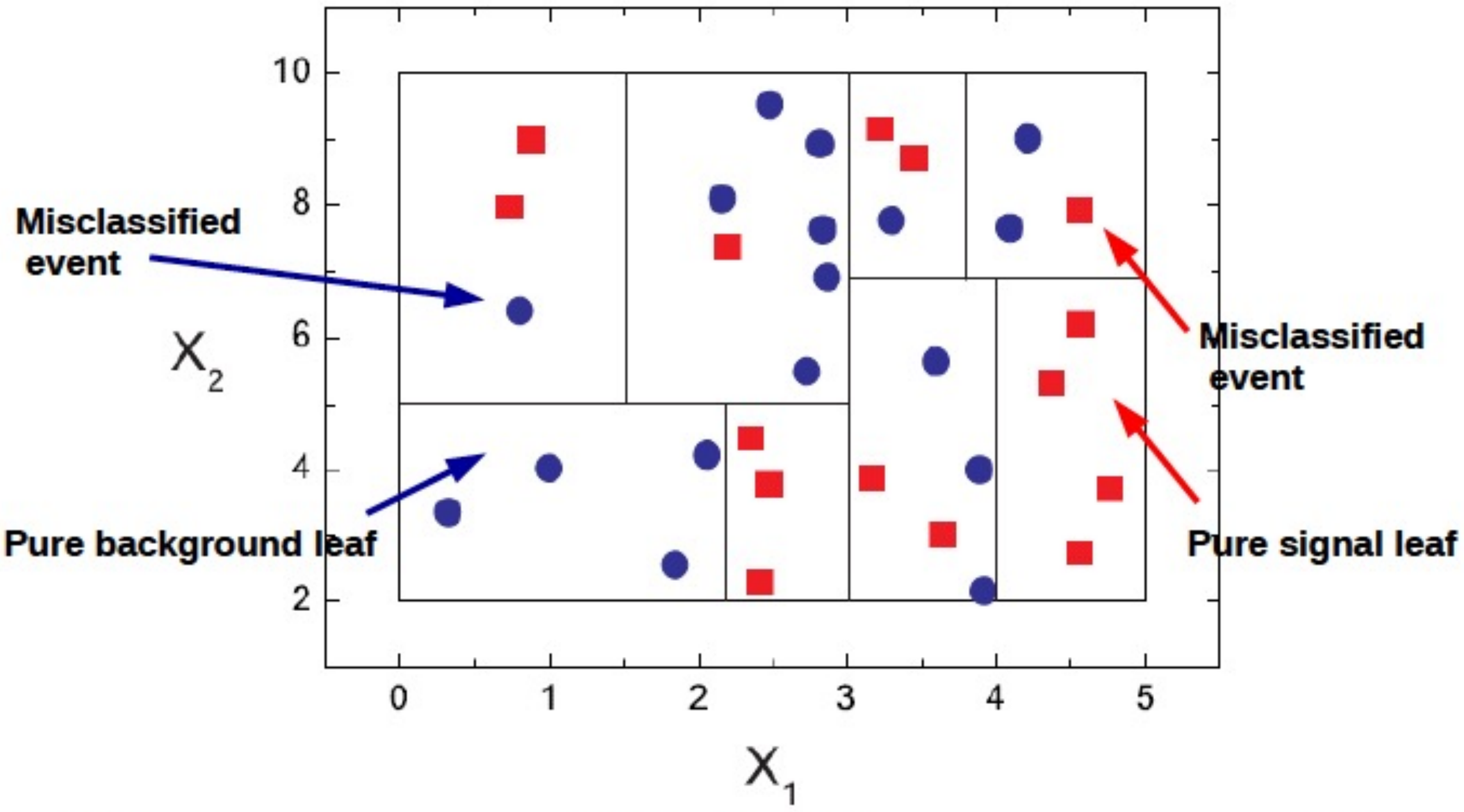
Binary splitting



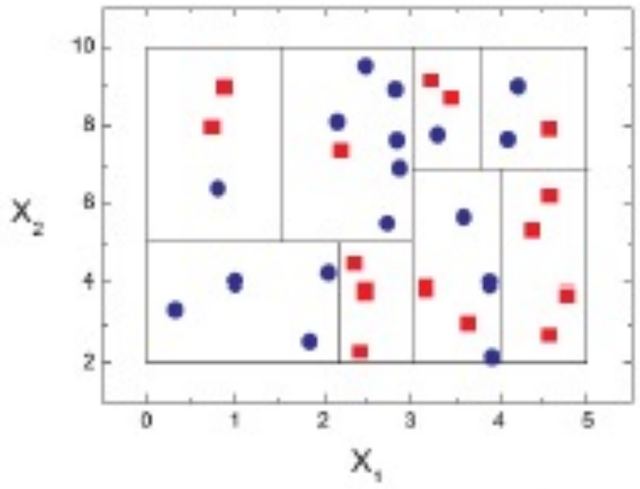
Binary splitting

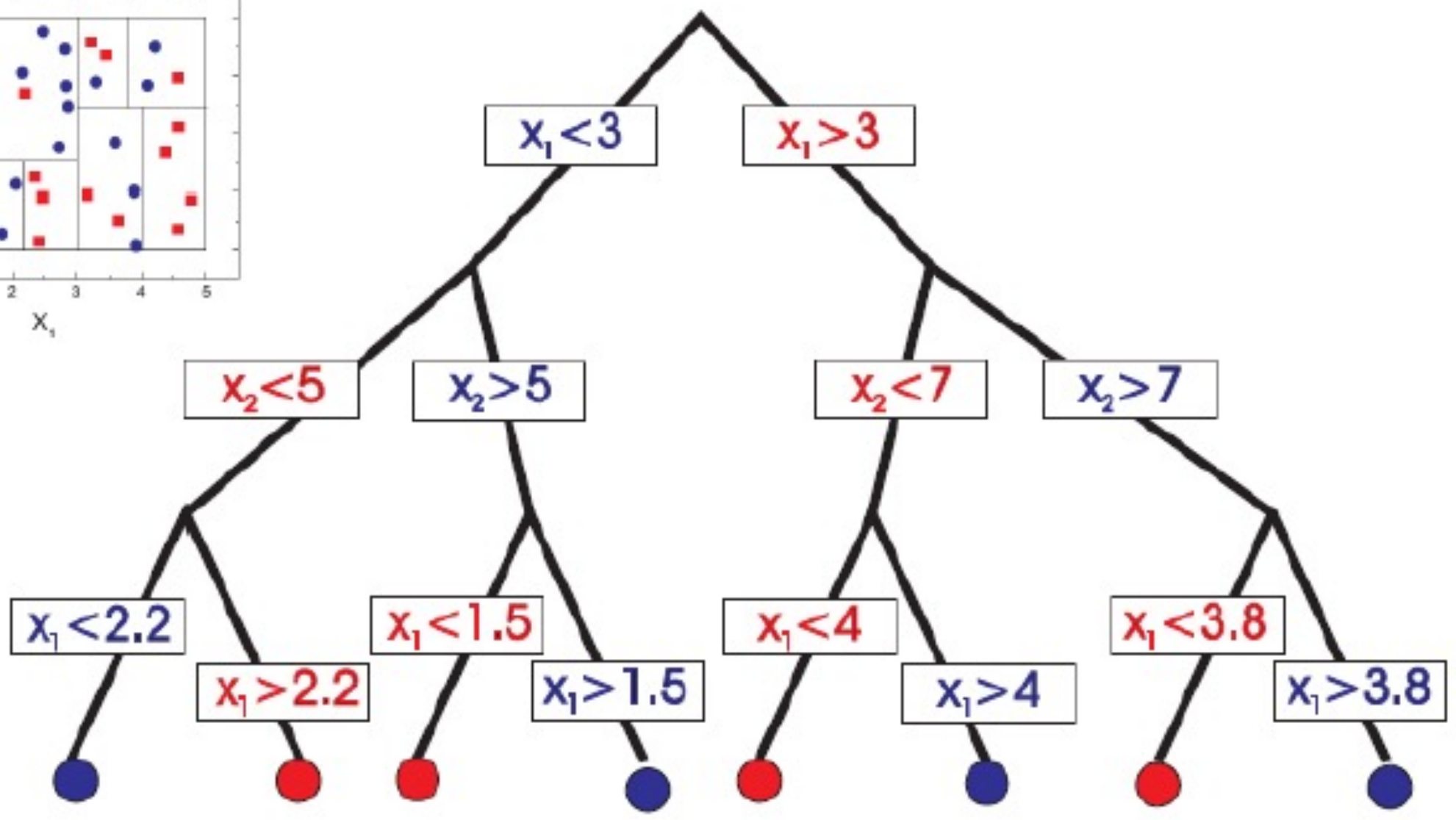


Binary splitting



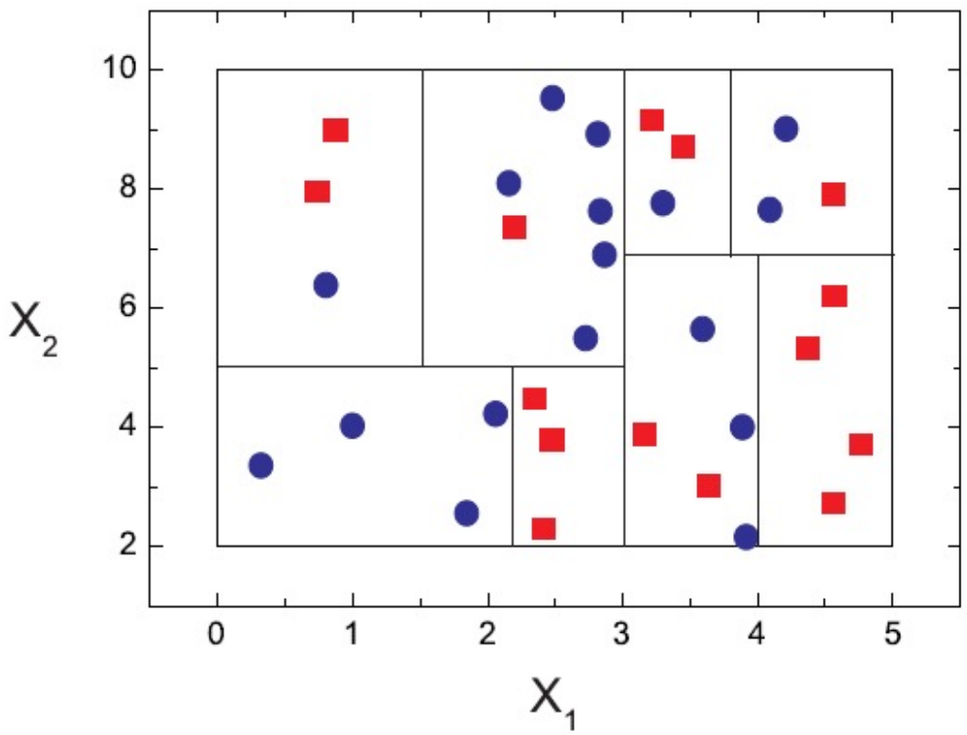
Tree diagram for binary split

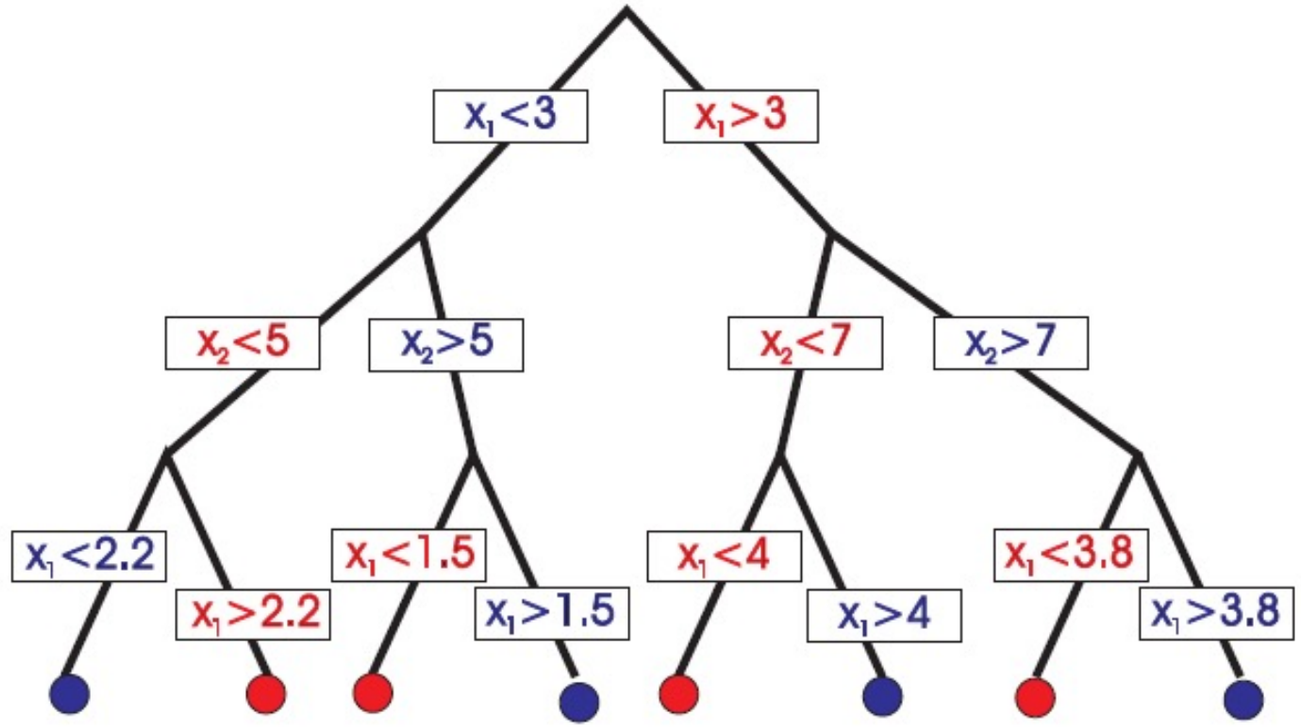




Binary Tree

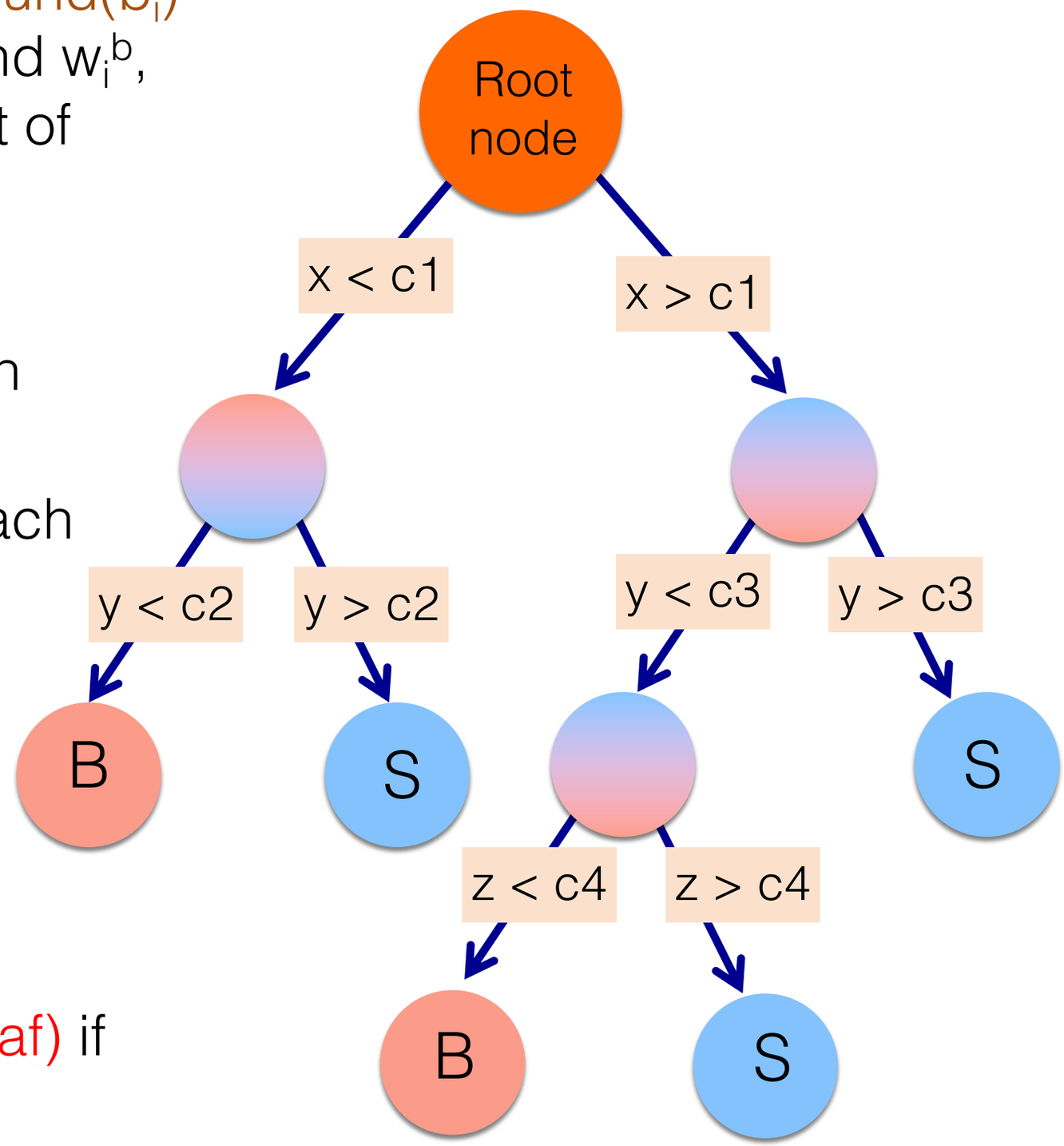
- A key advantage of the recursive binary tree is its interpretability.
  - The feature space partition is fully described by a single tree.
  - With more than two inputs, partitions like the left one are difficult to draw, but the binary tree representation works in the same way





