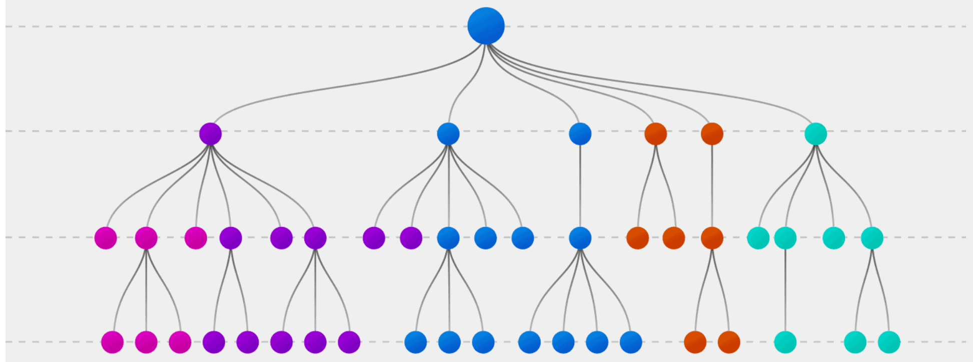
Supervised Learning: ”basic” learners: Decision Trees Classifiers and Regression Trees



We start by IGNORING the data complexity issue

- Let us assume our data are STRUCTURED (we have tables with either continuous or symbolic feature values)
- Let’s assume we have «good enough» and sufficient structured data
- Furthermore, we restrict to **supervised predictive tasks**:
    - Data are feature-value tables, and the task is to learn predicting the value of a specific feature y (symbolic or continuous)
    - We are provided with historical data, i.e. we can observe examples $\mathbf{x} \in \boldsymbol{X}$ of past data for which the value of the feature y to be predicted is known. We denote with D: <**x**,y> the training set of input-output tuples extracted from historical data
    - Task is to learn a model f(**x**) and use this model to predict y=f(**x**) for unseen values of new instances

Outline

- Formal definition of Classifiers and Regressors
- Tabular and geometric representation of data
- Basic classifiers:
    - Decision Trees
    - Regression trees
    - Fine-tuning the tree

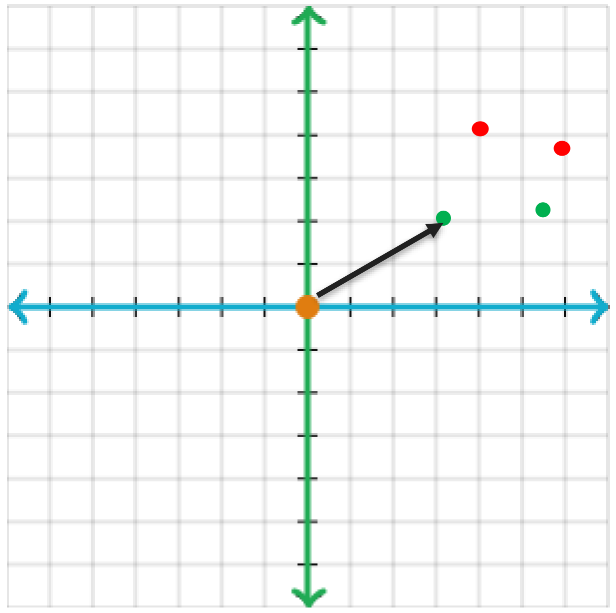
Supervised Classifiers and Regressors: definition of the task

- Let D: <$\mathbf{x}_i$,y_i> be a dataset representing historical data about a given domain
- Let's assume that D is a list of records, called *instances or samples or examples* $\mathbf{x}_i$: (x_{i1}…x_{in})
- Every x_{ij} represents the **value** of an attribute Xj, or *feature*, describing the istance (values can be discrete or continuous)
- Y is the **target** attribute for which we want to learn a prediction model f(**x**)
- In classification tasks, the target is a **discrete variable** (binary or multi-valued), called the «class» C so the training set D includes tuples $< \boldsymbol{x}_i, c_i >$
- In **classifiers**, the prediction model to be learned is often denoted as c(**x**) rather than f(**x**)

How do we represent our structured data D?

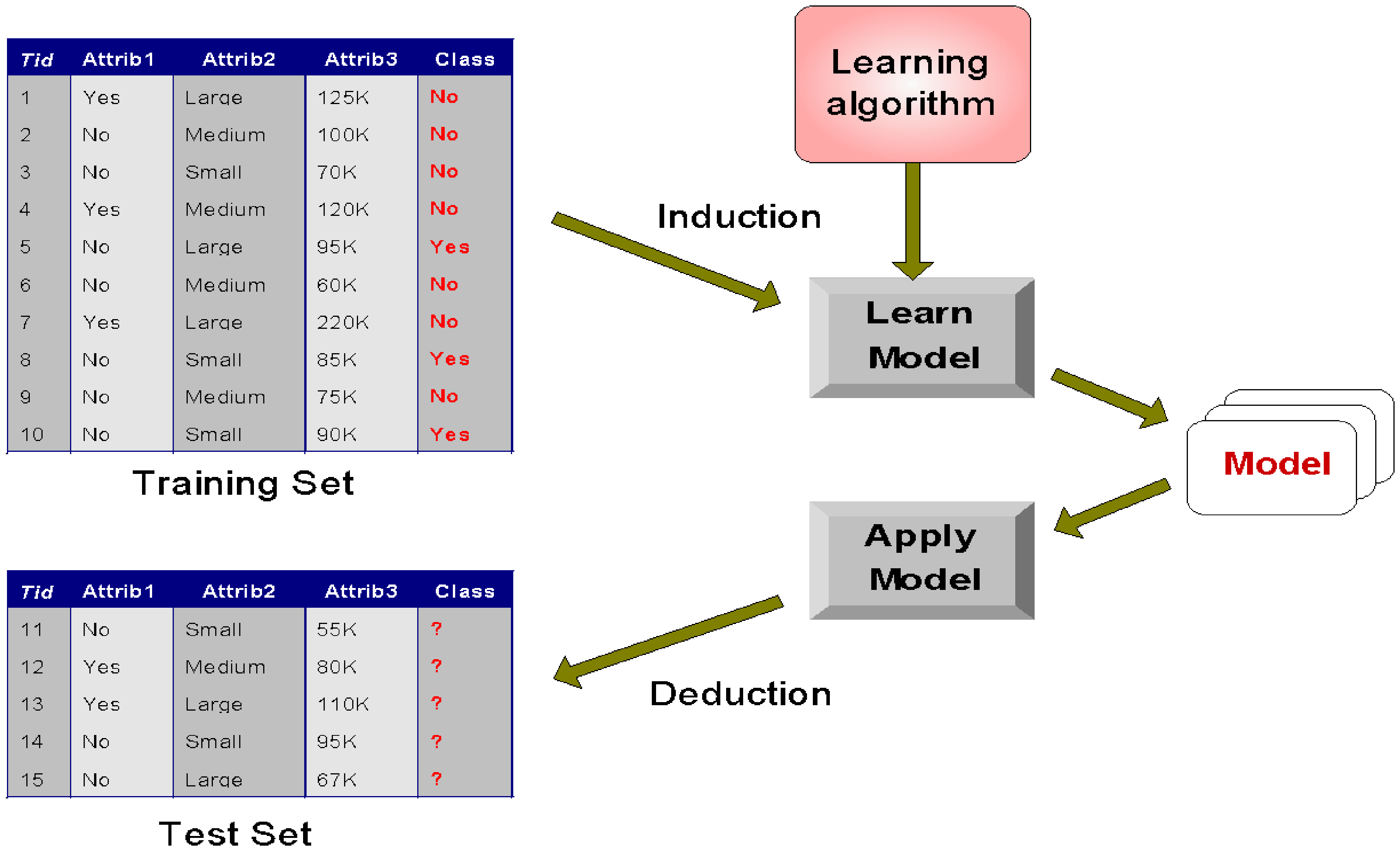
- **Tabular and geometric representation.** Points represent the rows (instances) and the colour is the value of the class variable (here, y=red n=green)

example	age	Bodymass index	Risk of a cardiac event?
1	32	20	n
2	60	35	y
3	55	21	n
4	40	40	y