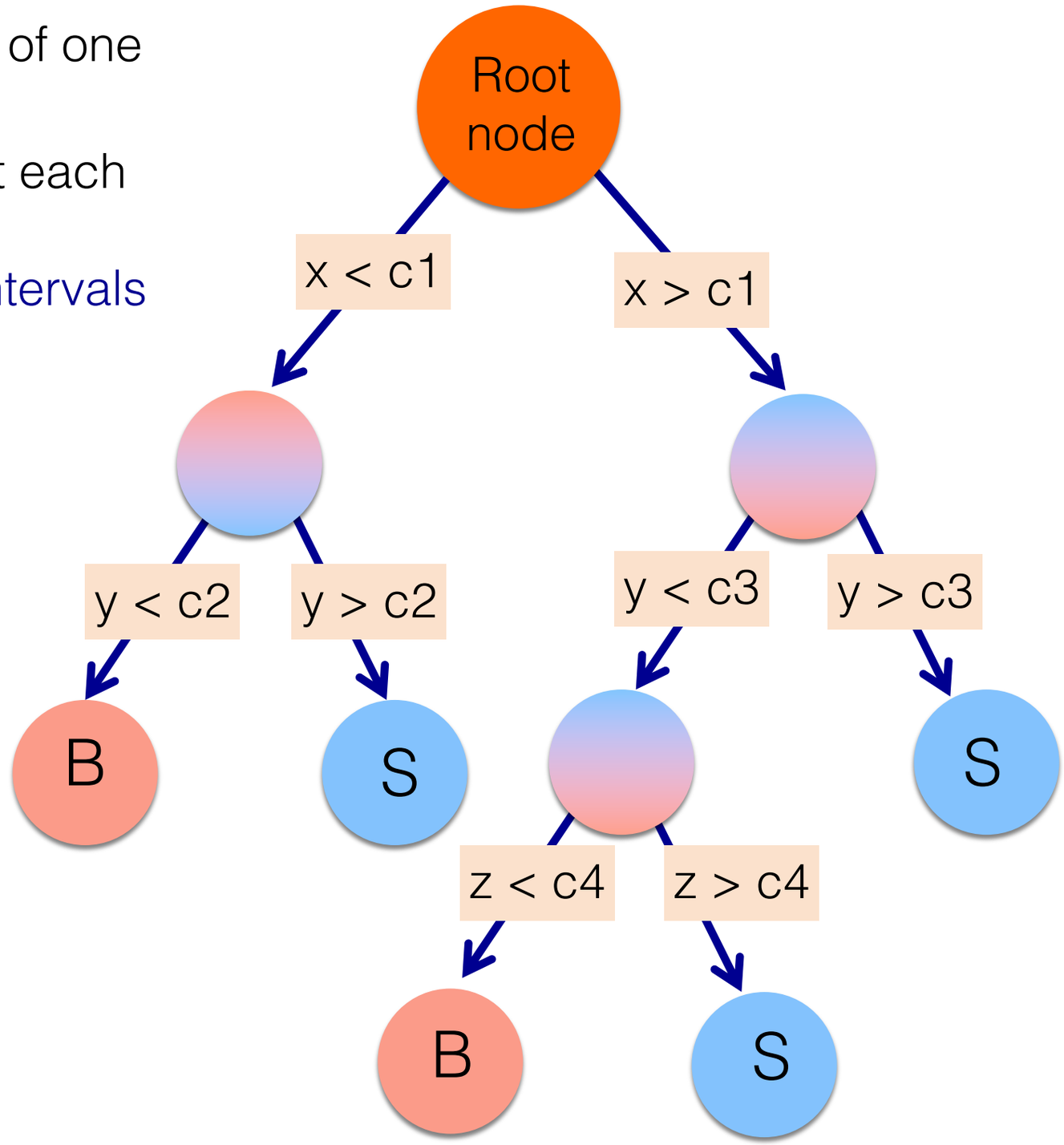
Growing a Decision Tree

- Sample: signal (s_i) and background(b_i) events, each with weights w_i^s and w_i^b, respectively, described by a set of variables (x_j) (features)
- Sample ←→ root node
- Sort all events according to each variable x_j
- Find best splitting position for each variable
- Select the variable and the splitting position that gives best separation
- Split the node to two new nodes (branches)
- Declare the node as terminal (leaf) if it satisfies stopping criterion



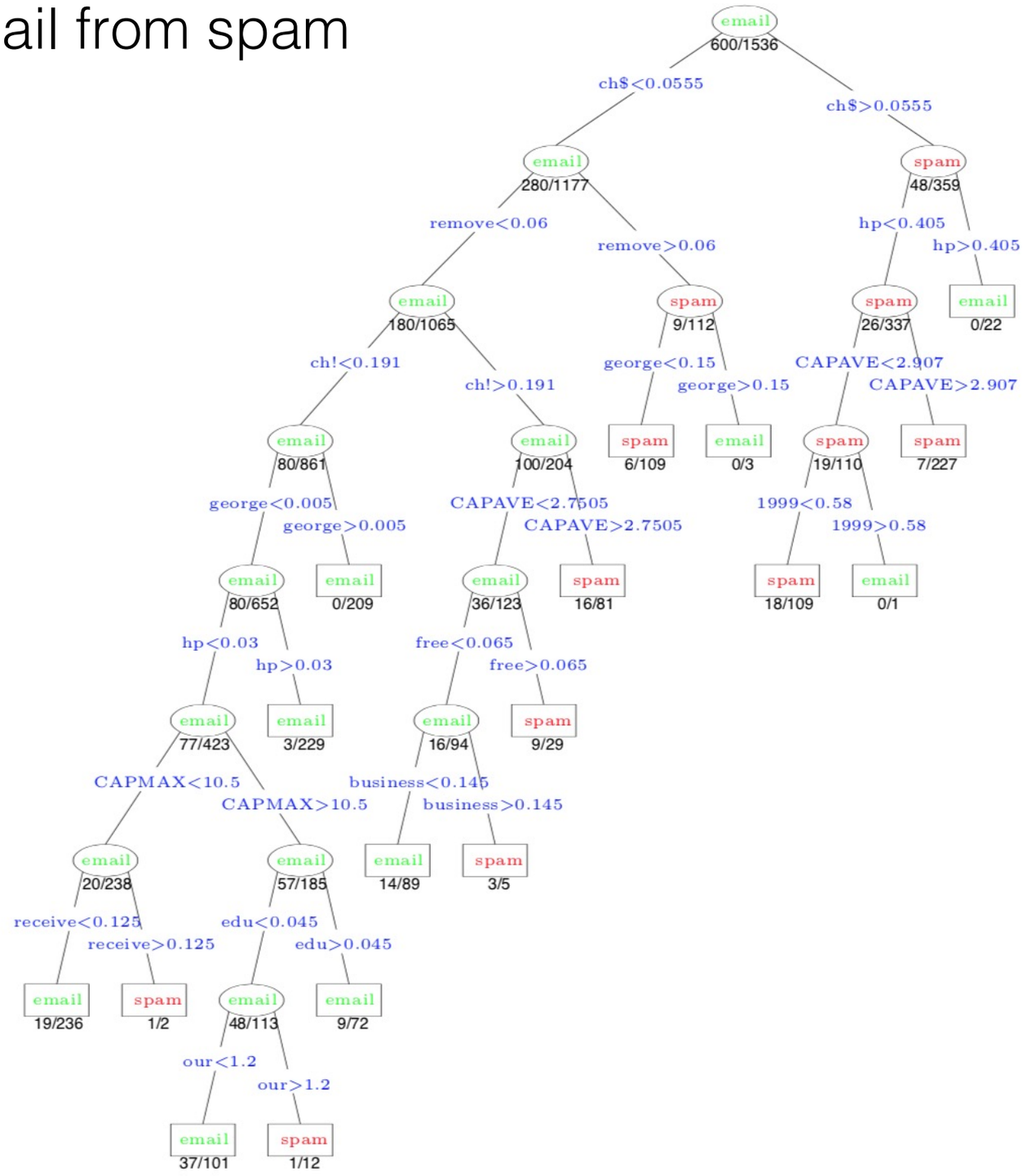
Decision Tree

- Nodes:
  - The data is split based on a value of one of the input features at each code
  - All variables can be considered at each node (irrespective of their use in previous node) → Allows to find intervals of interest in a particular variable
- Leaves:
  - Terminal nodes
  - Represent a class label or probability
  - When the outcome is a continuous variable it is considered a regression tree.
- The splits are created recursively
  - The process is repeated until some stop condition is met
  - Ex: depth of tree, no more information gain, etc...



Example

Classifying email from spam

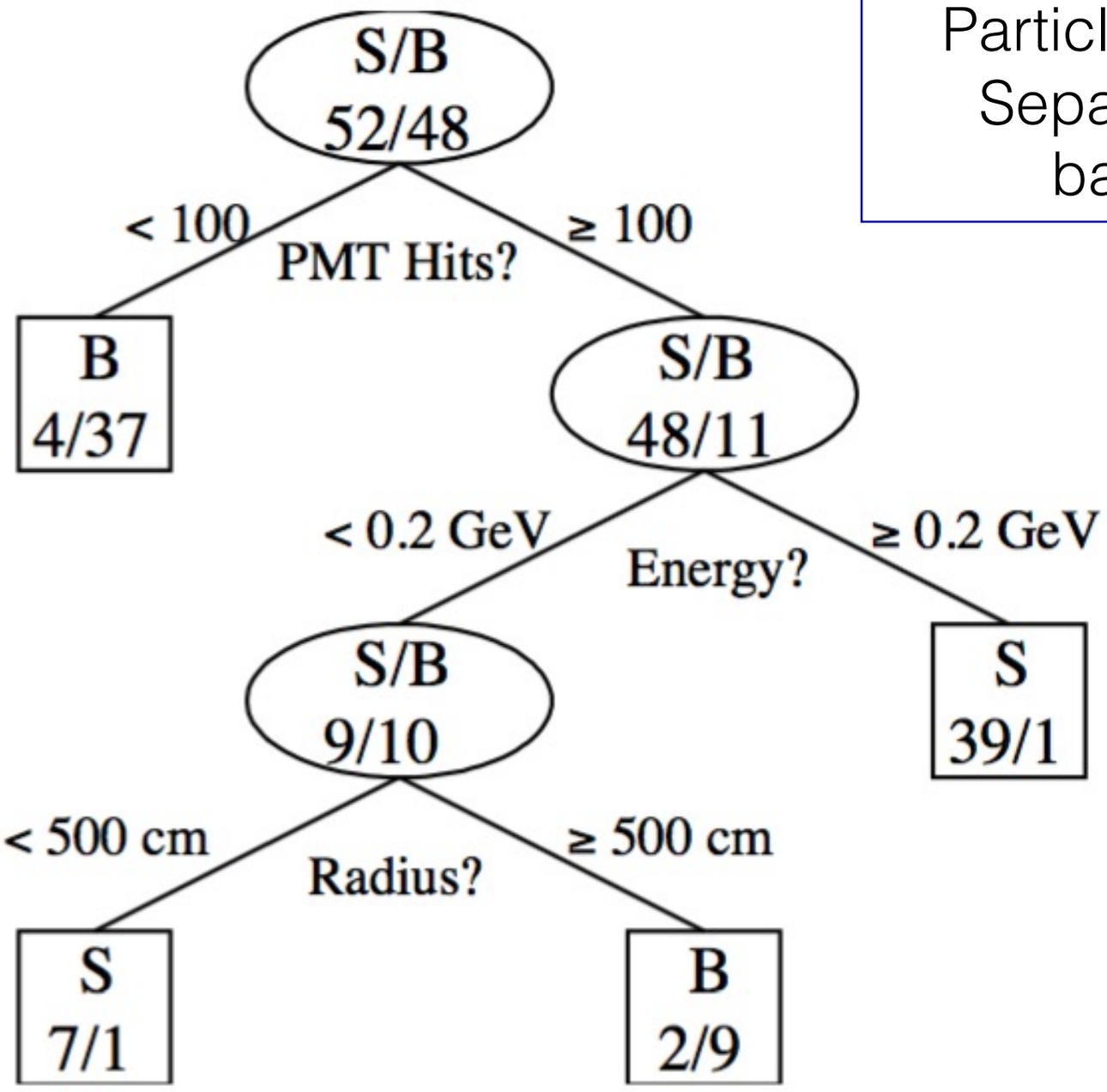


Ref: T. Hastie et al.

Example from HEP

MiniBooNE experiment
arXiv: 0408124

Signal/Background

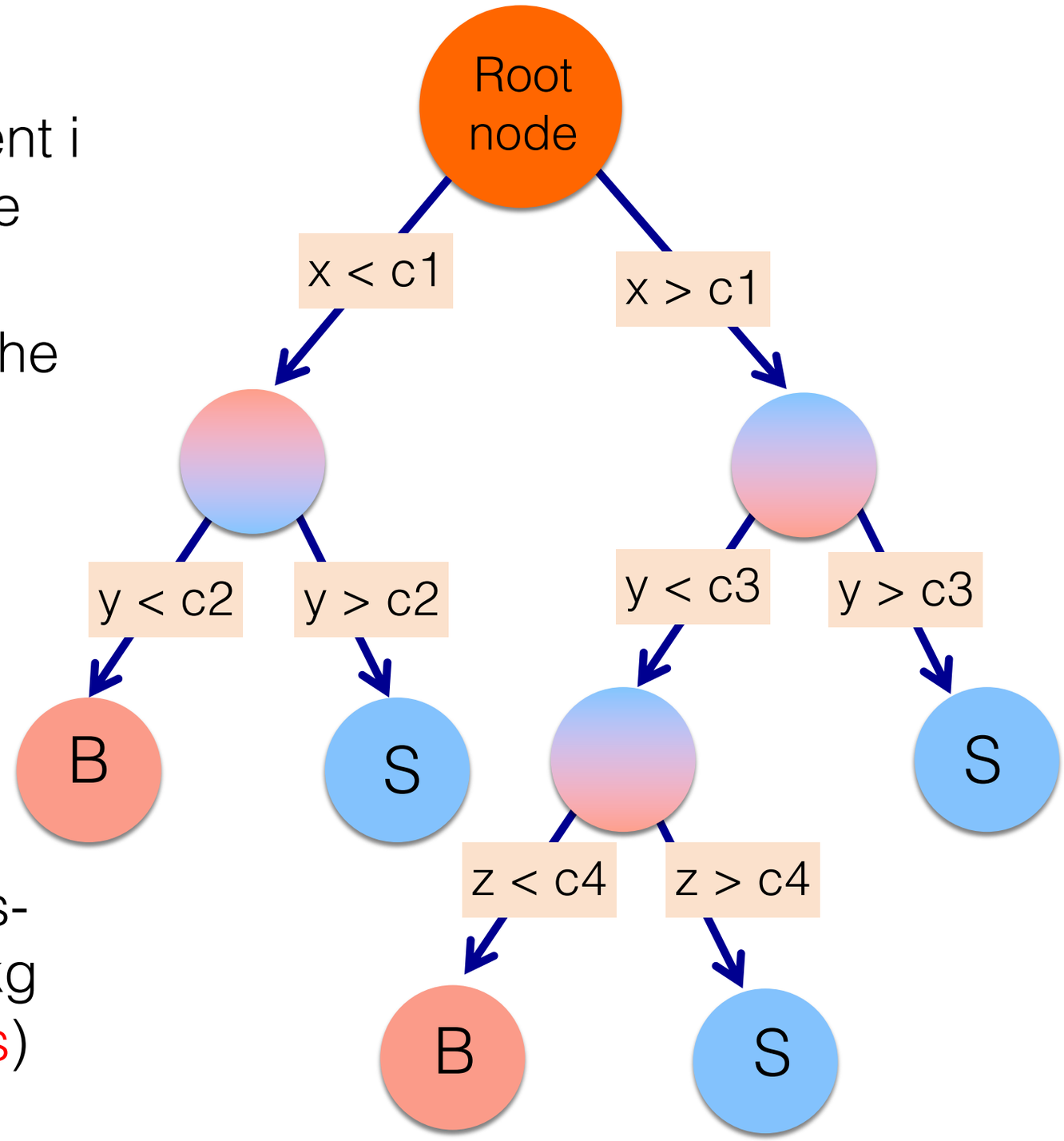


Particle identification: Separating ν_e from backgrounds

Decision Tree

Tree Output:

- The tree output for a given event i is the value associated with the leaf where the event falls.
- Several conventions used for the value associated to leaf:
  - Purity: p = s/(s+b), 0 < p < 1
  - Binary:
    - signal = 1 (if p > 0.5),
    - bkg = 0 or -1 (if p < 0.5)



- Some signal events can be mis-classified as background or bkg classified as signal: Error (Loss)

Tree Parameters

Decision trees have relatively limited number of parameters

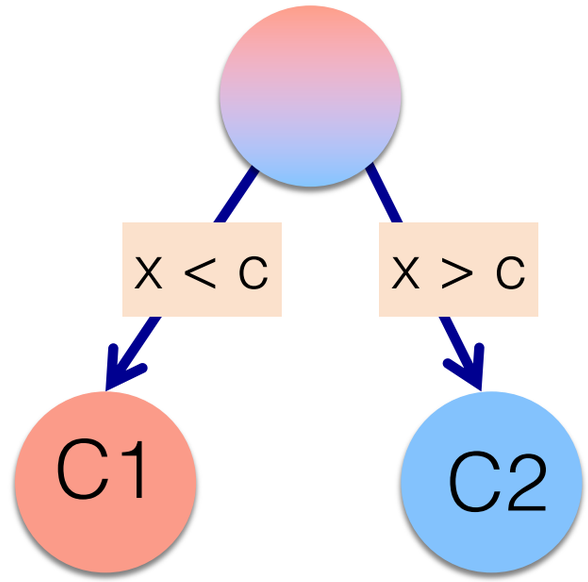
1. How to normalize signal & background before starting the training (applies to most training techniques)
   - Conventionally sum of weights (signal) = sum of weights (backgrounds)
     - Purity = 0.5
2. Selection criteria for splits
   - Requires a list of discriminating variables and a way to evaluate best separation
3. Stopping criteria
   - Minimum leaf size
     - Require at least N_{min} events in each node after splitting
       - Ensures statistical significance of purity measurement
   - Reached perfect separation
   - Insufficient improvement with further splitting
   - A maximal tree depth

Node Splitting

Basic Concept:

- Consider an impurity/error measure, E
  - Maximal for equal mix of signal and background
  - Minimal for perfect separation
  - Symmetric in signal & background purities
- Find the variable and its split value that decreases impurity
  - i.e. maximize $\Delta E = E(\text{parent}) - f_{c1} \times E(\text{child-1}) - f_{c2} \times E(\text{child-2})$

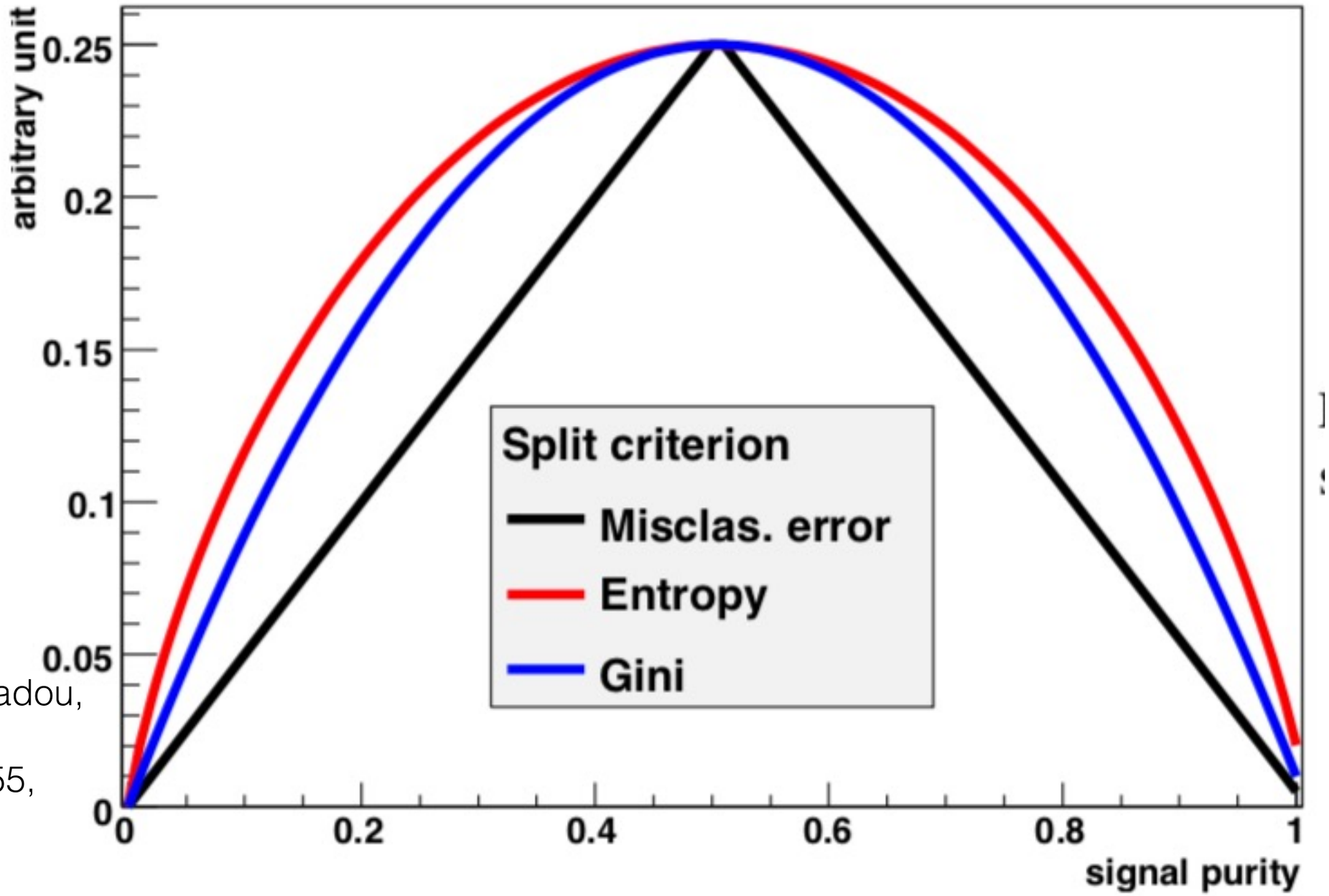
    $f_{c1/c2}$ are the fraction of events falling to child-1/2 node