- Rows in the table (array) are denotes as **feature vectors**
- So, by **x** (bold) we denote a feature vector whose dimensions are the features and whose coordinates are the feature values (e.g., the example 1 is geometrically represented by the vector **x1**: [32,20])
- Note that you can represent also symbolic features in the same way, but you need to «impose» an order to the symbolic values (e.g. *green, red, blue* if feature is «colour»), which might create problems with some algorithm (will see).

Illustrating the Classification Task



Tid	Attrib1	Attrib2	Attrib3	Class
1	Yes	Large	125K	No
2	No	Medium	100K	No
3	No	Small	70K	No
4	Yes	Medium	120K	No
5	No	Large	95K	Yes
6	No	Medium	60K	No
7	Yes	Large	220K	No
8	No	Small	85K	Yes
9	No	Medium	75K	No
10	No	Small	90K	Yes

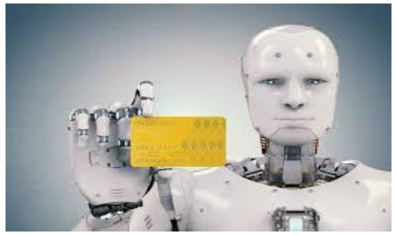
Training Set

Tid	Attrib1	Attrib2	Attrib3	Class
11	No	Small	55K	?
12	Yes	Medium	80K	?
13	Yes	Large	110K	?
14	No	Small	95K	?
15	No	Large	67K	?

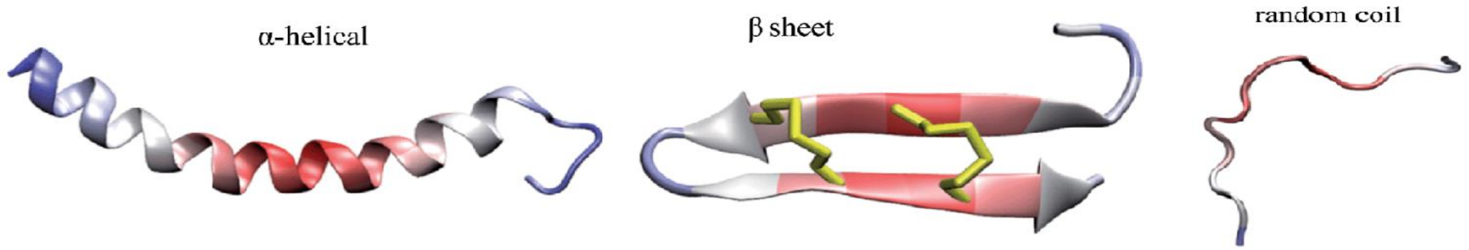
Test Set

Examples of Classification Task

- Predicting tumor cells as benign or malignant
(class is, e.g., *benign(x)* with values *pos* and *neg*)
- Classifying credit card transactions
as legitimate or fraudulent (class is, e.g. fraudulent(x), with values pos and neg)



- Classifying secondary structures of protein
as alpha-helical, beta-sheet, or random
coil



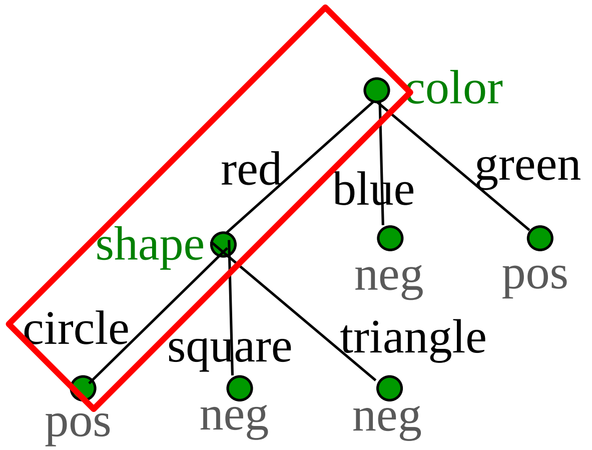
- Categorizing news stories as finance,
weather, entertainment, sports, etc (class is *category(x)* with values weather, sport,finance..)

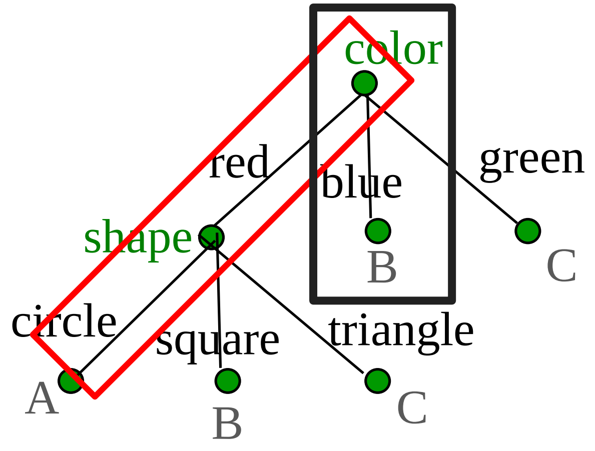
Decision trees

- Note: although Dtrees is a very old class of algorithms (on the other side, NN are also very old) it is at the basis of top performance non-deep Ensamble algorithms, such as Random Forest and Gradient Boosting –will se later
- These algorithms are often used as baselines, and in specific conditions still turn out to be competitive vrs deep methods.

Decision Trees

- The model output is a tree structure. **Every node** represents a test on a feature's value. Nodes are labelled with the feature name. Below each node there is one branch for each possible value of the feature. Branches are labelled with the value of the feature.
- **Leaf** nodes are "decisions", they specify the class label.



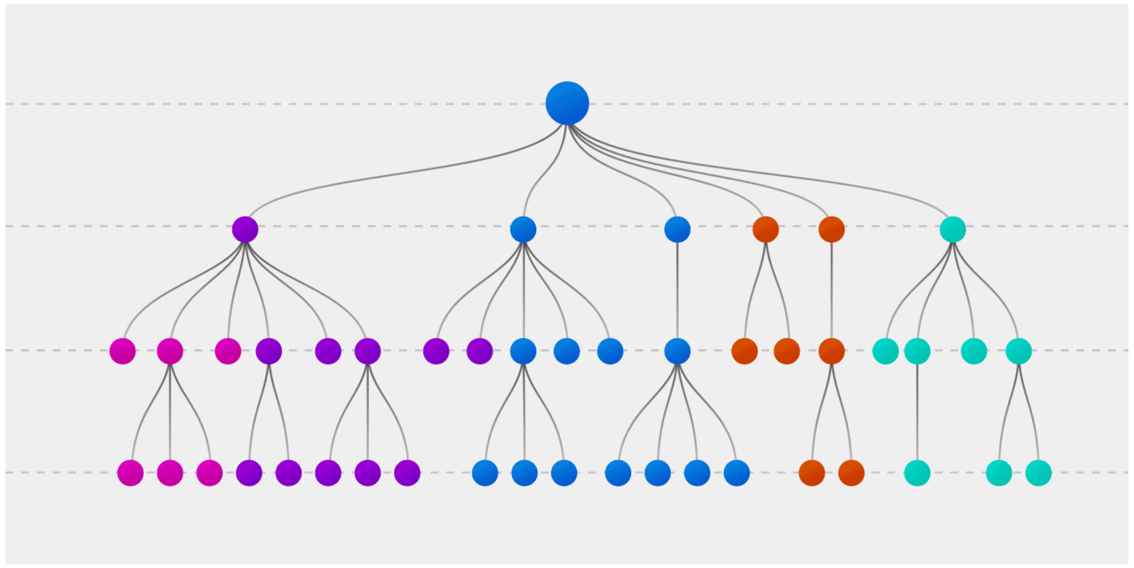


- A decision tree can represent **any boolean function** c(x), i.e., a *classification function over discrete-valued feature vectors*.
- **The tree can be rewritten as a set of rules, i.e. disjunctive normal form** (DNF). Example (for the left tree):
  - ➢ red ∧ circle ➔ pos
  - ➢ red ∧ circle ➔ A

    blue ➔ B; red ∧ square ➔ B

    green ➔ C; red ∧ triangle ➔ C

General shape of a decision tree

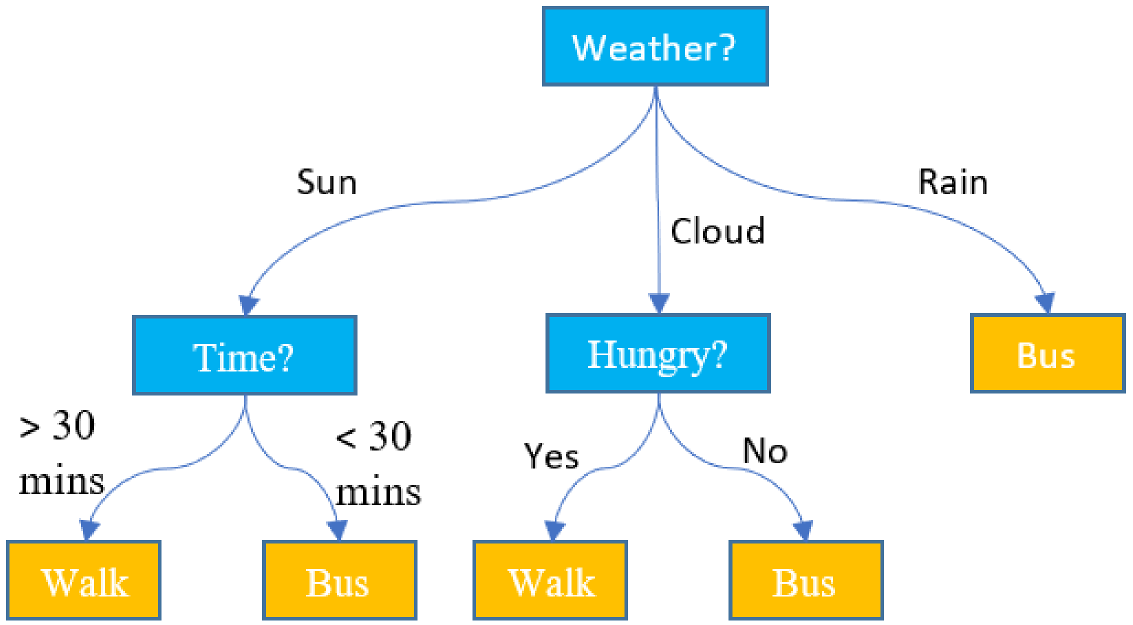


- ➢ Every test node is a **test** on the value (or range) of one features. For each possible outcome of the test, an edge is created that links to a subsequent test node or to a leaf node.
- ➢ Leaf nodes are **decisions** concerning the **value** of the classification.

Toy example

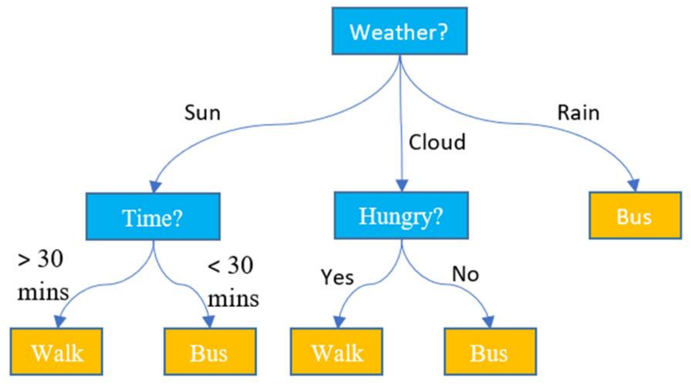
- Decision tree to decide whether to go home by bus or walking
- Binary class: values are either walk or bus
- Decision is taken based on the values of just 3 features: weather (W), time (T), and hungry (H)
- Features are either symbolic (sun-cloud-rain) or discretized (e.g., >30m or <30m)
- The decision tree can be re-written in terms of a decision function c(x) in first order logic:

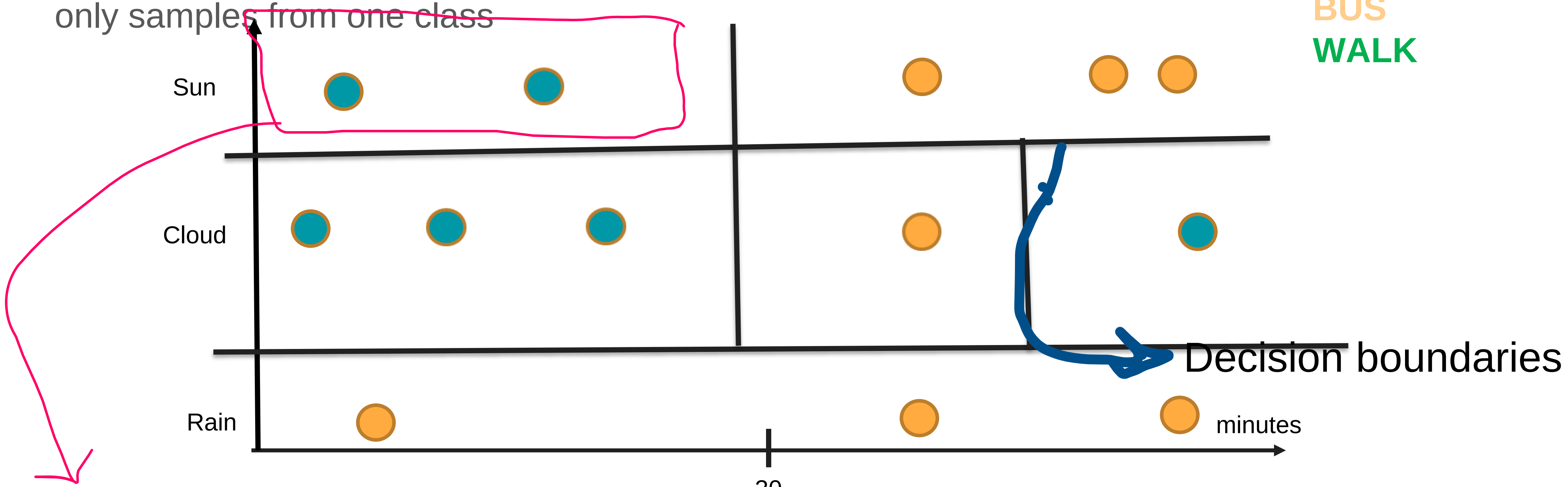
c(x(W,T,H))=(IF(W=Sun AND T>30) THEN Walk) OR (IF W=Sun AND T<30) THEN Bus) OR (IF W=Cloud AND H=Yes) THEN Walk) OR (etc. etc)



How do we learn a decision tree?