Common impurity functions:

- Misclassification error : $1 - \max(p, 1 - p)$
- The (cross) entropy : $- p \log p - (1 - p) \log(1 - p)$
- Gini index : $2p(1 - p)$

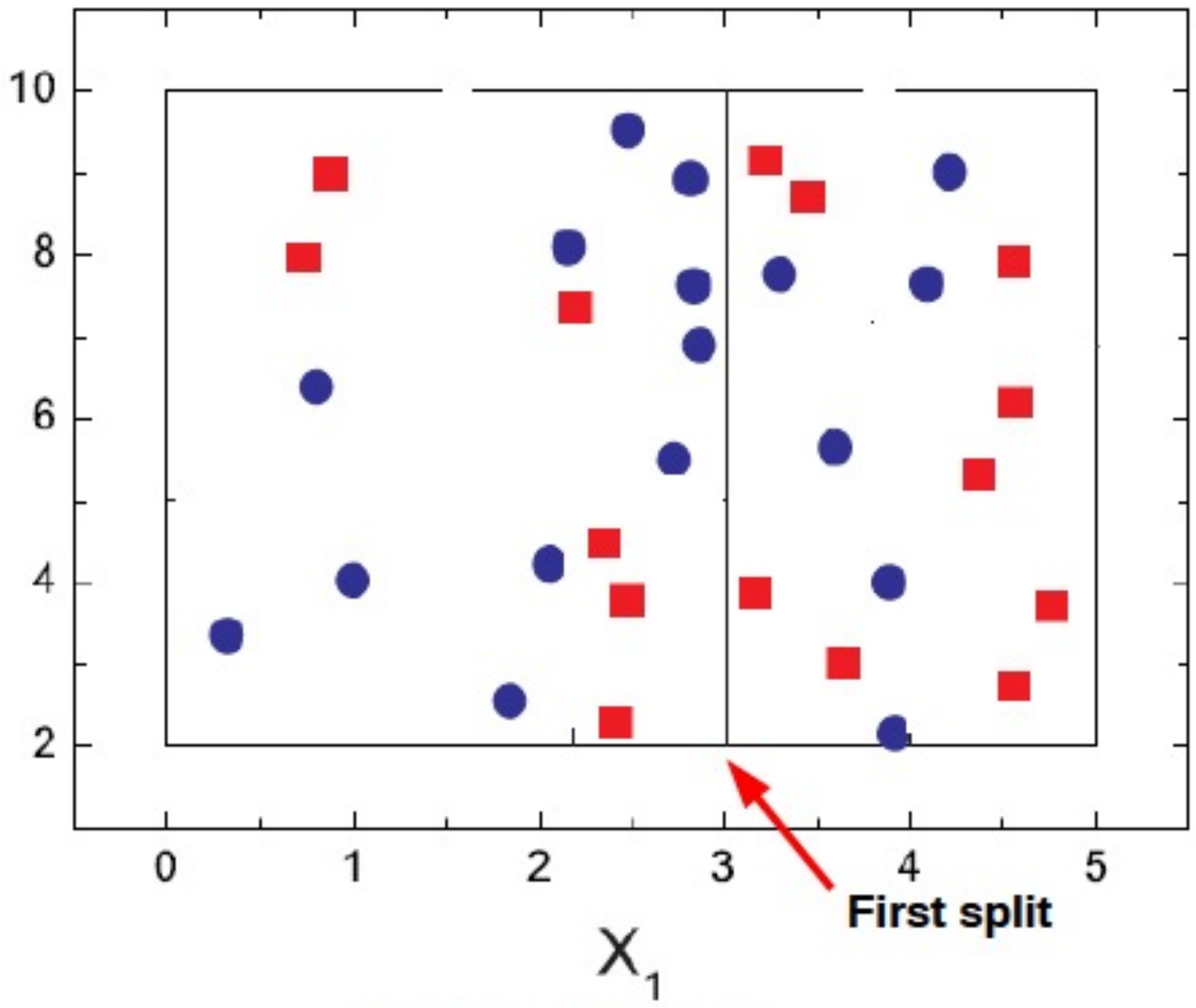
Impurity functions



Ref.: Yann Coadou, EPJ Web of Conferences 55, 02004 (2013)

Binary splitting

P = 15/32
1-P = 17/32
w_i = 1/32
G = 0.25



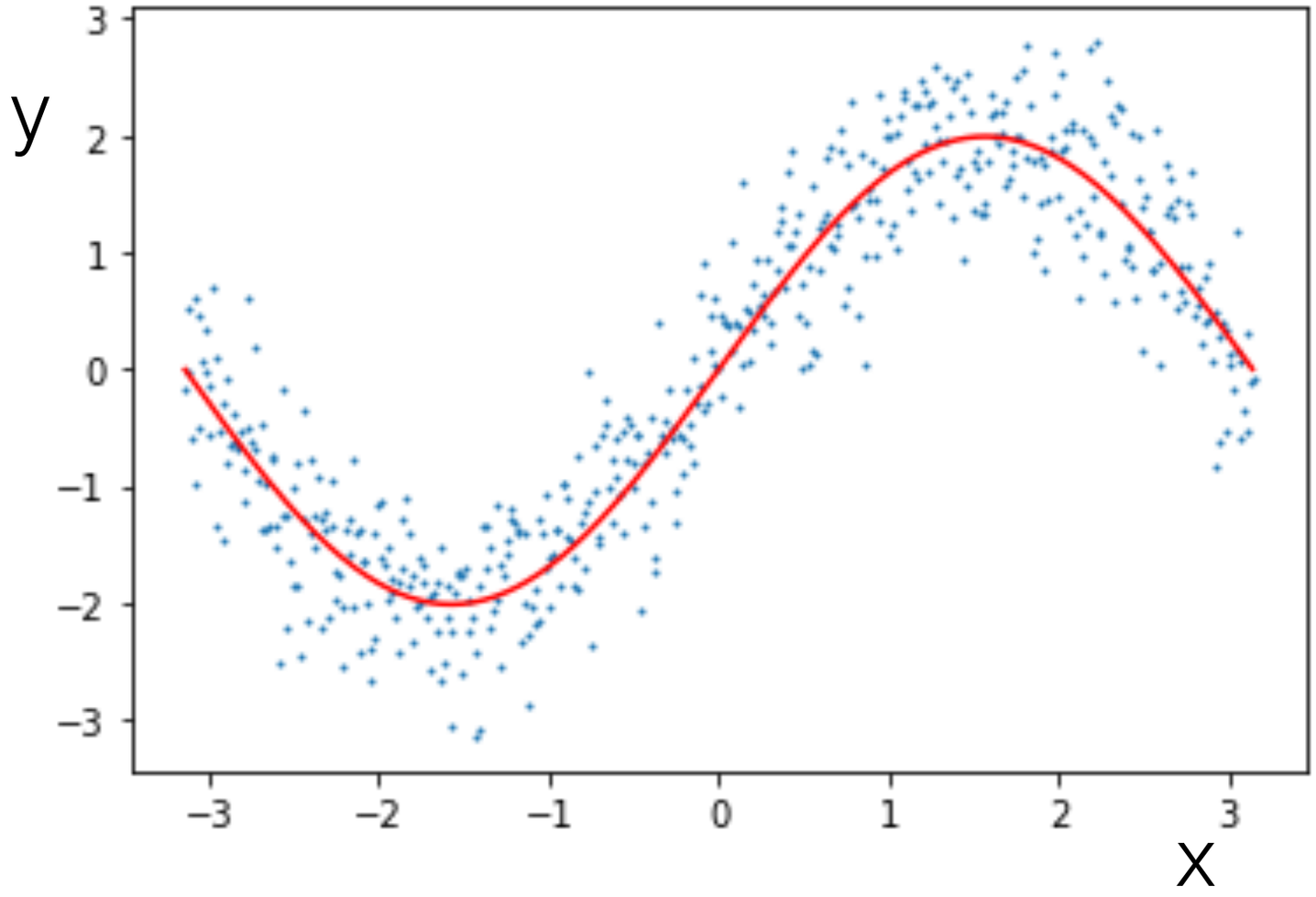
PL = 6/17
1-PL = 11/17
w_i = 1/17
GL = 0.23

PR = 9/15
1-PR = 6/15
w_i = 1/15
GR = 0.24

Regression

- Fitting to a set of data points
- If the functional behaviour is known, then, perform a parametric fit by minimising chi^2 (or any other loss function)
- Not always easy, especially if the input feature is multi-dimensional

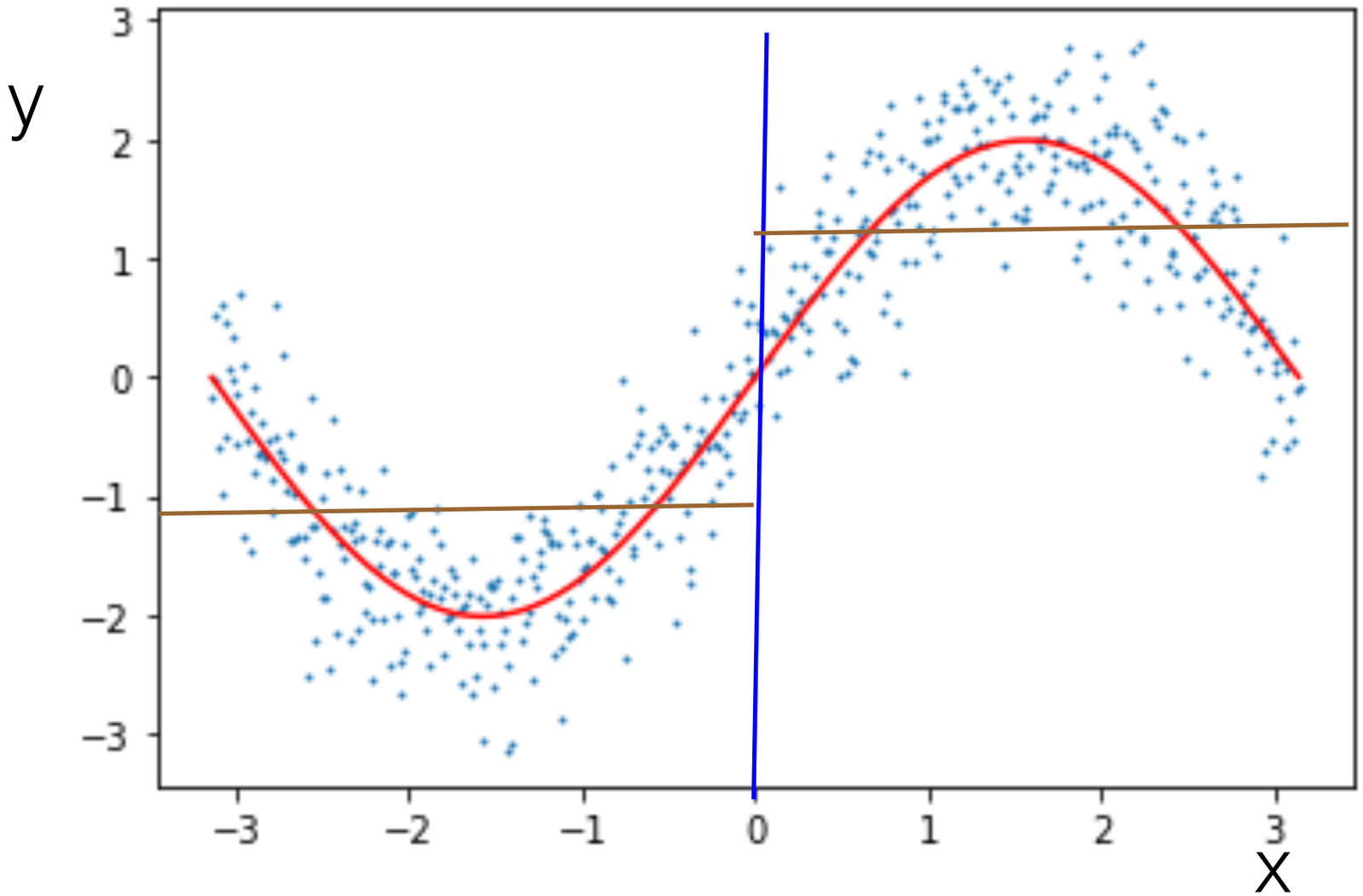
Regression: Finding the mapping function that describes an approximation of the underlying functional behaviour defining the target value



Regression Tree

Growing a regression tree is similar to that of classification tree, except

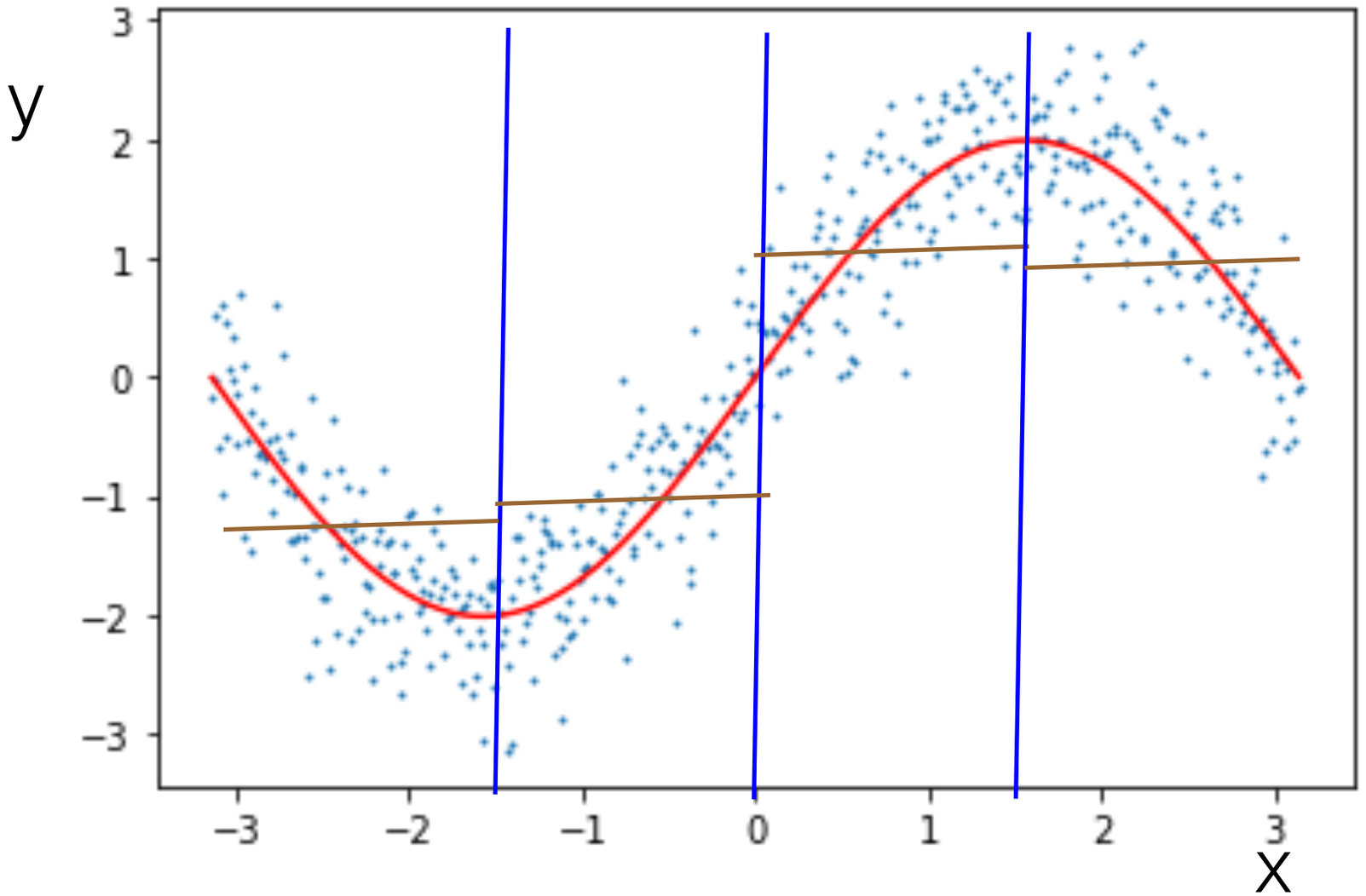
- Model the response in each region as a constant (average of y in the region)
- Impurity function for node splitting: sum of squares $\Sigma(y - y')^2$



Regression Tree

Growing a regression tree is similar to that of classification tree, except

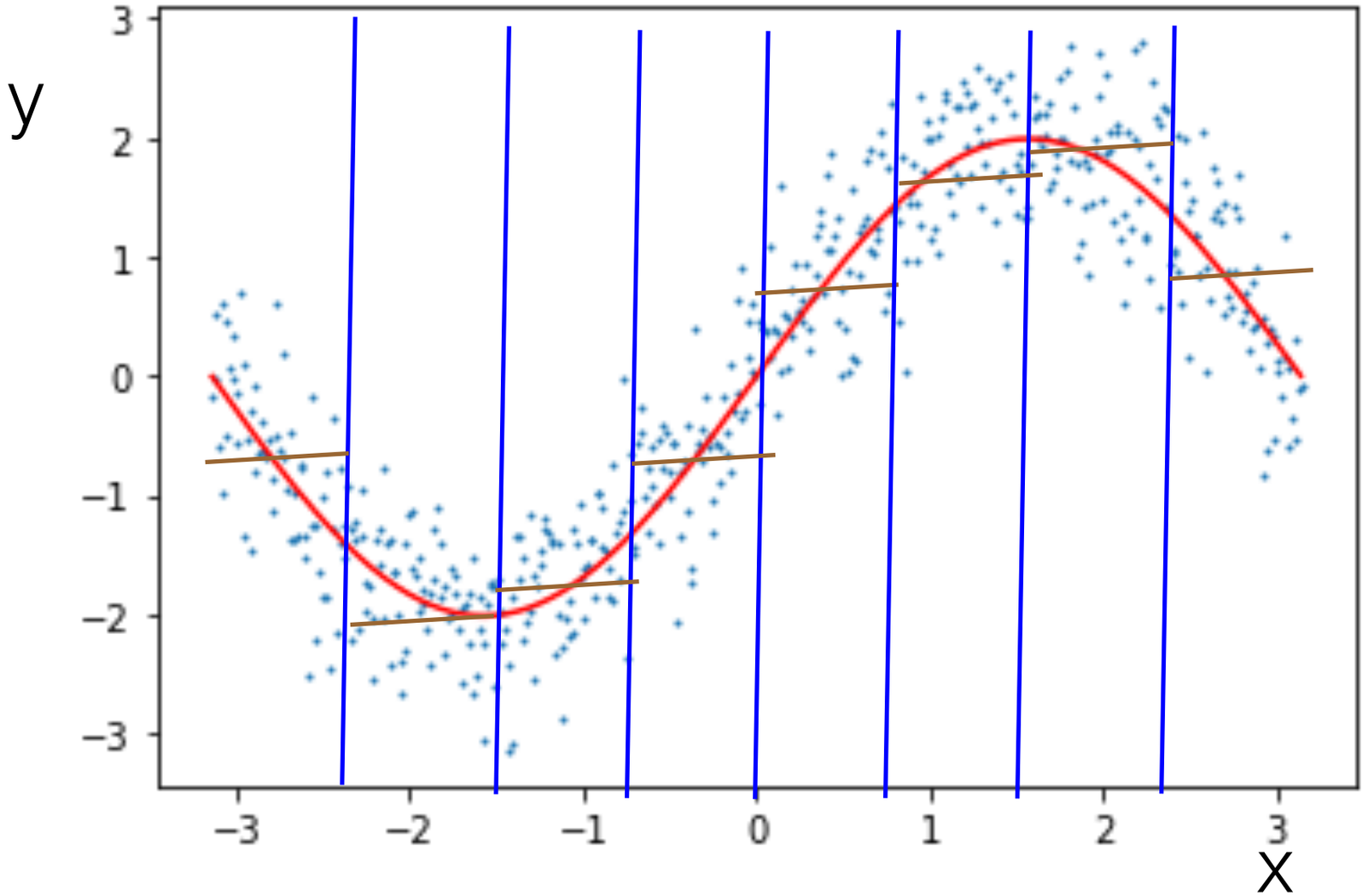
- Model the response in each region as a constant (average of y in the region)
- Impurity function for node splitting: sum of squares $\Sigma(y_i - f(x_i))^2$