- We use historical data for which the value of the class is known (training set)
- The basic process is **greedy recursive partition** of the decision space into regions – **optimal** partition is one in which every region includes only samples from one class

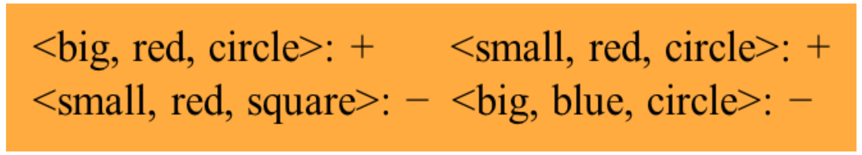




IF y= Sun AND 0<=x< 30 THEN BUS

Toy dataset

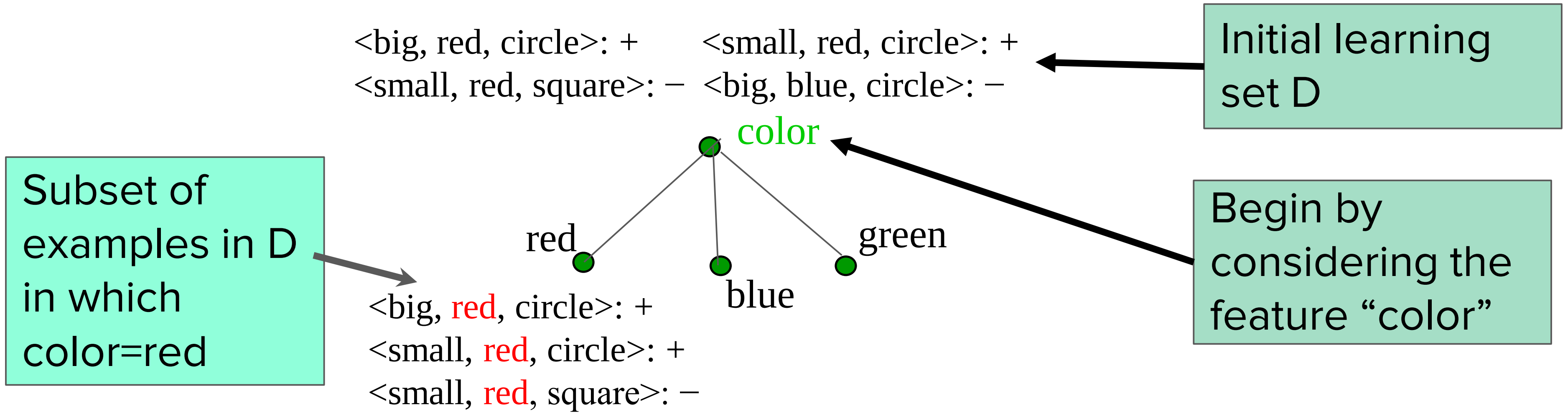
Dimension	Color	shape	class
Big	Red	Circle	Positive (+)
Small	Red	Circle	Positive (+)
Small	Red	Square	Negative (-)
Big	Blue	Circle	Negative (-)



<big, red, circle>: + <small, red, circle>: +
<small, red, square>: − <big, blue, circle>: −

How does it work: Top-Down Decision Tree Induction

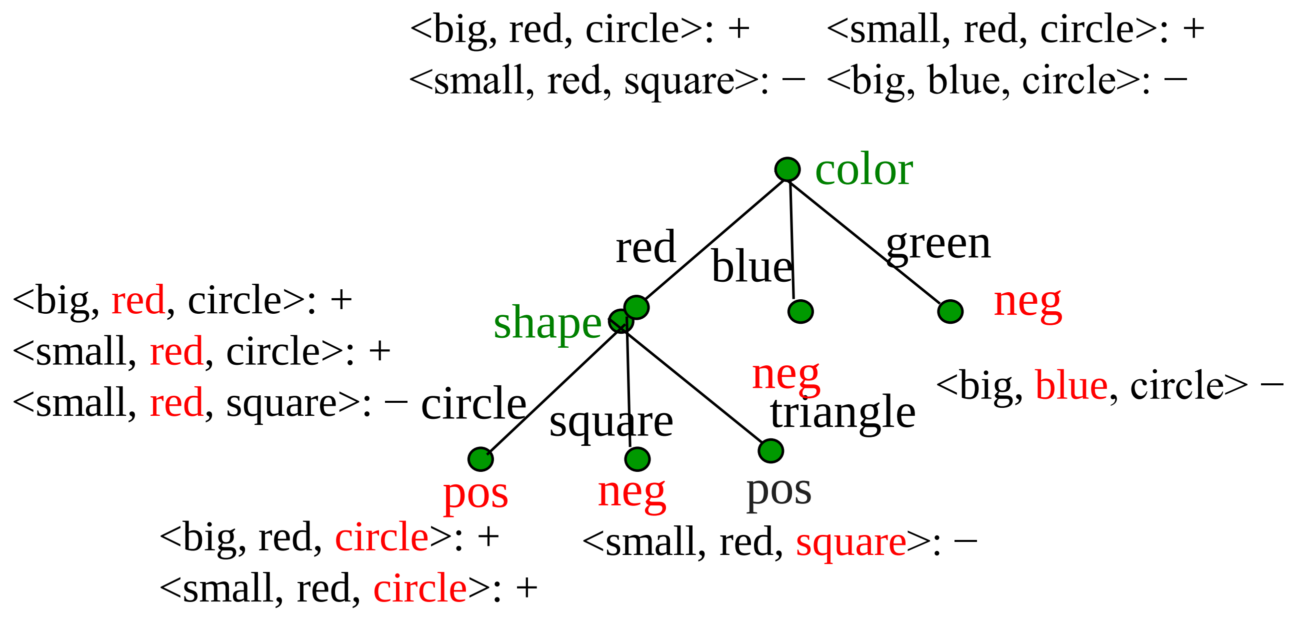
- Recursively build a tree top-down by **divide and conquer.**



At each step, we aim to find the "**best split**" of our data. What is a good split?
One which reduces the uncertainty of classification for "some" split!
Learning best splits (best ordering of tests on features, and best split over its values) is learning the DT **parameters.**

Top-Down Decision Tree Induction

- Recursively build a tree top-down by divide and conquer.

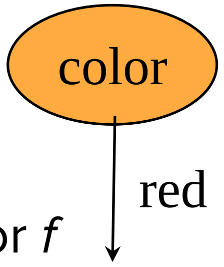


The process ends when we can output decisions (= the class labels), but: How do we decide the order in which we test attributes?
How do we decide the class of nodes for which we have no examples?

Let's ignore for now these 2 issues and describe the algorithm first

Decision Tree Induction Pseudocode

```
Algorithm DTree(examples D, features F) returns a tree:
        a) If all examples D are in one category, return a leaf node with that
        category label
        b) Else if the set of features F is empty, return a leaf node with the
        category label that is the most common in examples.
 Else pick a feature f in F and create a node R for it
          For each possible value $x_i$ of f:
                Let $s_i$ be the subset of examples that have value $x_i$ for f
                Add an outgoing edge E to node R labeled with the value $x_i$.
                If $s_i$ is empty
            then attach a leaf node to edge E labeled with the
                category that is the most common in examples
            else call DTree( $s_i$ , features – {f}) and attach the resulting tree as
            the subtree under edge E.
Return the subtree rooted at R.
```



a) and b) are the termination conditions

e.g. color=red

<big, red, circle>: +
<small, red, circle>: +
<small, red, square>: –

Example

1

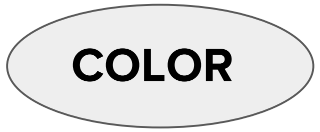
Instances:

<big, red, circle>: +	<small, red, circle>: +
<small, red, square>: −	<big, blue, circle>: −

Features:

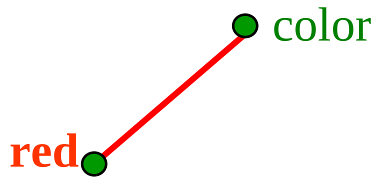
- **dimension, shape,color**

1. **Pick** a feature *f* and create a node *R* for it, eg. Color



2. **For each** possible value x_i of *f* ***(red for example)***:
    1. **Let** s_i be the subset of examples that have value v_i for *f*.
    2. **Add** an **outgoing** edge *E* to node *R* labeled with the valuex.
3. **if** (...) **else** call DTree(*example*(s_i) , *features* – {*f*}) and attach the resulting tree as the subtree under edge *E*.

<big, red, circle>: +
<small, red, circle>: +
<small, red, square>: −



call the algorithm on the subset for the feature red:

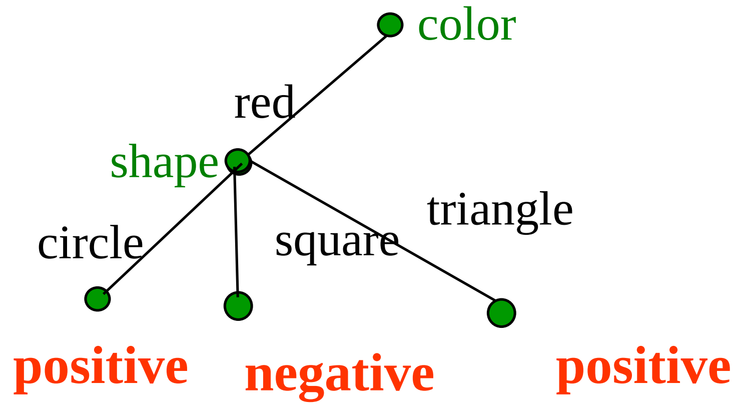
Dtree(<big, red, circle>: + <small, red, circle>: + <small, red, square>: − , <dimension,shape>)

Example

2

Dtree(<big, red, circle>: +
<small, red, circle>: +
<small, red, square>: − , <dimension,shape>)

4. Pick a feature *f* and create a node *R* for it, eg. **shape**



<big, red, circle>: +
<small, red, circle>: +

<small, red, square>: −

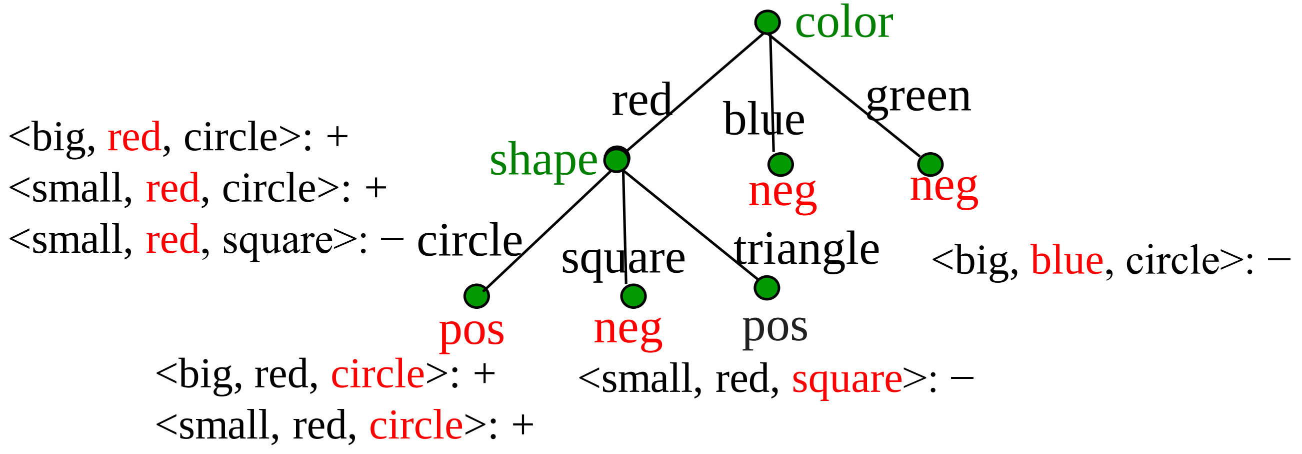
5. If all s_i **are in one category**, return a leaf node with that category label.

6. If s_i **is empty**. then attach a leaf node to edge *E* labeled with the category that is the most common in *examples*.

Example: 3

Backtrack to color (blue)

<big, **blue**, circle>: –



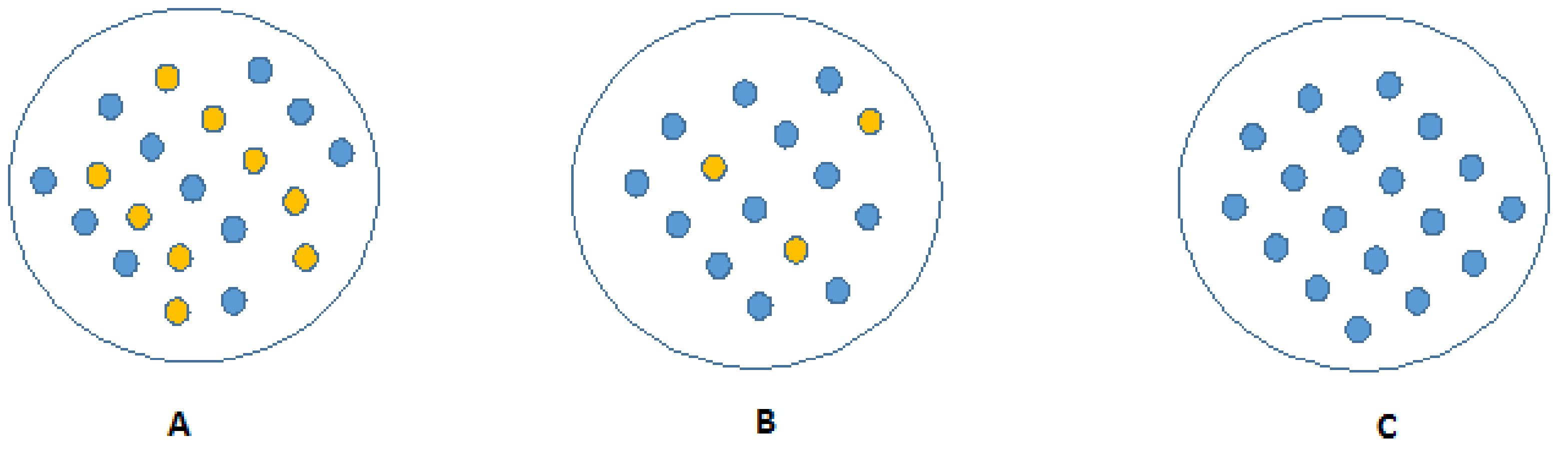
Now we know how to decide the class when we have no examples, but how do we decide the **order** in which we create nodes?