Regression Tree

Growing a regression tree is similar to that of classification tree, except

- Model the response in each region as a constant (average of y in the region)
- Impurity function for node splitting: sum of squares $\Sigma(y_i - f(x_i))^2$

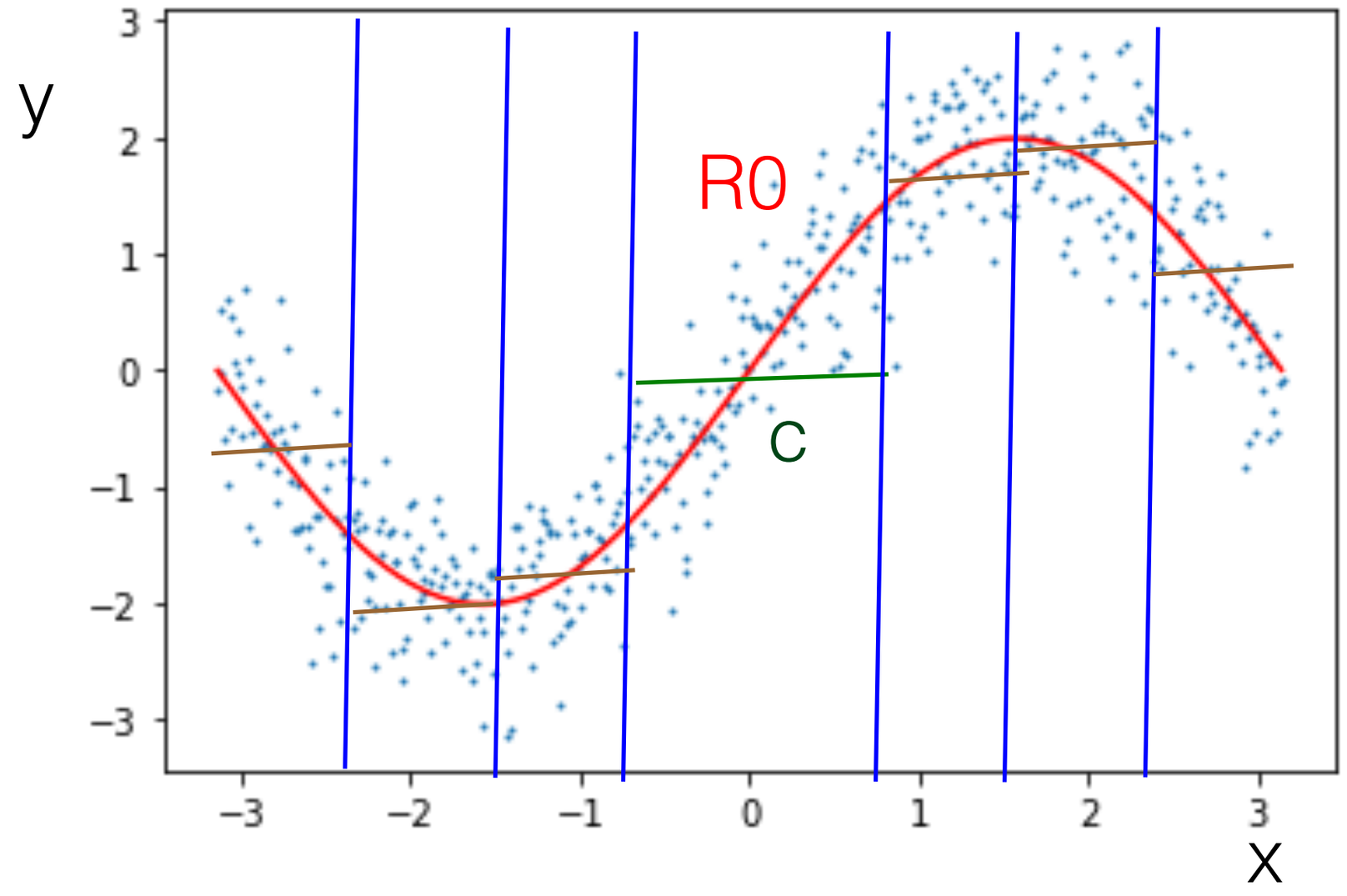


Regression Tree

Splitting minimises sum of squares $\Sigma(y_i - f(x_i))^2$

i.e. split such that $\Sigma(y_j - c1)^2 + \Sigma(y_k - c2)^2 < \Sigma(y_i - c)^2$

Where, $c = \langle y_i | x_i \text{ in } R_0 \rangle$, $c1 = \langle y_j | x_j \text{ in } R_1 \rangle$, $c2 = \langle y_k | x_k \text{ in } R_2 \rangle$

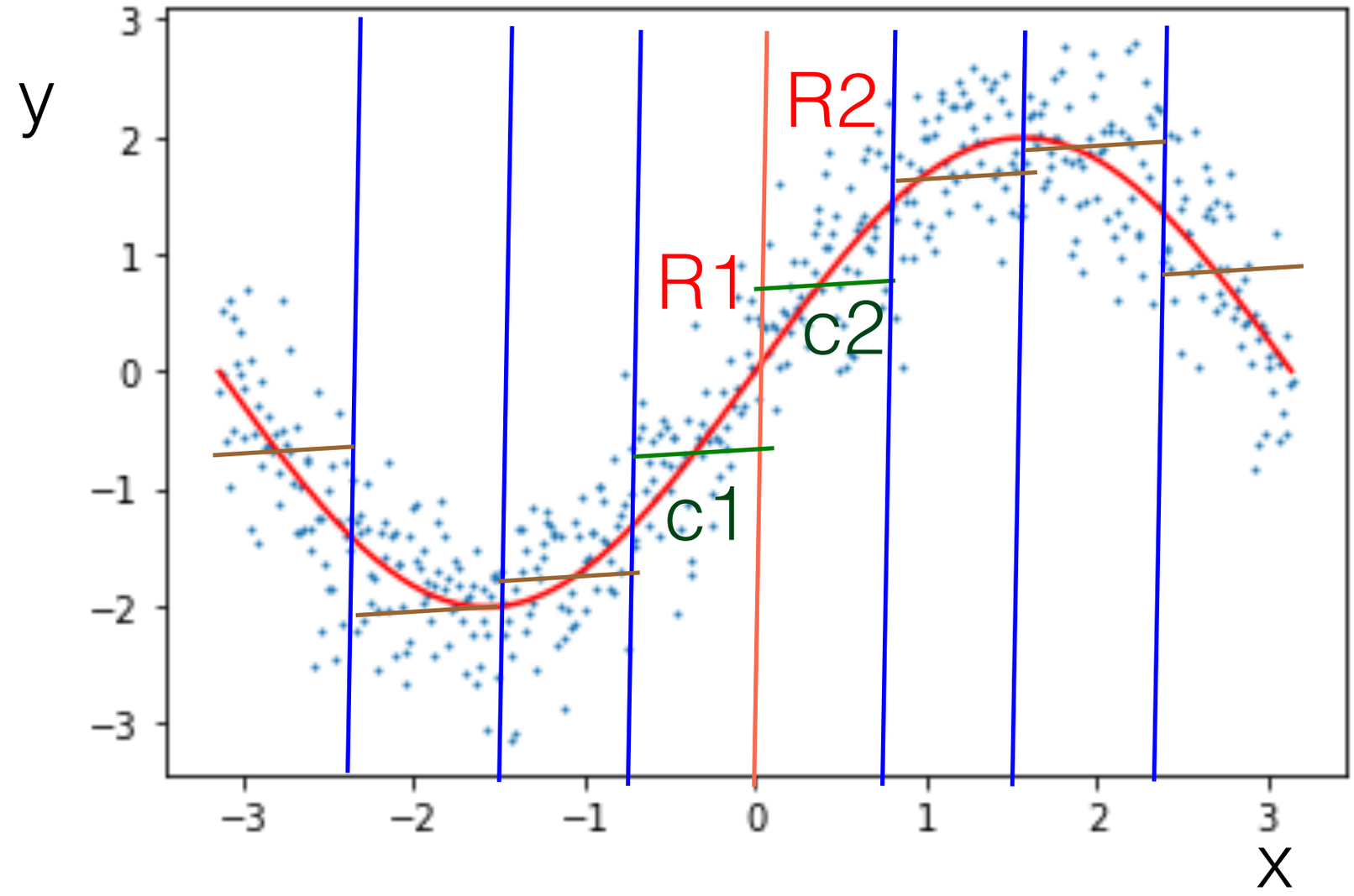


Regression Tree

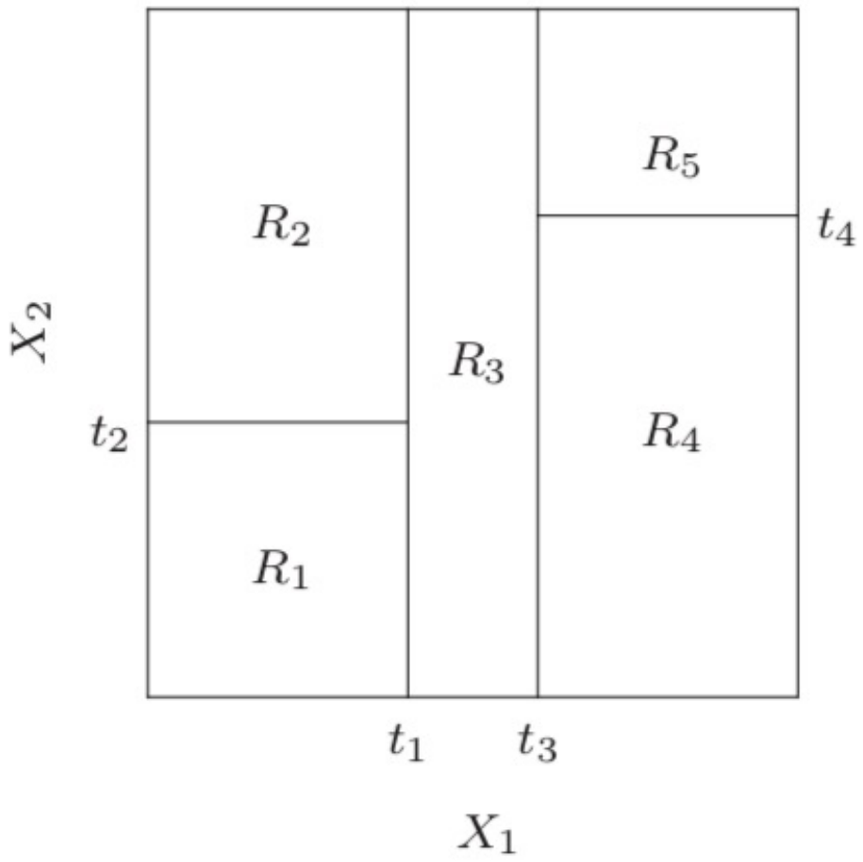
Splitting minimises sum of squares $\Sigma(y_i - f(x_i))^2$

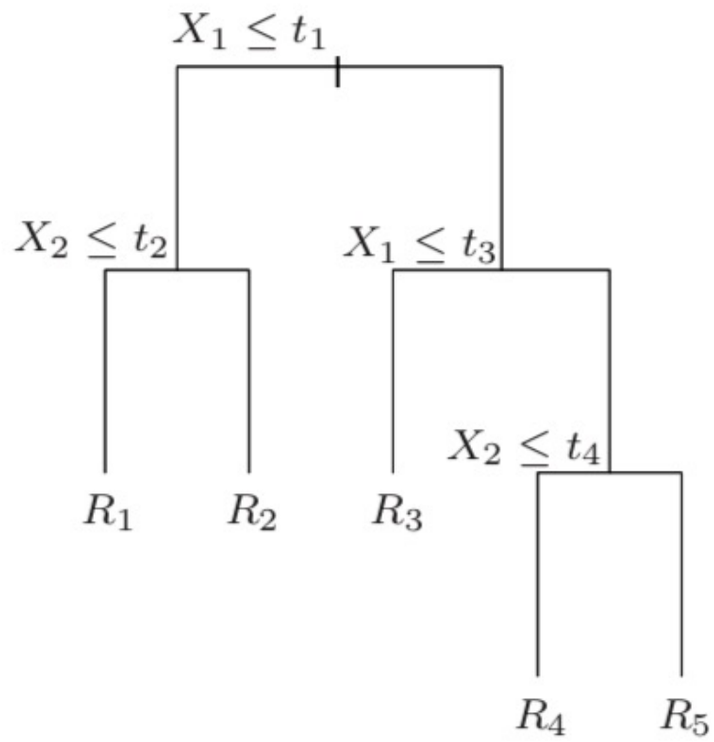
i.e. split such that $\Sigma(y_j - c1)^2 + \Sigma(y_k - c2)^2 < \Sigma(y_i - c)^2$

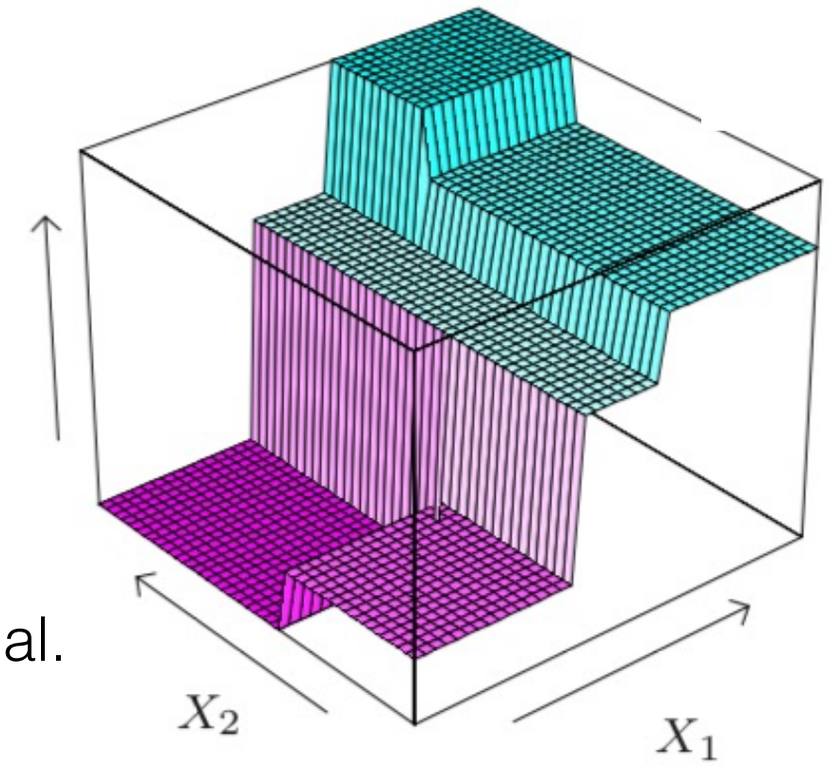
Where, $c = \langle y_i | x_i \text{ in } R_0 \rangle$, $c1 = \langle y_j | x_j \text{ in } R_1 \rangle$, $c2 = \langle y_k | x_k \text{ in } R_2 \rangle$



Example







regression surface Represented by node means

Ref: Figure 9.2,T. Hastie et al.

Tree Stability

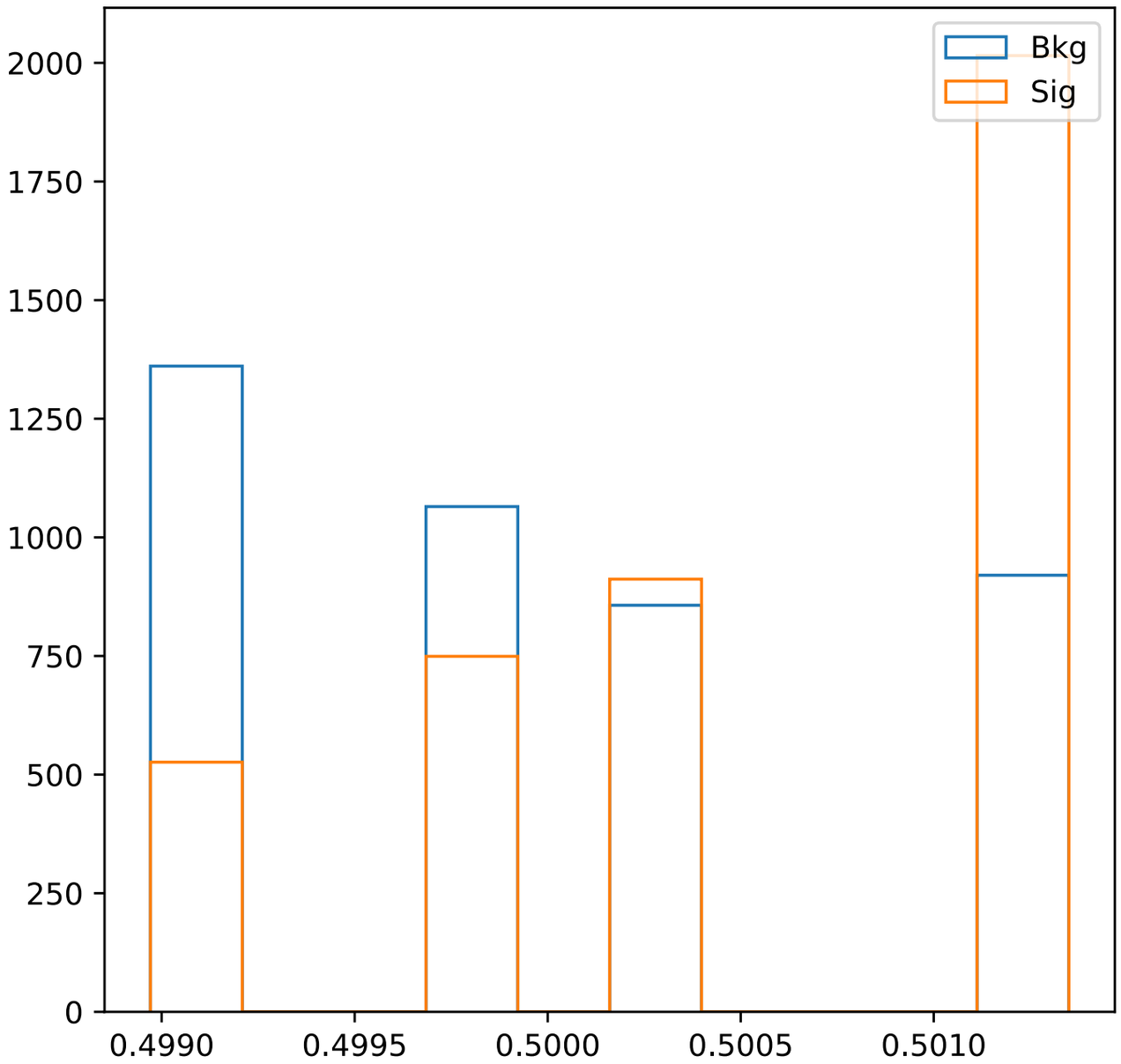
Decision trees are usually unstable

- If too optimised for the training sample, may not generalize very well to the unknown events
- subject to statistical uncertainty in training sample
  - A small change in training sample can lead to drastically different tree structure
- Output of a single decision tree is discrete
  - Delta functions at ±1 for binary
  - Spikes at specific purity values

Solution to these shortcomings are averaging

- Also increases discriminating power
- Several techniques: Bagging, Boosting, Random Forests

Example output of a single decision tree



Single tree
With max_depth=10

Tree Boosting

Boosting is a procedure that combines many “weak” classifiers to produce a powerful one

- Weak classifier is one that is slightly better than random guessing
- The general idea is to use the weak learning method several times to get a succession of hypotheses, each one refocused on the events that the previous ones found difficult and misclassified

Popular Boosting Methods:

- AdaBoost
- ε-Boost (shrinkage)
- Gradient Boost

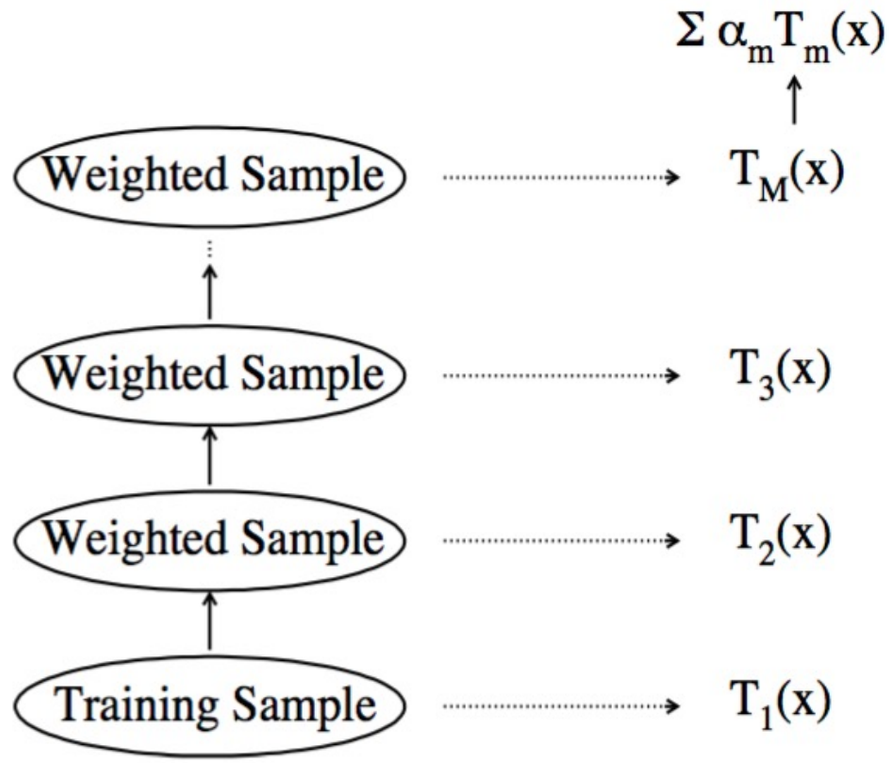
AdaBoost

General Procedure:

- Each tree is created iteratively
- The tree's output is given a weight relative to its accuracy
- Events which are misclassified, increase their weights
- Build a new tree, repeat the procedure for several trees
- The final score of an event is the weighted average of scores from all trees

$$\hat{y} = \sum_m \alpha_m T_m(x),$$

This means that samples that are difficult to classify receive increasing larger weights until the algorithm identifies a model that correctly classifies these samples



The goal is to minimize training loss, defined by

$$L = \sum_{i=1}^{n} l(y_i, \hat{y}_i)$$

(distance between truth and prediction value for i_{th} sample)

AdaBoost

Let's weights of each event is w_i

For m_{th} tree, define error and tree weight

$$err_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq T_m(x_i))}{\sum_{i=1}^{N} w_i},$$

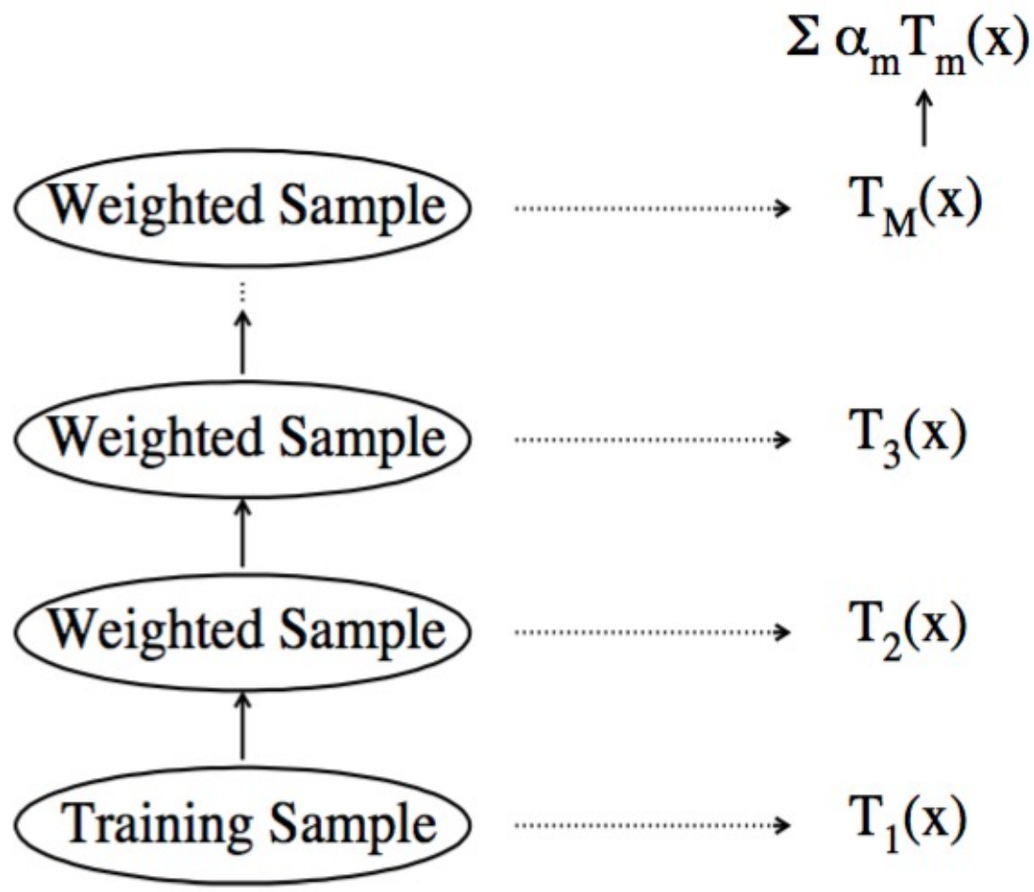
$$\alpha_m = \beta \times \log((1 - err_m)/err_m)$$

Change the weight of each event by

$$w_i \rightarrow w_i \times \exp(\alpha_m I(y_i \neq T_m(x_i)))$$

Normalize weights

$$w_i \rightarrow w_i / \sum_{i=1}^{N} w_i$$



Repeat training for N_{trees}
The score of a given event is:

$$T(x) = \sum_{m=1}^{N_{tree}} \alpha_m T_m(x)$$

ε-Boost (shrinkage)

Change the weight of the i_{th} event as

$$w_i \rightarrow w_i \times \exp(2\varepsilon I(y_i \neq T_m(x_i)))$$

ε is a constant of the order of 0.01

Normalize weights

$$w_i \rightarrow w_i / \sum_{i=1}^{N} w_i$$

The score for a given event is

$$T(x) = \sum_{m=1}^{N_{tree}} \varepsilon T_m(x)$$

renormalized, but unweighted, sum of the scores over individual trees.

Both the boosting algorithms minimize the expectation value of the loss function:

$$L(T(x), y) = e^{-yT(x)}$$

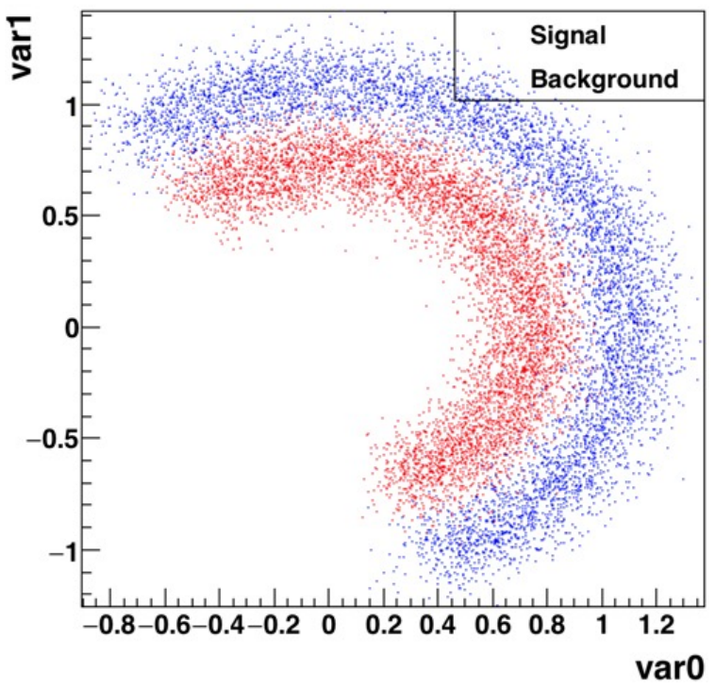
Where y = 1 for signal,
-1 for background

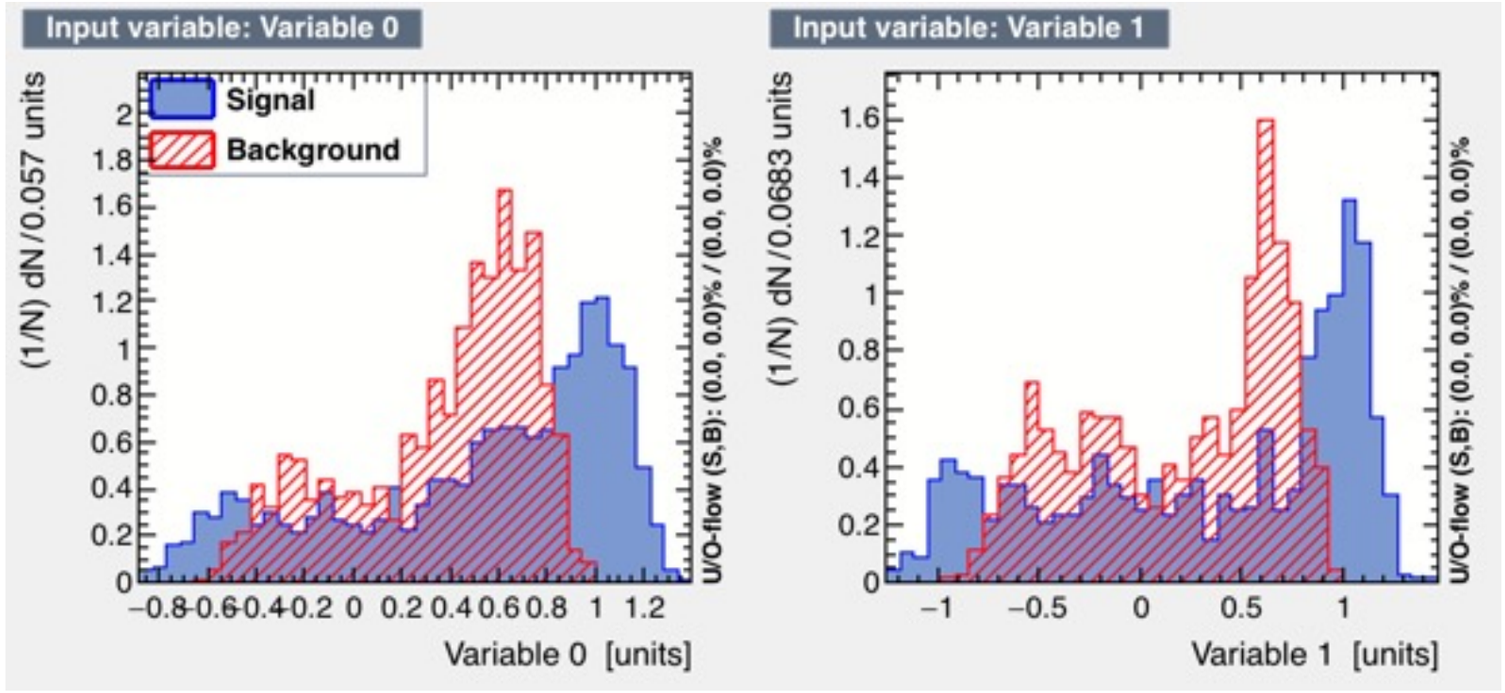
$$T(x) = \Sigma_{m=1}^{N_{tree}} (\alpha_m \, or \, \varepsilon) T_m(x)$$

$T_m(x) = 1$ If event lands on signal leaf

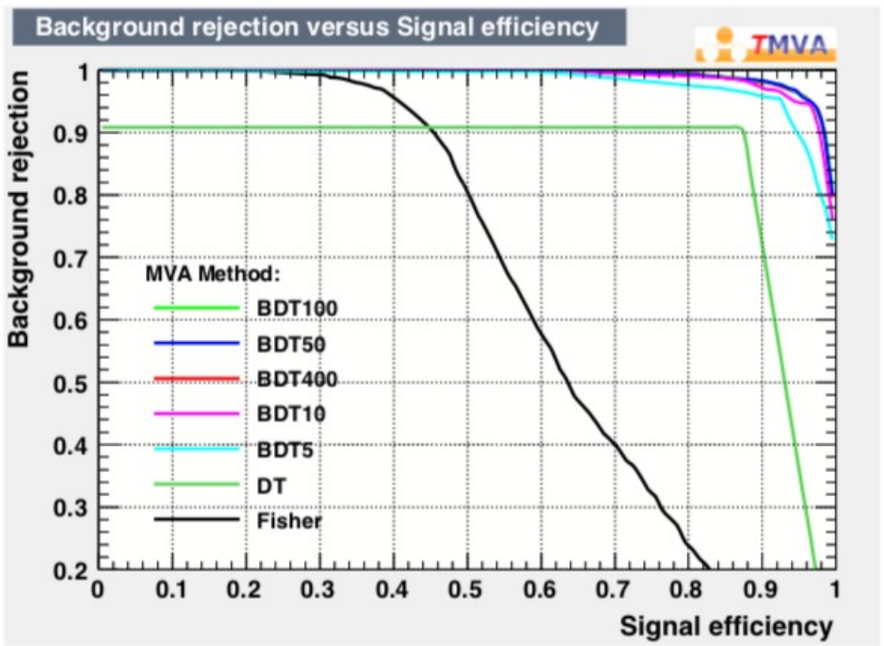
$T_m(x) = -1$ If event lands on background leaf

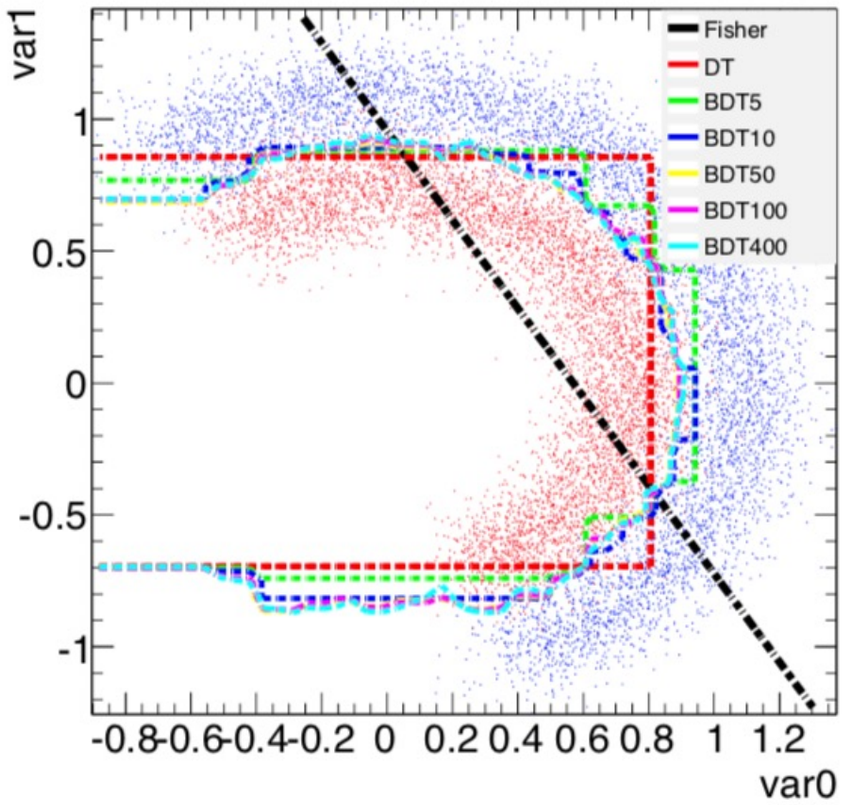
A simple example





Ref.: Yann Coadou, EPJ Web of Conferences 55, 02004 (2013)





Gradient Boosting

- Exponential loss has the shortcoming that it lacks robustness in presence of outliers or mislabelled data points
  - The performance of AdaBoost therefore is expected to degrade in noisy settings
- The Gradient Boosting algorithm attempts to cure this weakness by allowing for other, potentially more robust, loss functions without giving up on the good out-of-the-box performance of AdaBoost

(ref: TMVA userguide)

Gradient Boosting

The goal is to minimize $L(f) = \Sigma\, L(y_i, f(x_i))$ with respect to f,

Where, $f(x)$ = sum of trees = $\Sigma\, T_m(x)$

The gradient of L with respect to f, evaluated at m^{th} iteration

$$\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x_i)=f_{m-1}(x_i)} = g_{im}$$

Add a Tree such that

$$f_m = f_{m-1} + T_m = f_{m-1} - \eta_m g_m$$

Where, η_m is a scalar, called "step length or learning length"

i.e., at m_{th} iteration, introduce a Tree, whose predictions are as close as possible to negative gradient ($-g_{im}$) (for $\eta_m = 1$)

Gradient Boosting

If $L(f) = \frac{1}{2}\Sigma (y_i - f(x_i))^2$, i.e. square error loss

The negative gradient of L would be

$$-\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right] = -g_{im} = (y_i - f(x_i))$$