Picking a Good Split Feature

- The goal is to have the resulting tree be **as small as possible**, per Occam's razor.
- Finding a **minimal decision** tree (nodes, leaves, or depth) is an **NP-hard** optimization problem.
- The top-down divide-and-conquer method does a greedy search for a simple tree but does not guarantee to find the smallest.
    - The **general lesson in ML: "Greed is good."**
- Want to pick a feature that creates subsets of examples that are relatively "pure" in a single class so they are "closer" to being leaf nodes.
- There are a variety of methods for picking a good test, a popular one is based on information gain that originated with the ID3 system of Quinlan (1979). The choice of the method to be used is an hyperparameter of the DT model

About “purity”

- Look at the image below and think which group can be described easily. Intuitively, the answer is C because it requires less information, as all values are similar (=blue).
- On the other hand, B requires more information to describe it, and A requires the maximum information. In other words, we can say that C is a “pure” node, B is impure and A is more impure than B.



How to measure purity

- **Entropy** (disorder, impurity) of a set of examples, D, relative to **binary** classification is:

$$E(D) = -p_1 log_2(p_1) - p_0 log_2(p_0) = -p_1 log_2(p_1) - (1-p_1) log_2(1-p_1)$$

  - where p_1 is the fraction of positive examples in D and p_0 is the fraction of negatives. (Notice that if S is a **sample** of a population, *Entropy(S)* is an **estimate** of the population entropy).
- If all examples are in one category (as for node C of the previous example), entropy is zero (we define $0 \cdot \log(0)=0$).
- If examples are equally mixed (p_1=p_0=0.5), entropy is a **maximum** of 1.

Entropy and Binary Entropy

- Entropy can be viewed as *the number of bits* required on average to encode the class of an example in *D*. It is also an estimate of the initial "disorder" or "uncertainty" about a classification, given the set D.

- **General Formula of Entropy**: For multi-class problems with **C** category values, entropy generalizes to:

$$E(D) = -\sum_{i=0}^{C} p_i \log_{\underset{b}{2}}(p_i)$$

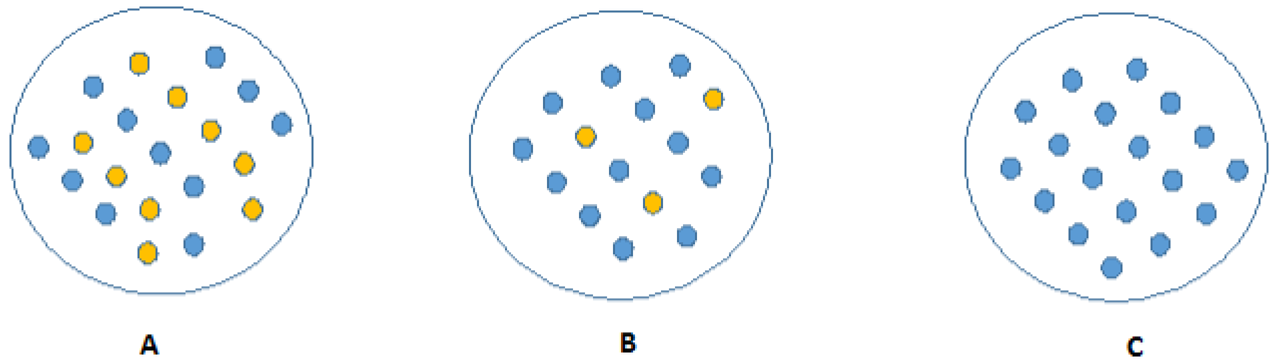
Example: Entropy Computation

- We have two class labels, blue and yellow.
- In group C, we have 18 objects, and they are all blue:

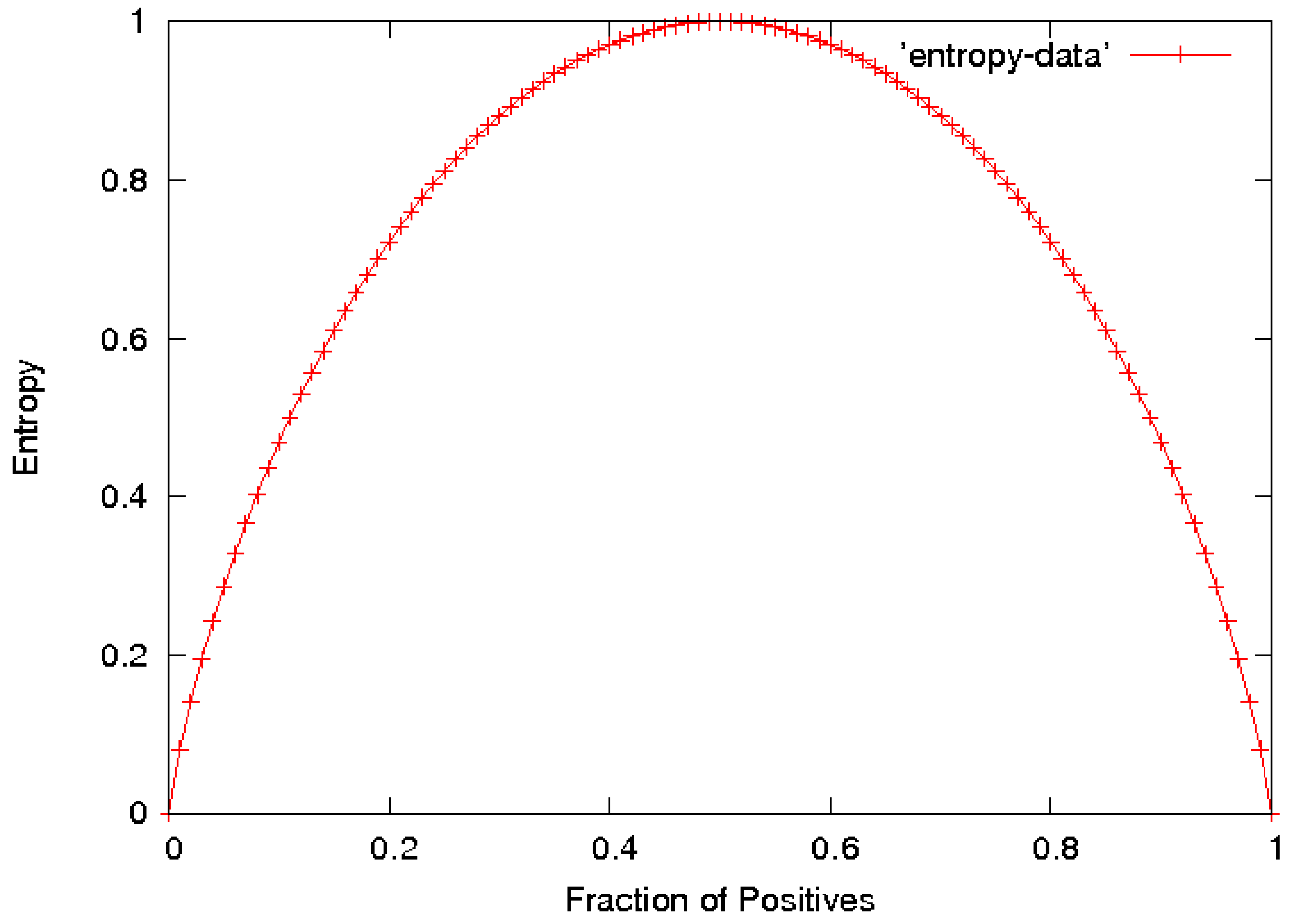
$$Entropy(C) = -p_{blue}\log(p_{blue}) - p_{yellow}\log(p_{yellow})$$
$$= -{}^{18}/_{18} \times \log(1) - {}^{0}/_{18} \times \log(0) = 0$$