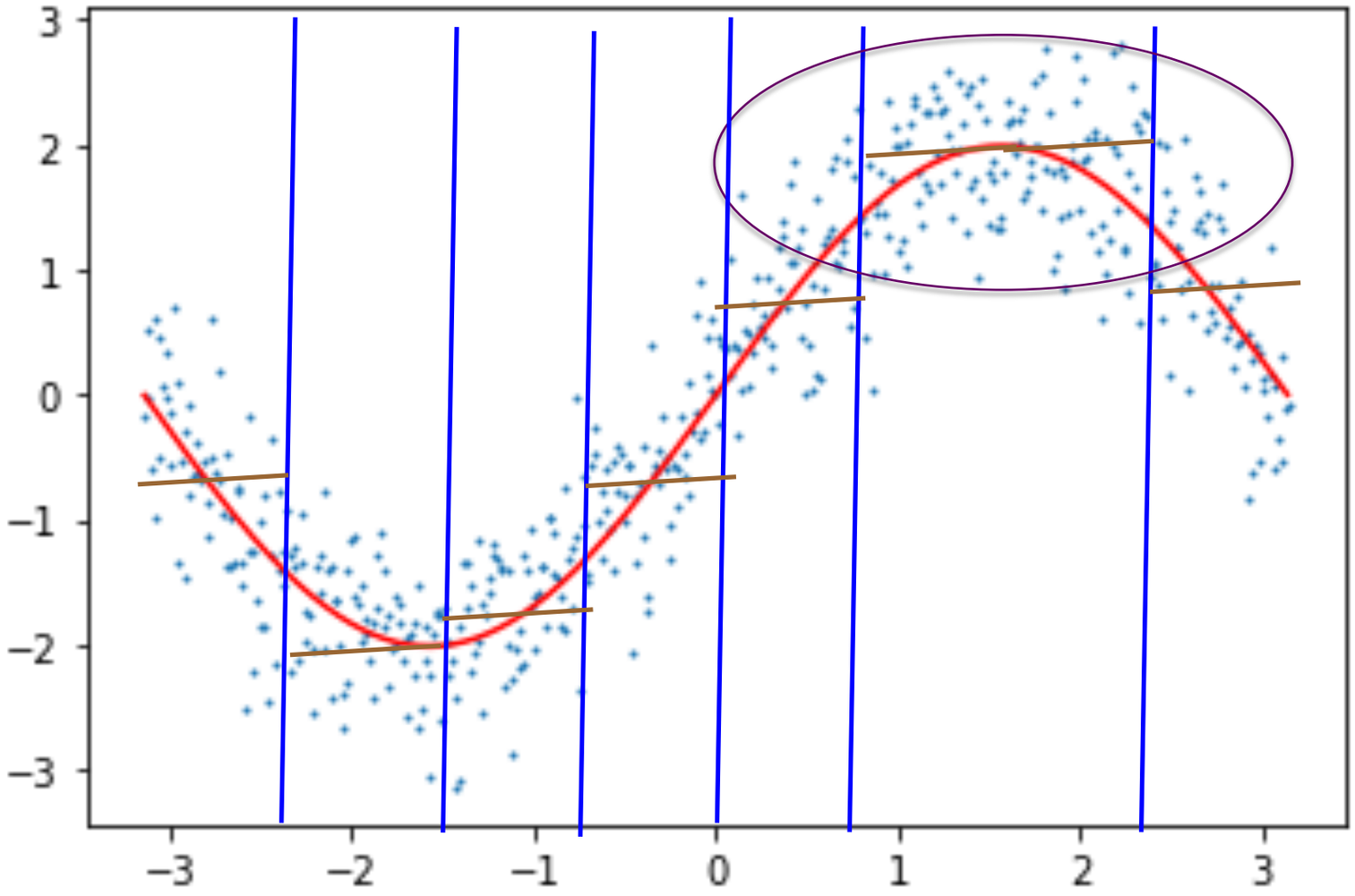
$$(y_i - f(x_i)) = y_i - < y_i \mid x_i >$$

i.e. Residual

Thus, at each iteration, add a Tree, which fits to the residual from previous iteration

Gradient Boosting

Residual for each x_i : y_i - $\langle y_i \rangle$



Gradient Boosting

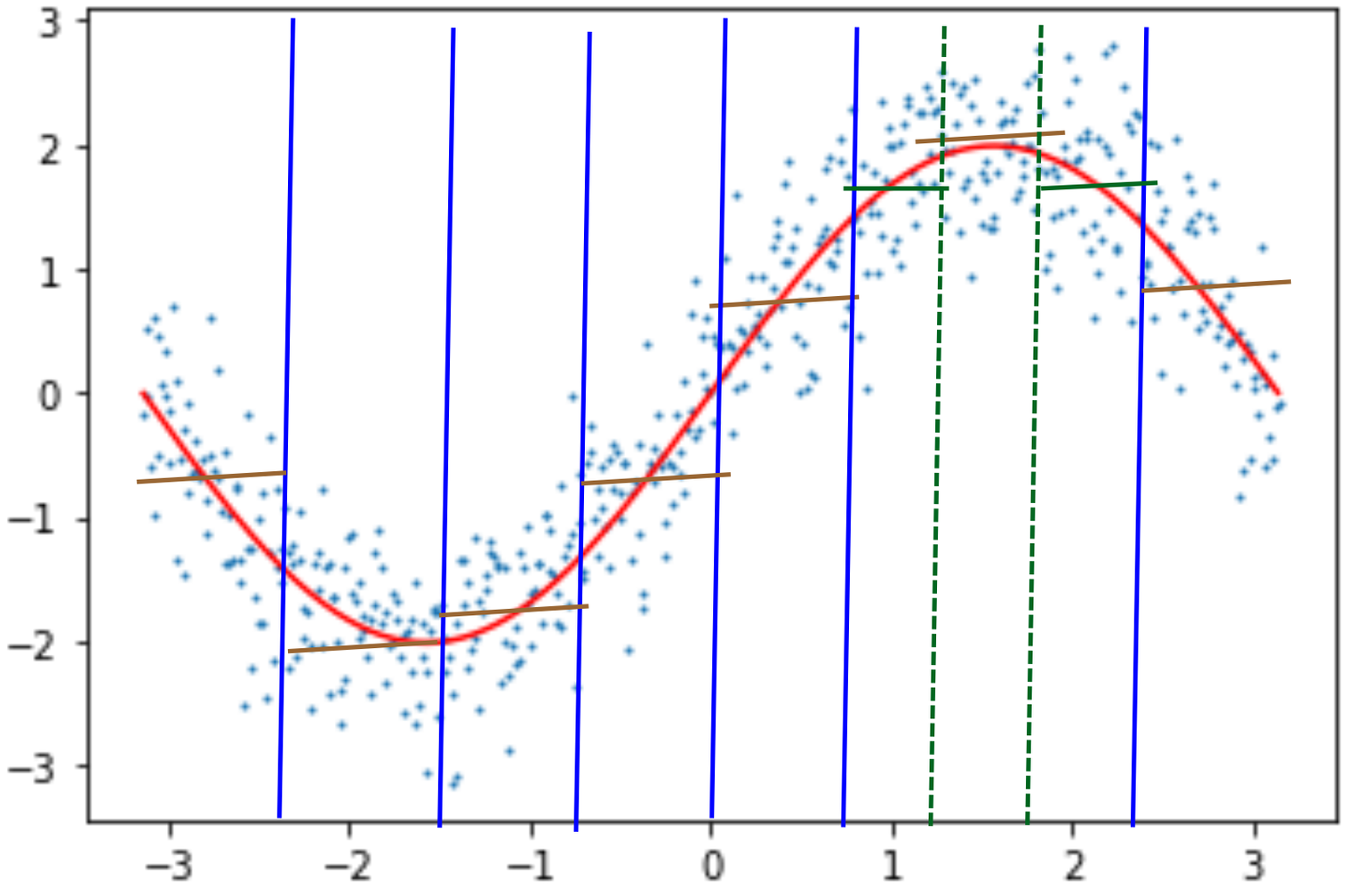
Residual for each x_i = y_i - $\langle y_i \rangle$

Fitting to residual gives new average $\langle y_i \rangle$ for each x_i

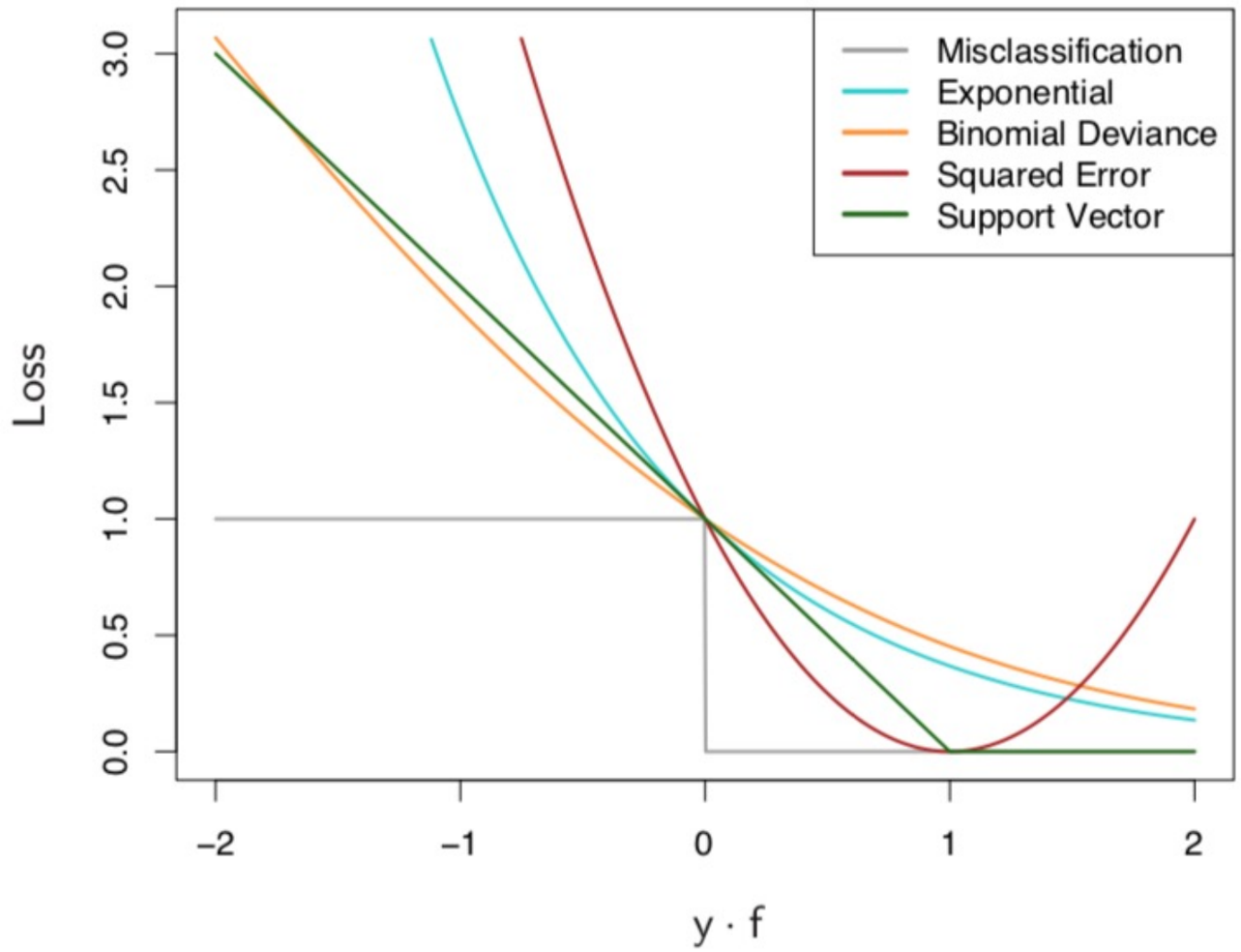
So, the net output for x_i is $f(x_i) = \langle y_i \rangle + \langle y_i \rangle$

The loss is reduced at every iteration

Boundaries of the new tree are in general different



Loss Functions for Classification



$$-l(Y, f(x)) = \log\left(1 + e^{-2Yf(x)}\right)$$

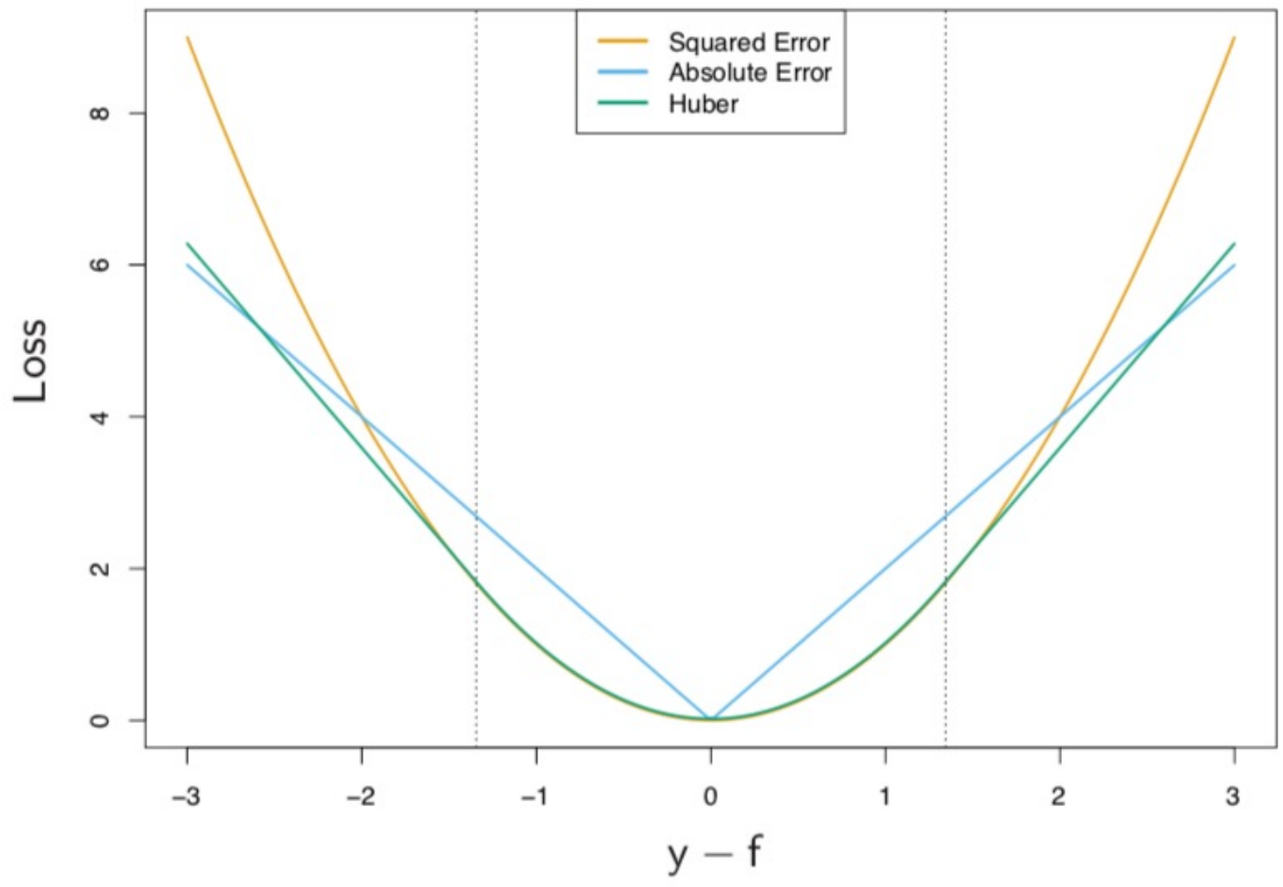
FIGURE 10.4. *Loss functions for two-class classification. The response is* $y = \pm 1$*; the prediction is* f*, with class prediction* $\text{sign}(f)$*. The losses are misclassification:* $I(\text{sign}(f) \neq y)$*; exponential:* $\exp(-yf)$*; binomial deviance:* $\log(1 + \exp(-2yf))$*; squared error:* $(y - f)^2$*; and support vector:* $(1 - yf)_+$ *(see Section 12.3). Each function has been scaled so that it passes through the point* $(0, 1)$*.*

T. Hastie et al.

Loss Functions for Regression

- Huber loss is more robust against outliers
- Squared error gives more weightage to outliers, making the trees less robust.

$$L(y, f(x)) = \begin{cases} [y - f(x)]^2 & \text{for } |y - f(x)| \le \delta, \\ 2\delta|y - f(x)| - \delta^2 & \text{otherwise.} \end{cases}$$



T. Hastie et al.

FIGURE 10.5. *A comparison of three loss functions for regression, plotted as a function of the margin $y-f$. The Huber loss function combines the good properties of squared-error loss near zero and absolute error loss when $|y - f|$ is large.*

Gradients of loss functions

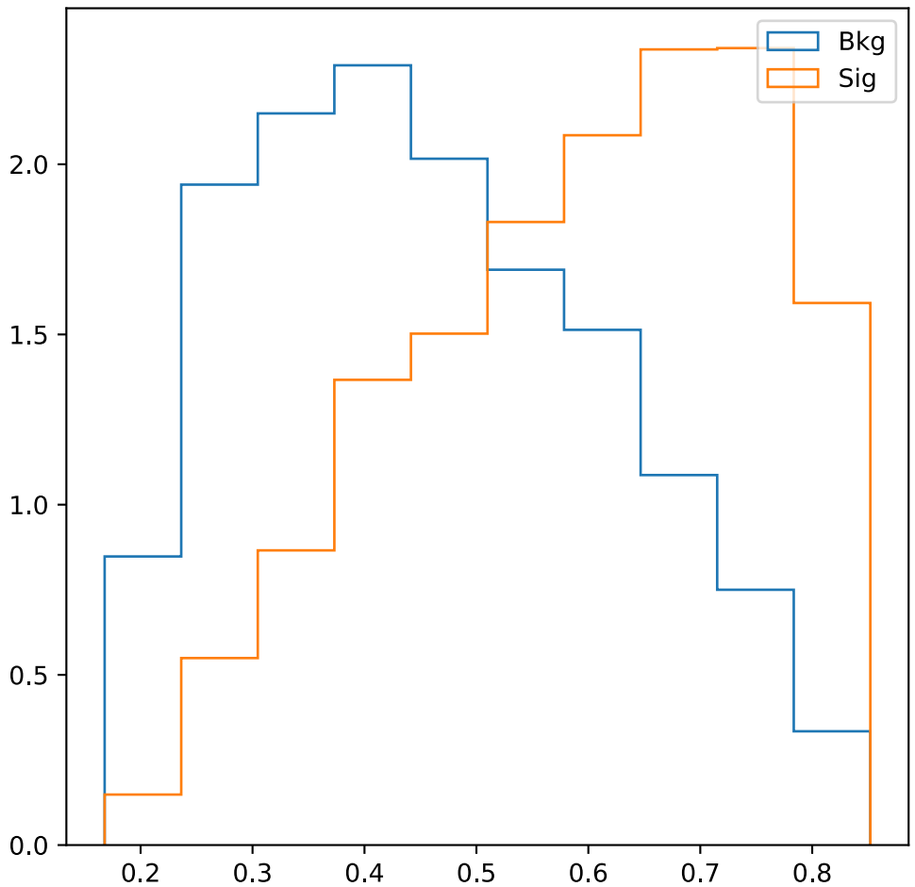
TABLE 10.2. *Gradients for commonly used loss functions.*

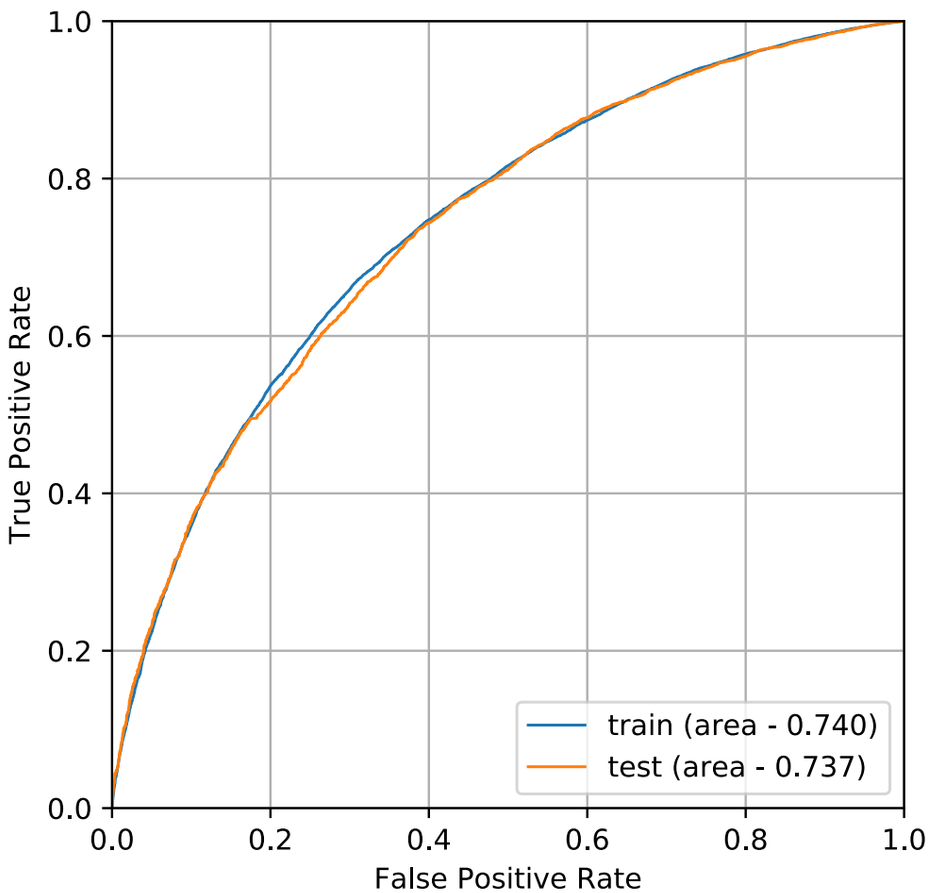
Setting	Loss Function	$-\partial L(y_i, f(x_i))/\partial f(x_i)$
Regression	$\frac{1}{2}[y_i - f(x_i)]^2$	$y_i - f(x_i)$
Regression	$\lvert y_i - f(x_i)\rvert$	$\text{sign}[y_i - f(x_i)]$
Regression	Huber	$y_i - f(x_i)$ for $\lvert y_i - f(x_i)\rvert \le \delta_m$ $\delta_m \text{sign}[y_i - f(x_i)]$ for $\lvert y_i - f(x_i)\rvert > \delta_m$ where $\delta_m = \alpha\text{th-quantile}\{\lvert y_i - f(x_i)\rvert\}$
Classification	Deviance	kth component: $I(y_i = \mathcal{G}_k) - p_k(x_i)$

T. Hastie et al.

Classification Example

- XGBoost Algorithm trained on 1000 trees, Depth=4 ,learning rate = 0.01, with bagged boost.
- Signal: tth, h → di-tau (with SS leptons), bkg: ttbar

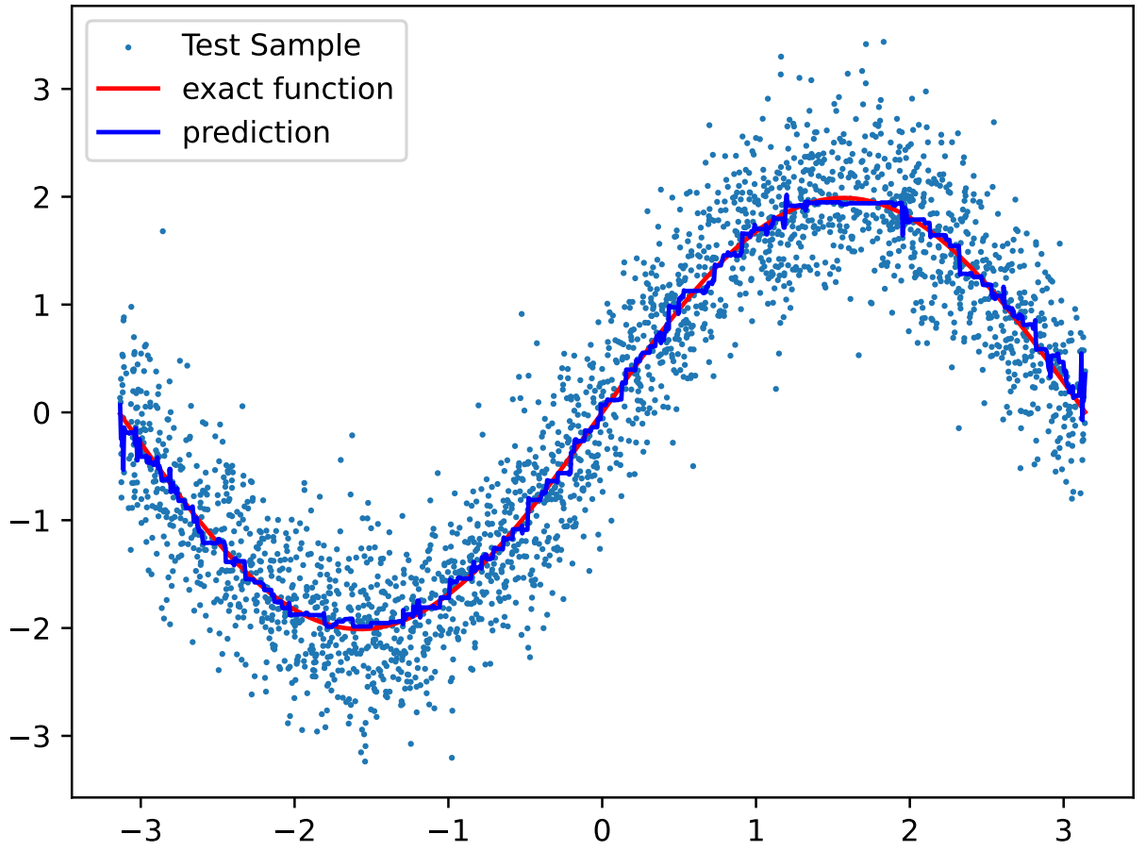




Regression Example

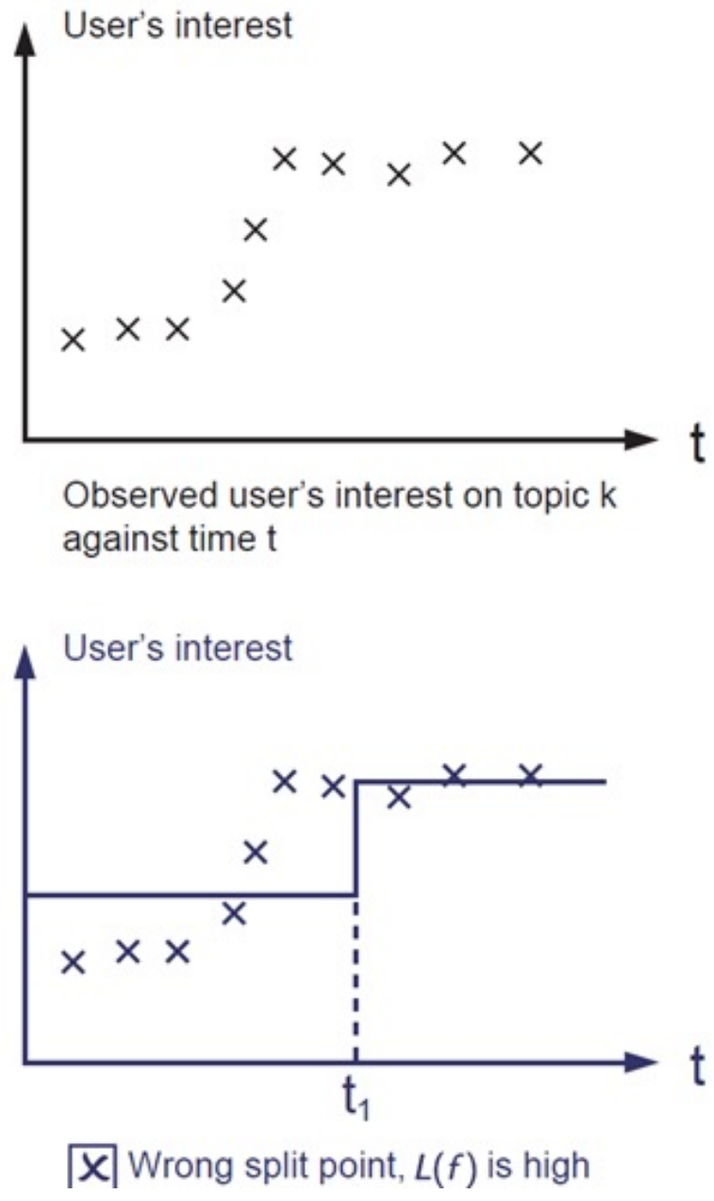
- GradientBoostingRegressor in scikit-learn, trained with 1000 trees, Depth=3 ,learning rate = 0.01
- 10000 events (5K train + 5K test) generated with a function

  2*sin(x)*exp(-x/300.)+random.normal(0,0.5,*x.shape)



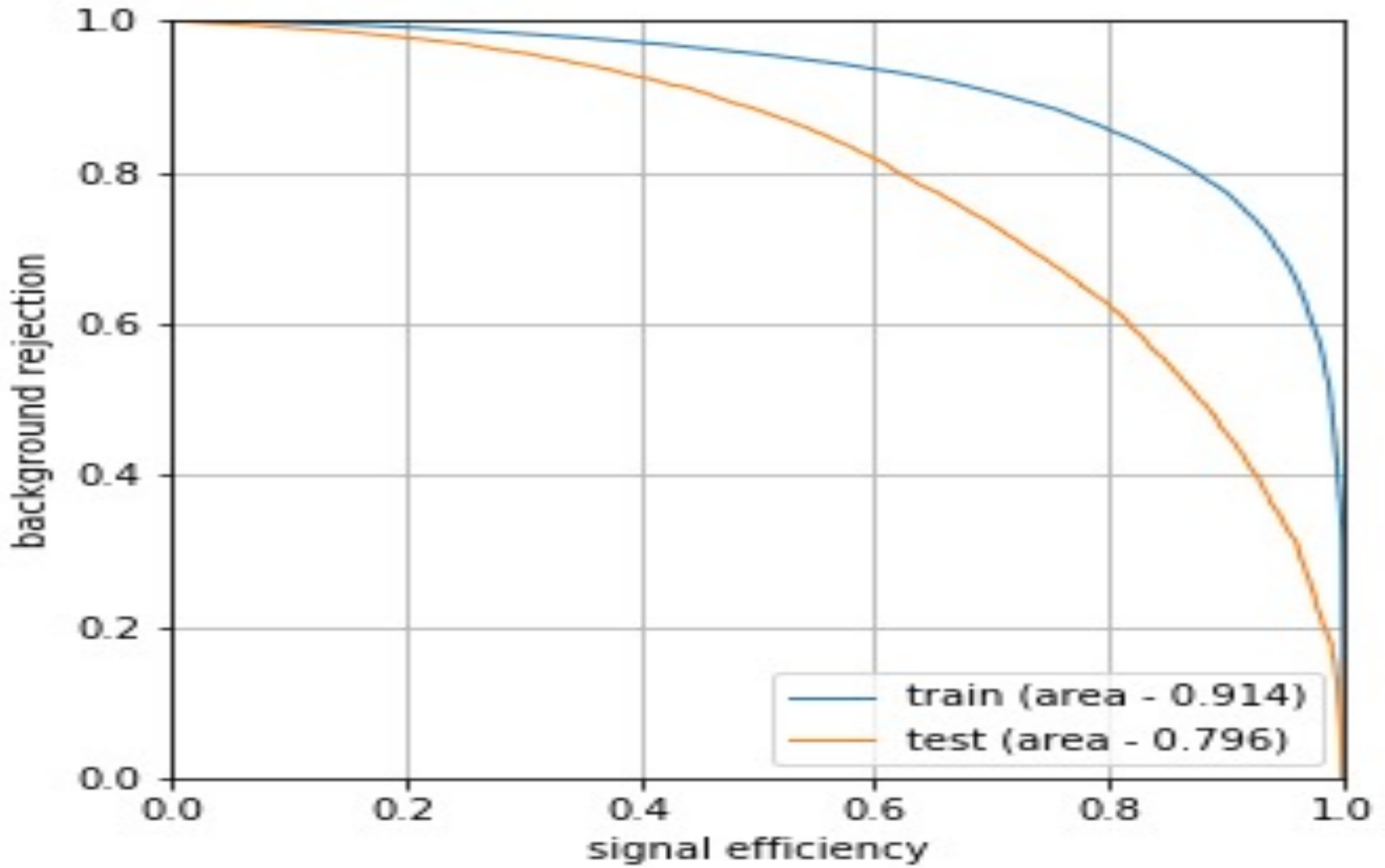
Regularization

- Fitting the training data too well can lead to overfitting and degrade future predictions
- The regularization controls the complexity of the model, which helps us to avoid overfitting



(ref: XGBoost userguide)

Overtraining example



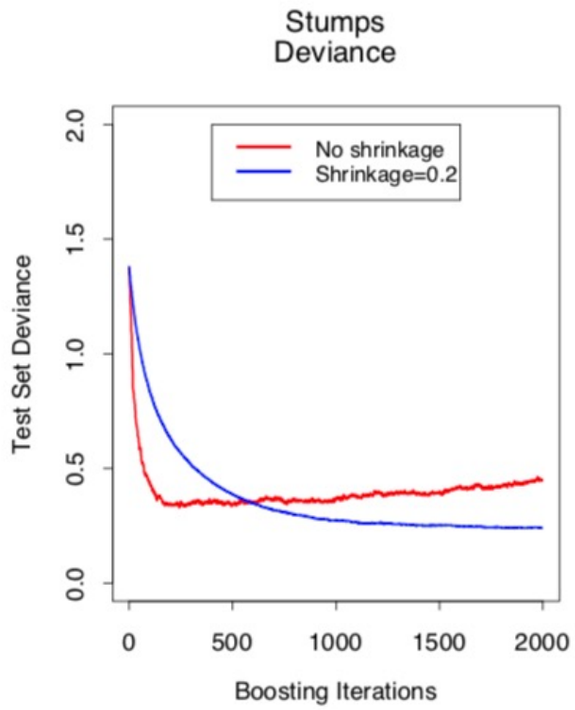
Regularization

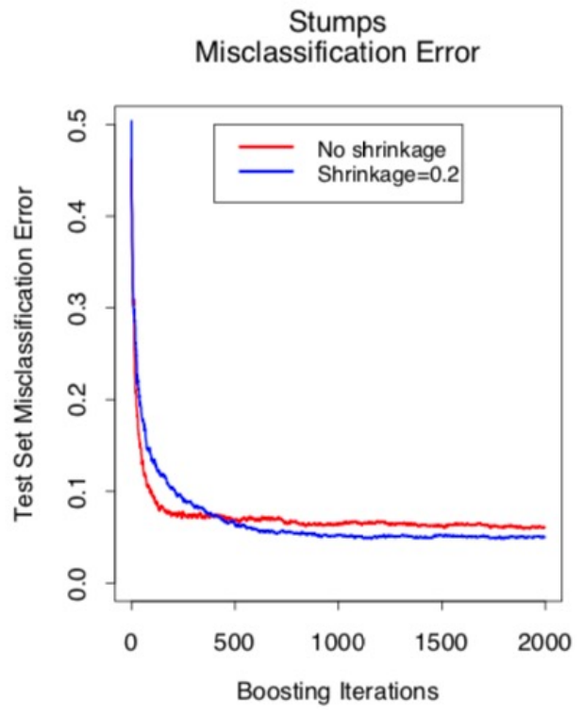
There are various parameters of BDTs that can be optimized to reduce overtraining

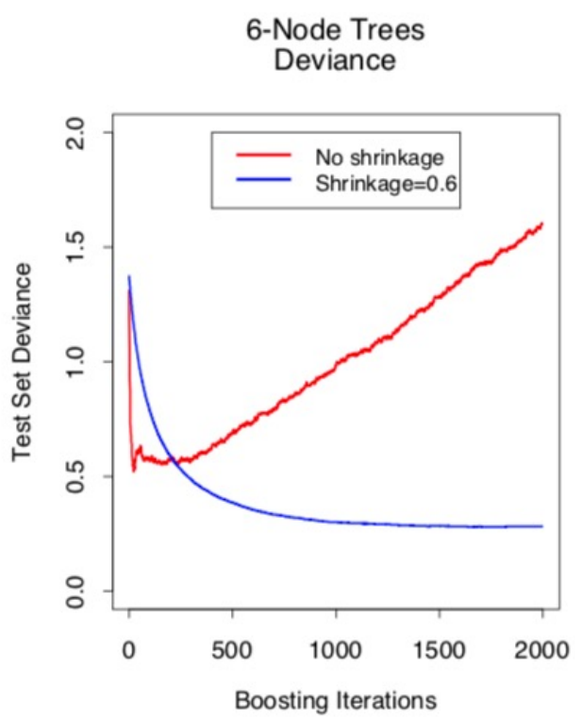
- Number of Boosting Iterations (M_{tree}):
  - Having large enough M_{tree} can make training loss arbitrarily small, but can lead to large overtraining problem
  - One can monitor the prediction loss on a validation sample as function of M_{tree} and find an optimal value for M_{tree} for a given training sample (similar to early stopping in NN)

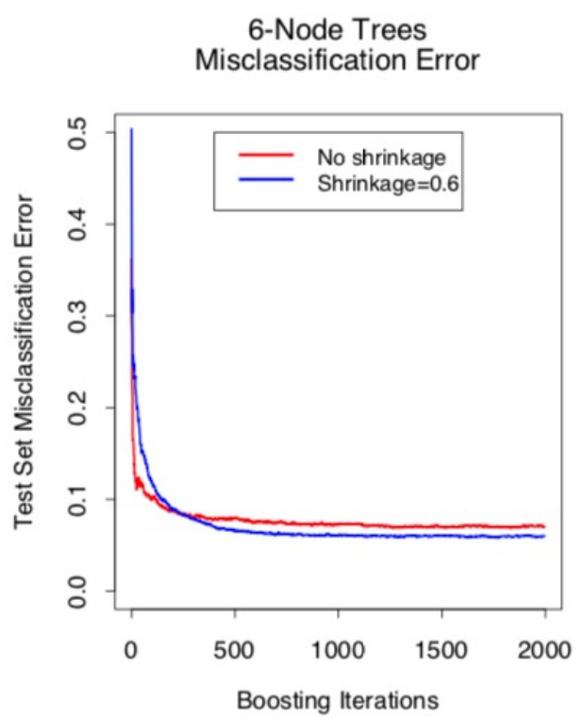
- Shrinkage (learning rate):
  - Scale the contribution of each tree by a factor $0 < \eta < 1$
  - Controls the learning rate of the boosting procedure
  - η and Mtree are related: small η ➔ large M_{tree} and vice-versa
  - Typical values of $\eta < 0.1$

Shrinkage









Ref: Fig.10.11

T. Hastie et al.

Regularization

- Subsampling (Stochastic Gradient Boosting):
  - Introduce a resampling procedure using random subsamples of the training events for growing trees – called "bootstrap averaging" or "Bagging"
  - The sample fraction used in each iteration can be controlled through a parameter, typical values to get best results are 0.5—0.8.
  - Stability against statistical fluctuations

Stochastic Gradient Boosting

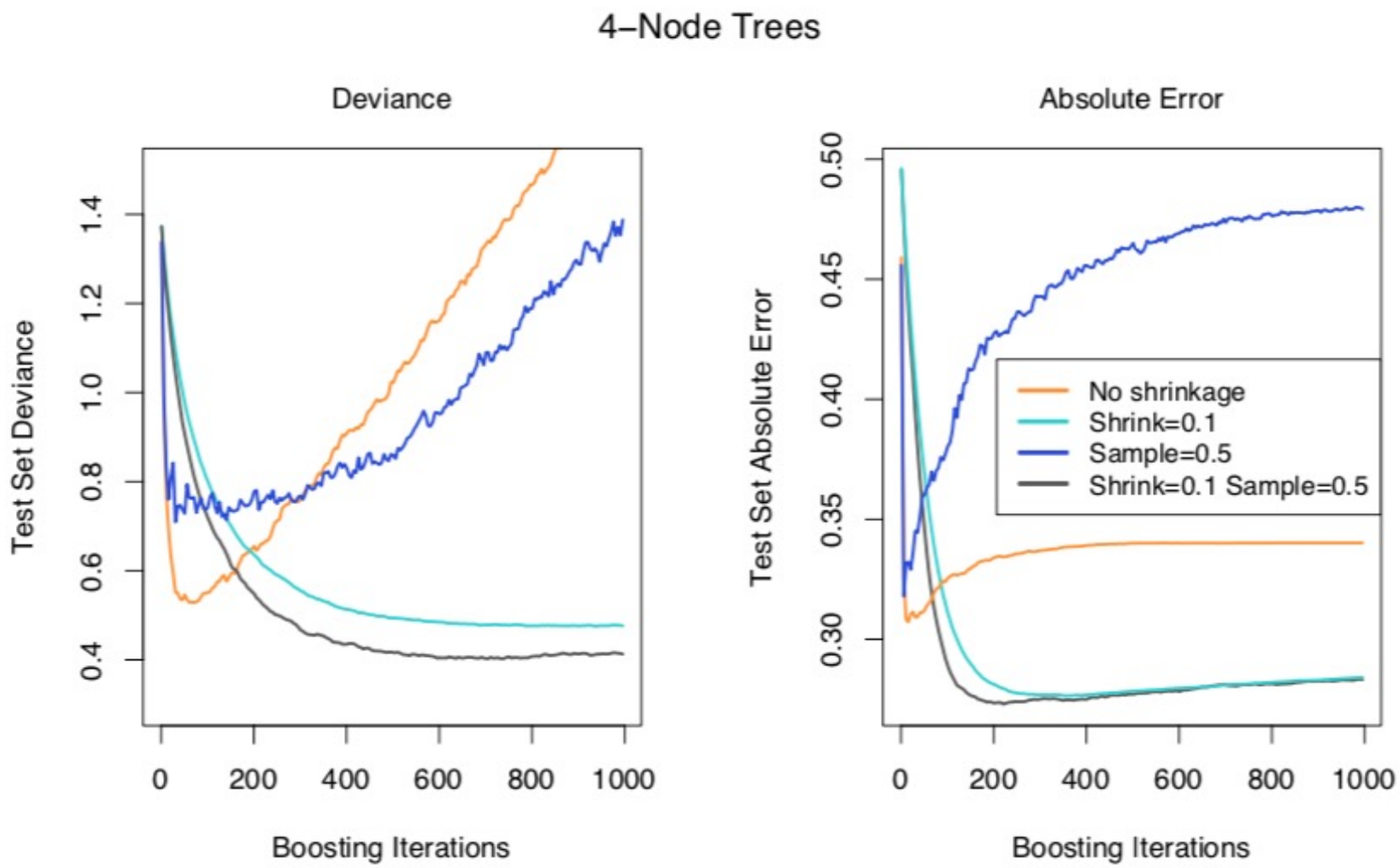


FIGURE 10.12. *Test-error curves for the simulated example (10.2), showing the effect of stochasticity. For the curves labeled "Sample= 0.5", a different 50% subsample of the training data was used each time a tree was grown. In the left panel the models were fit by* `gbm` *using a binomial deviance loss function; in the right-hand panel using square-error loss.*