- In group A, we have 20 objects, 9 are yellow, 11 are blue

$$Entropy(A) = -p_{blue}\log(p_{blue}) - p_{yellow}\log(p_{yellow})$$
$$= -{}^{11}/_{20} \times \log({}^{11}/_{20}) - {}^{9}/_{20} \times \log({}^{9}/_{20}) = 0.328 + 0.36 = 0.688$$



Entropy Plot for Binary Classification (only 2 class labels)



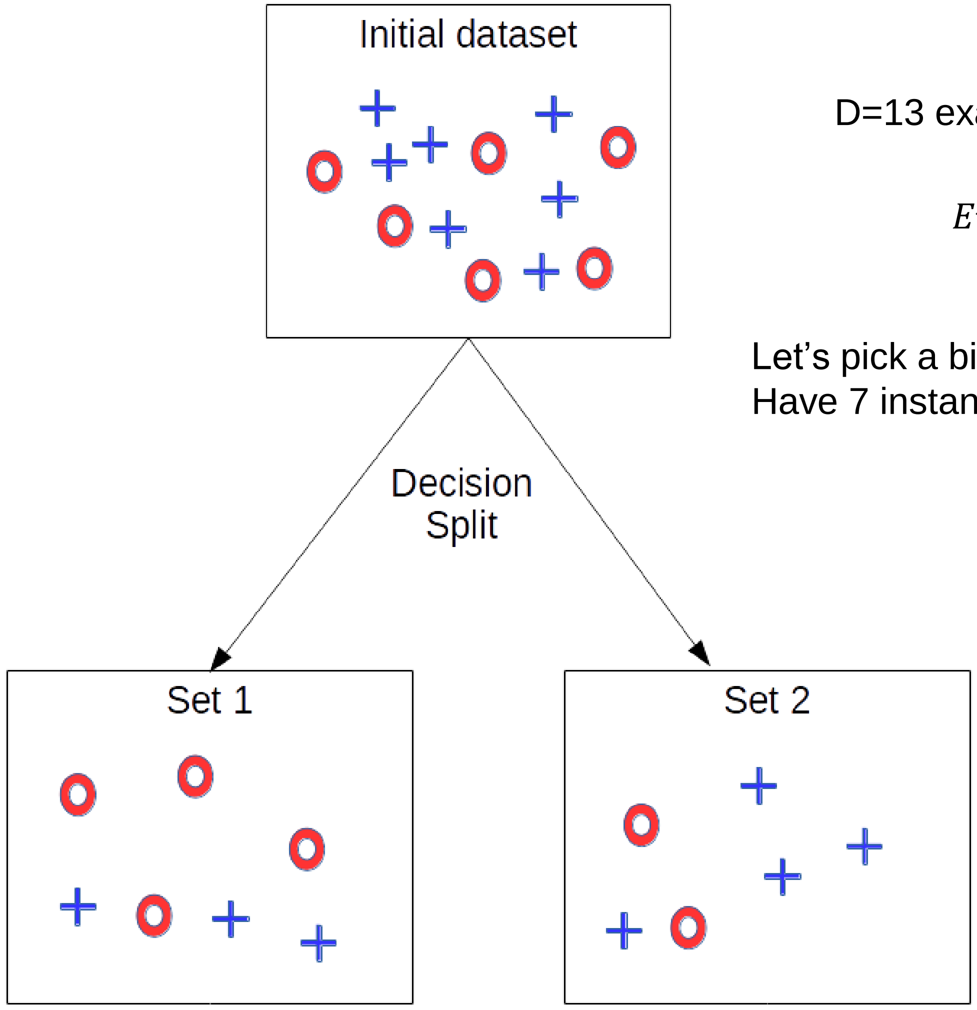
Information Gain

The **Information Gain** (IG, Gain) of a feature *f* is the **expected reduction in entropy resulting from splitting on this feature**.

$$Gain(D, f) = Entropy(D) - \sum_{v \in Values(f)} \frac{|D_v|}{|D|} Entropy(D_v)$$

➢ where D_v is the subset of *D* having value *v* for feature *f* (e.g, if *f*=color and *v*=red)

Example of Gain computation



$$\frac{D_{f=a}}{D} = \frac{7}{13}$$

D=13 examples, 7 positive 6 negative

$$Entropy(D) = -\frac{7}{13}\log(\frac{7}{13}) - \frac{6}{13}log\frac{6}{13}$$

Let's pick a binary feature *f* with values *a* and *b,* and let's suppose that we Have 7 instances out of 13 for which *f=a*, and 6 for which *f=b*

$$\frac{D_{f=b}}{D} = \frac{6}{13}$$

$$Gain(D,f) = Entropy(D) - \frac{7}{13}Entropy(Set1) - \frac{6}{13}Entropy(Set2)$$

$$Entropy(Set1) = -\frac{3}{7}\log(\frac{3}{7}) - \frac{4}{7}log\frac{4}{7}$$

$$Entropy(Set2) = -\frac{3}{6}\log(\frac{3}{6}) - \frac{2}{6}log\frac{2}{6}$$

Information Gain

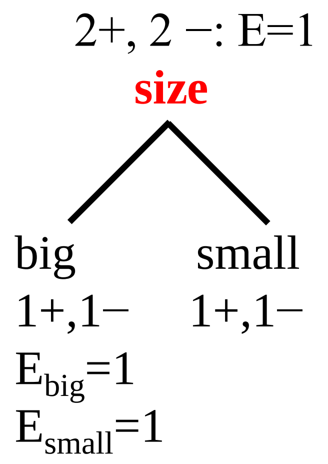
- Entropy of each resulting subset weighted by its relative size…

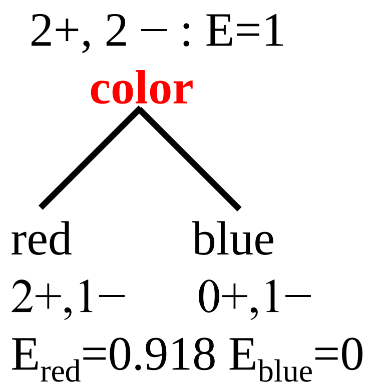
Example:

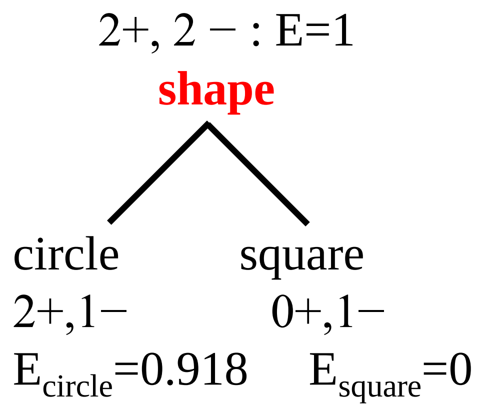
<big, red, circle>: + <small, red, circle>: +