T. Hastie et al.

Regularization

- Add regularization term to the objective function (it penalizes the complexity of the k_{th} tree)

$$obj = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

In XGBoost, the tree complexity function is defined as:

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} \omega_j^2$$

ω is the vector of scores on leaves

T is the number of leaves

Variable Selection

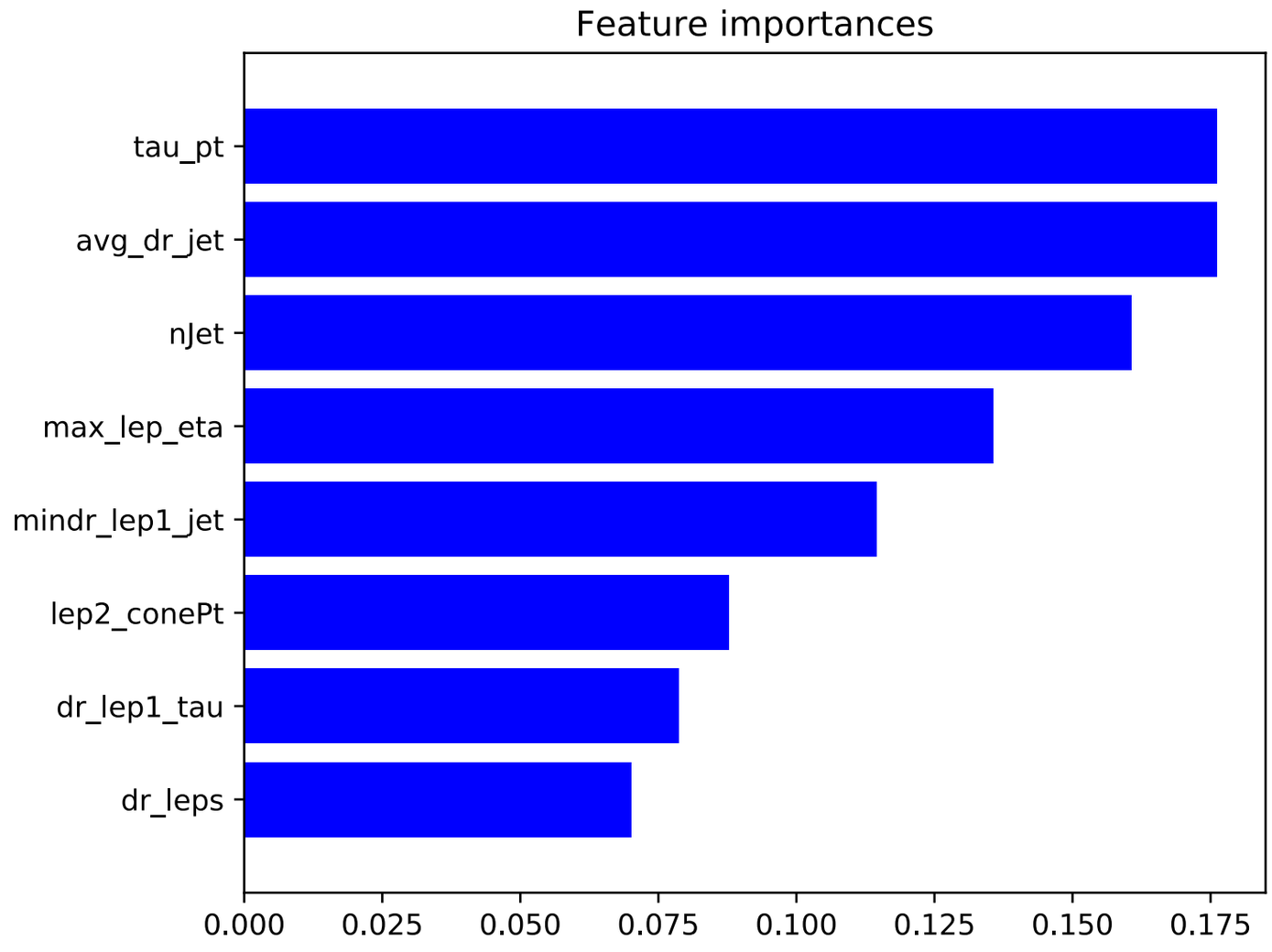
- In general it is good to check the data/mc agreement of a variable before using in any MVA methods
- BDTs can handle large number of input variables, but the hyper-parameters need to be optimized to get best out of them, and also large number of training events may be needed.
- This might increase the CPU processing time
- BDTs are immune to duplicate variables. The sorting of events according to them would be identical, leading to same tree
- If variables are not discriminating, they are typically ignored.
- Unlike NNs, it is not necessary to transform (scale) the variables to certain range. Since the separation by split on a variable and its transformed variable would be almost same.

Relative Importance of Input Variables

The relative importance of an input variable in BDT is derive by counting the number of times the variable is used to split decision tree nodes and by weighting each split occurrence by the separation gain-squared it has achieved and by the number of events in the node

However, variable ranking should not be taken at face value, because of possibility variable masking.

e.g. variable x_j may be little worse than x_i, but end up never being picked in the decision tree growing process, and so, ranked as irrelevant. So, you may conclude it has no discriminating power



Tools

TMVA:

Based on ROOT framework (available directly in ROOT package)

Many MVA methods implemented including ANNs.

BDT: with BoostType “AdaBoost, Grad, Bagging”

https://root.cern.ch/download/doc/tmva/TMVAUsersGuide.pdf

Scikit-Learn:

Based on python. Supports binary and multiclass classification

BDT methods: DecisionTreeClassifiers, AdaBoostClassifier,

GradientBoostingClassifier, BaggingClassifier, RandomForestClassifier,

SGDClassifier

https://scikit-learn.org/stable/documentation.html

Tools

XGBoost (eXtreme Gradient Boosting):

It is an implementation of gradient boosted decision trees designed for speed and performance

The implementation of the model supports the features of the scikit-learn

Models supported: Gradient Boosting, Stochastic Gradient Boosting, Regularized Gradient Boosting

Some system Features:

Parallelization of tree construction using all of your CPU cores during training.
Distributed Computing for training very large models using a cluster of machines.
Out-of-Core Computing for very large datasets that don't fit into memory.
Cache Optimization of data structures and algorithm to make best use of hardware.

https://xgboost.readthedocs.io/en/latest/

https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf

arXiv:1603.02754v3 [cs]

References

1. The elements of statistical learning, Hastie, Tibshirani, Friedman
2. Yann Coadou, EPJ Web of Conferences 55, 02004 (2013)
3. TMVA userguide
4. XGBoost userguide

Thanks

Installation Guide

We will install all packages using ANACONDA (Python, Numpy, Scipy, Pandas, ROOT/TMVA, Scikit-Learn, XGBoost etc..)

Download and install anaconda for Python 3 version, according to your operating system and machine architecture:
https://www.anaconda.com/distribution/

Go to a terminal and activate anaconda environment using command
conda activate

Then Install XGBoost using command:
conda install -c conda-forge xgboost

Install root_numpy:
conda install -c conda-forge root_numpy

Then, check if ROOT is automatically installed using the command:
root (it should create root prompt)

Otherwise, install ROOT:
conda install -c conda-forge root

Download data files from this location

https://www.dropbox.com/sh/zc6r87qnn1y284a/AADzktHKBG5NVlJbkiW9FJrRa?dl=0

Gradient Boosting in TMVA

- Current TMVA implementation uses the binomial log-likelihood loss:

$$L(F, y) = \ln\left(1 + e^{-2F(\mathbf{x})y}\right)$$

- Minimization is performed using steepest-descent approach
- Implementation in TMVA: Calculate the current gradient of the loss function and then grow a regression tree whose leaf values are adjusted to match the mean value of the gradient in each region defined by the tree structure – www.jstor.org/stable/2699986
- Iterating this procedure yields the desired set of decision trees which minimizes the loss function
- Robustness can be enhanced by reducing the learning rate of the algorithm (shrinkage), which controls the weight of the individual trees

(ref: TMVA userguide)

Tree Boosting in XGBoost

Procedure:

- Each tree is created iteratively
- The tree's output ($f(x)$) is given a weight (w) relative to its accuracy
- Events which are misclassified, increase their weights
- Build a new tree, repeat the procedure for several trees
- The final score of an event is the weighted average of scores from all trees

$$\hat{y} = \sum_k w_k f_k(x), \quad f_k \in F \longrightarrow$$

Space of functions containing all regression tree

- *This means that samples that are difficult to classify receive increasing larger weights until the algorithm identifies a model that correctly classifies these samples*
- The goal is to minimize an objective function

$$obj = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

$l(\hat{y}_i, y_i)$ is the loss function (distance between truth and prediction value for i_{th} sample)

$\Omega(f_t)$ is the regularization function (it penalizes the complexity of the k_{th} tree)

Gradient Boosting: How does it learn?

Objective function: $$obj = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

What are the parameters of the trees?

- What we need to learn are these functions
- Contains the structure of the tree and the leaf scores f_k
- Instead of learning weights, we are learning functions (trees)

It is not easy to use steepest decent method to find f (since these are trees, instead of just numerical vectors)

Additive Training:

Start from a constant prediction and add one new tree a time

$$\hat{y}_i^{(0)} = 0$$

$$\hat{y}_i^{(1)} = f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i)$$

$$\hat{y}_i^{(2)} = f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i)$$

.....

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)$$ ← Add new function

Model at training round t

Keep functions in previous round

Additive Training

Which tree do we want at each step?

A natural thing is to add the one that optimizes our objective.

$$Obj^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{k=1}^{t} \Omega(f_k)$$

$$= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + constant$$

Goal is to find f_t that minimize it.

Consider square loss

$$Obj^{(t)} = \sum_{i=1}^{n} \left(y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)) \right)^2 + \Omega(f_t) + constant$$

$$= \sum_{i=1}^{n} \left(2(\hat{y}_i^{(t-1)} - y_i) f_t(x_i) + f_t(x_i)^2 \right) + \Omega(f_t) + const.$$

Residual from previous round

For other losses of interest (for example, logistic loss), it is not so easy to get such a nice form. So in the general case, the Taylor expansion of the loss function up to the second order is considered.

Additive Training

Taking the Taylor expansion of the loss function and keeping up to 2nd order

$$Obj^{(t)} = \sum_{i=1}^{n}\left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) + const.$$

Where

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

$$h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

After removing all the constants, the specific objective at step t becomes

$$\sum_{i=1}^{n}\left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t)$$ ⟵ This becomes the optimization goal for the new tree

- g_i and h_i comes from definition of loss function
- The learning of function only depend on the objective via g_i and h_i

This is how XGBoost supports custom loss functions

Tree Complexity

One can refine the definition of the tree

$$f_t(x) = \omega_{q(x)}$$

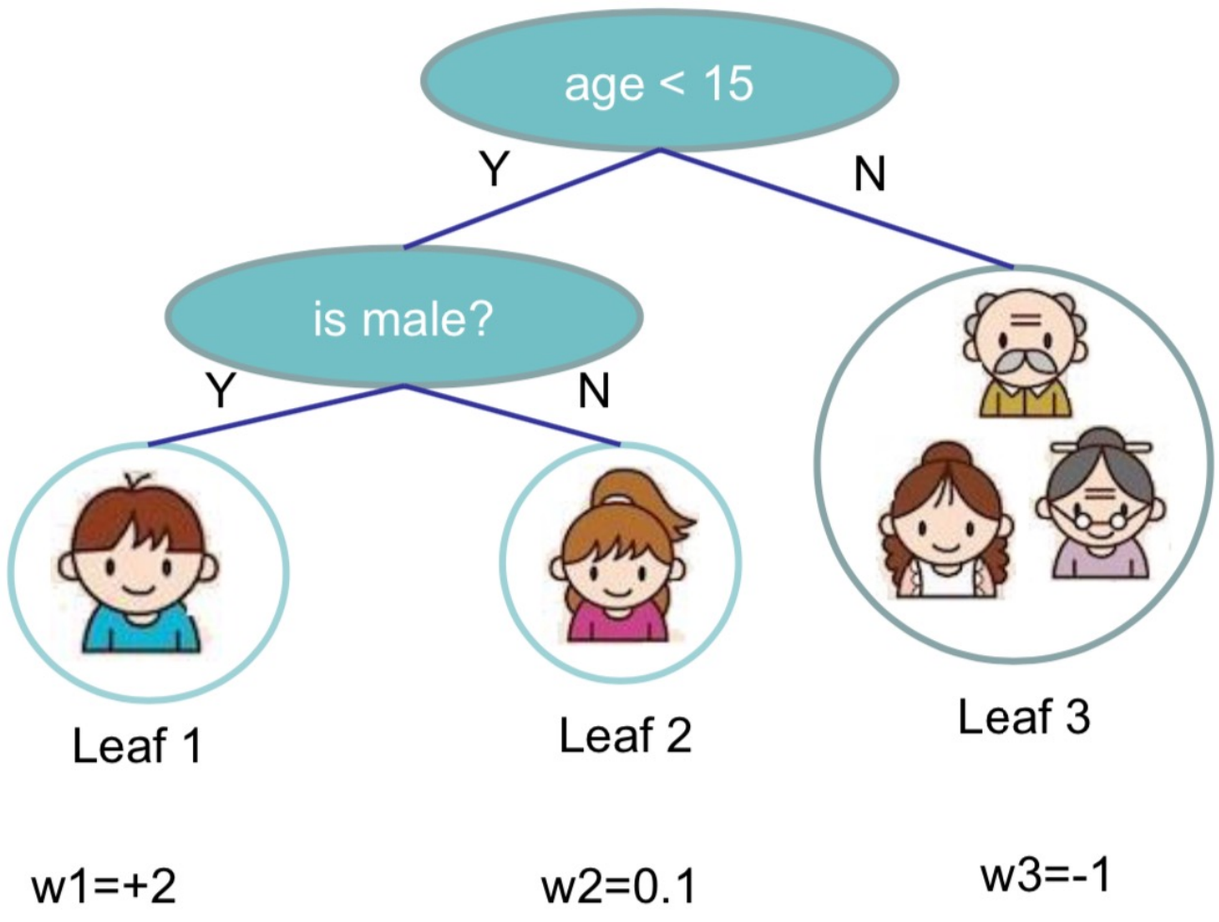
In XGBoost, the complexity is defined as

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} \omega_j^2$$

ω is the vector of scores on leaves

q(x) is a function assigning each data point to the corresponding leaf.

T is the number of leaves



$$\Omega = \gamma 3 + \tfrac{1}{2}\lambda(4 + 0.01 + 1)$$

The Structure Score

One can write the objective value with the t_{th} tree as

$$Obj^{(t)} \approx \sum_{i=1}^{n}\left[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)\right] + \Omega(f_t)$$

$$= \sum_{i=1}^{T}\left[g_i \omega_{q(x_i)} + \frac{1}{2}h_i \omega_{q(x_i)}^2\right] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T}\omega_j^2$$

$$= \sum_{i=1}^{T}\left[(\sum_{i\in I_j} g_i)\omega_j + \frac{1}{2}(\sum_{i\in I_j} h_i + \lambda)\omega_j^2\right] + \gamma T$$

where $I_j=\{i|q(x_i)=j\}$ is the set of indices of data points assigned to the j_{th} leaf

Compressing further, by defining $G_j=\sum_{i\in Ij} g_i$ and $H_j=\sum_{i\in Ij} h_i$

$$Obj^{(t)} \approx \sum_{i=1}^{T}\left[G_j\omega_j + \frac{1}{2}(H_j+\lambda)\omega_j^2\right] + \gamma T$$

ω_j are independent with respect to each other

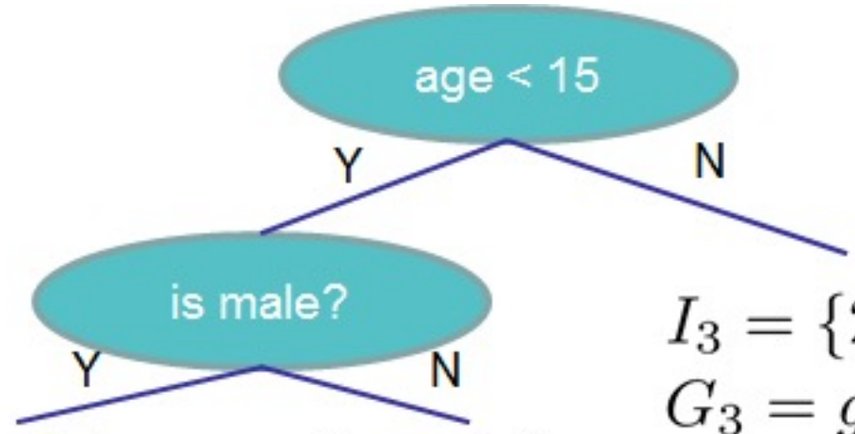
Since it is quadratic, the best ω_j for a given structure $q(x)$ and the best objective reduction one can get is:

$$\omega_j^* = -\frac{G_j}{H_j+\lambda} \qquad \text{and} \qquad Obj^* = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j+\lambda} + \gamma T$$

The Structure Score

This is a simple pictorial example

Instance index		gradient statistics
1		g1, h1
2		g2, h2
3		g3, h3
4		g4, h4
5		g5, h5



$$I_1 = \{1\} \qquad I_2 = \{4\} \qquad I_3 = \{2, 3, 5\}$$
$$G_1 = g_1 \qquad G_2 = g_4 \qquad G_3 = g_2 + g_3 + g_5$$
$$H_1 = h_1 \qquad H_4 = h_4 \qquad H_3 = h_2 + h_3 + h_5$$

$$Obj = -\sum_j \frac{G_j^2}{H_j + \lambda} + 3\gamma$$

The smaller the score is, the better the structure is

Learning the Tree Structure

Ideally one would enumerate all possible trees and pick the best possible one
(by computing the tree structure score and optimal leaf weight)
However, there can be infinite possible tree structures.
Thus, in practice, one tries to optimize one level of the tree at a time.

The idea is:

- Start from a tree with depth 0
- For each leaf node, split the leaf into two leaves and compute the gain in the objective score:

$$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} + \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma$$

The complexity cost by introducing additional leaf

the score of left child

the score of right child

the score of parent

Important fact: if the gain is smaller than γ, it would be better not to add that branch

To search for an optimal split and to do it efficiently, place all the instances in sorted order, like in this picture