<small, red, square>: − <big, blue, circle>: −

Initial Entropy is 1







Gain=$1-(0.5\cdot1 + 0.5\cdot1) = 0$

Gain=$1-(0.75\cdot0.918 + 0.25\cdot0) = 0.311$

Gain=$1-(0.75\cdot0.918 + 0.25\cdot0) = 0.311$

New pseudo-code

```
DTree(examples, features) returns a tree
  a) If all examples are in one category, return a leaf node with that category
     label.
  b) Else if the set of features is empty, return a leaf node with the category
     label that is the most common in examples.

Else pick the best feature f according to IG and create a node R for it
    For each possible value $x_i$ of f :
        Let $s_i$ be the subset of examples that have value $x_i$ for f
        Add an outgoing edge E to node R labeled with the value $x_i$.
    If $s_i$ is empty
        then attach a leaf node to edge E labeled with the
                  category that is the most common in examples.
        else call DTree($s_i$ , features – {f}) and attach the resulting
              tree as the subtree under edge E.
Return the subtree rooted at R.
```

A complete example

A Decision Tree example: “play Tennis”

- Data Example: “When do you play tennis?”

instance	*Outlook*	*Temperature*	*Humidity*	*Windy*	Play
x1	Sunny	Hot	High	False	No
x2	Sunny	Hot	High	True	No
x3	Overcast	Hot	High	False	Yes
x4	Rainy	Mild	High	False	Yes
x5	Rainy	Cool	Normal	False	Yes
x6	Rainy	Cool	Normal	True	No
x7	Overcast	Cool	Normal	True	Yes
x8	Sunny	Mild	High	False	No
x9	Sunny	Cool	Normal	False	Yes
x10	Rainy	Mild	Normal	False	Yes
x11	Sunny	Mild	Normal	True	Yes
x12	Overcast	Mild	High	True	Yes
x13	Overcast	Hot	Normal	False	Yes
x14	Rainy	Mild	High	True	No

The Process of Constructing a Decision Tree

- Select an attribute to place at the root of the decision tree and make one branch for every possible value.
- Repeat the process recursively for each branch.

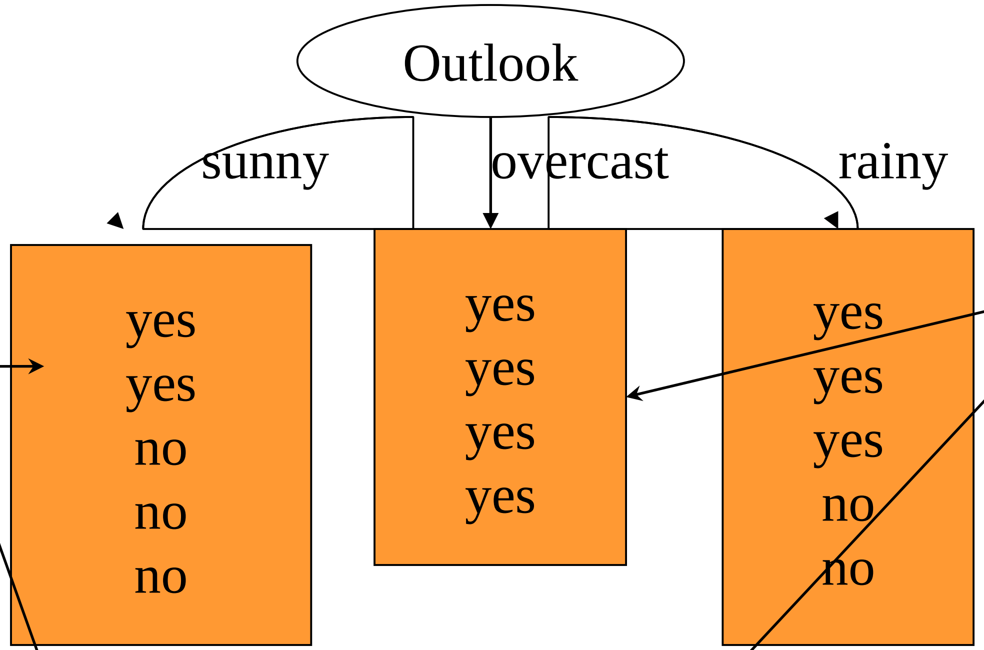
Information Gained by knowing the Result of a Decision

- In the “play tennis” example, there are 9 instances of which the decision to play is “yes” and there are 5 instances of which the decision to play is “no’. Then, the initial data entropy is:

$$\frac{9}{14}\times\left(-\log\frac{9}{14}\right)+\left(\frac{5}{14}\right)\times\left(-\log\frac{5}{14}\right)=0.940$$

The information initially required to correctly separate the data is 0.940 bits

Information further required if "Outlook" is placed at the root



0.971 is the entropy of 5 instances of which 3 have a label and 2 have the other label

0 is the entropy of a dataset where all instances have the same class label

Probability of outlook= sunny

Information further required =

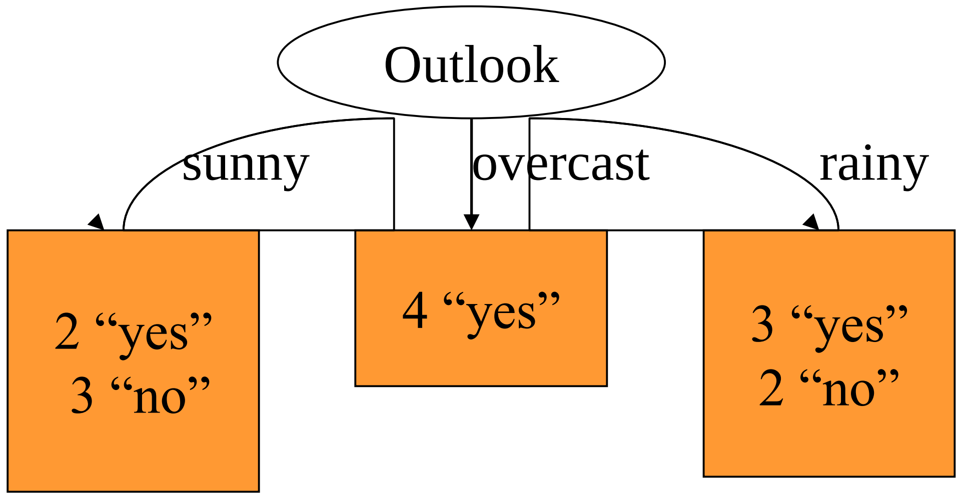
$$\left(\frac{5}{14}\right)\times 0.971+\left(\frac{4}{14}\right)\times 0+\left(\frac{5}{14}\right)\times 0.971=0.693\mathit{bits}.$$

Information Gained by Placing Each of the 4 Attributes

- Gain(outlook) = 0.940 bits – 0.693 bits = 0.247 bits.
- Gain(temperature) = 0.029 bits.
- Gain(humidity) = 0.152 bits.
- Gain(windy) = 0.048 bits.

The Strategy for Selecting an Attribute to Place at a Node

- Select the attribute that gives us the largest information gain.
- In this example, it is the attribute “Outlook”.



The Recursive Procedure for Constructing a Decision Tree

- Apply to each branch recursively to construct the decision tree.
- For example, for the branch “Outlook = Sunny”, we evaluate the information gained by applying each of the remaining 3 attributes.

  - ➢Gain(Outlook=sunny;Temperature) = 0.971 – 0.4 = 0.571
  - ➢Gain(Outlook=sunny;Humidity) = 0.971 – 0 = 0.971
  - ➢Gain(Outlook=sunny;Windy) = 0.971 – 0.951 = 0.02

Recursive selection

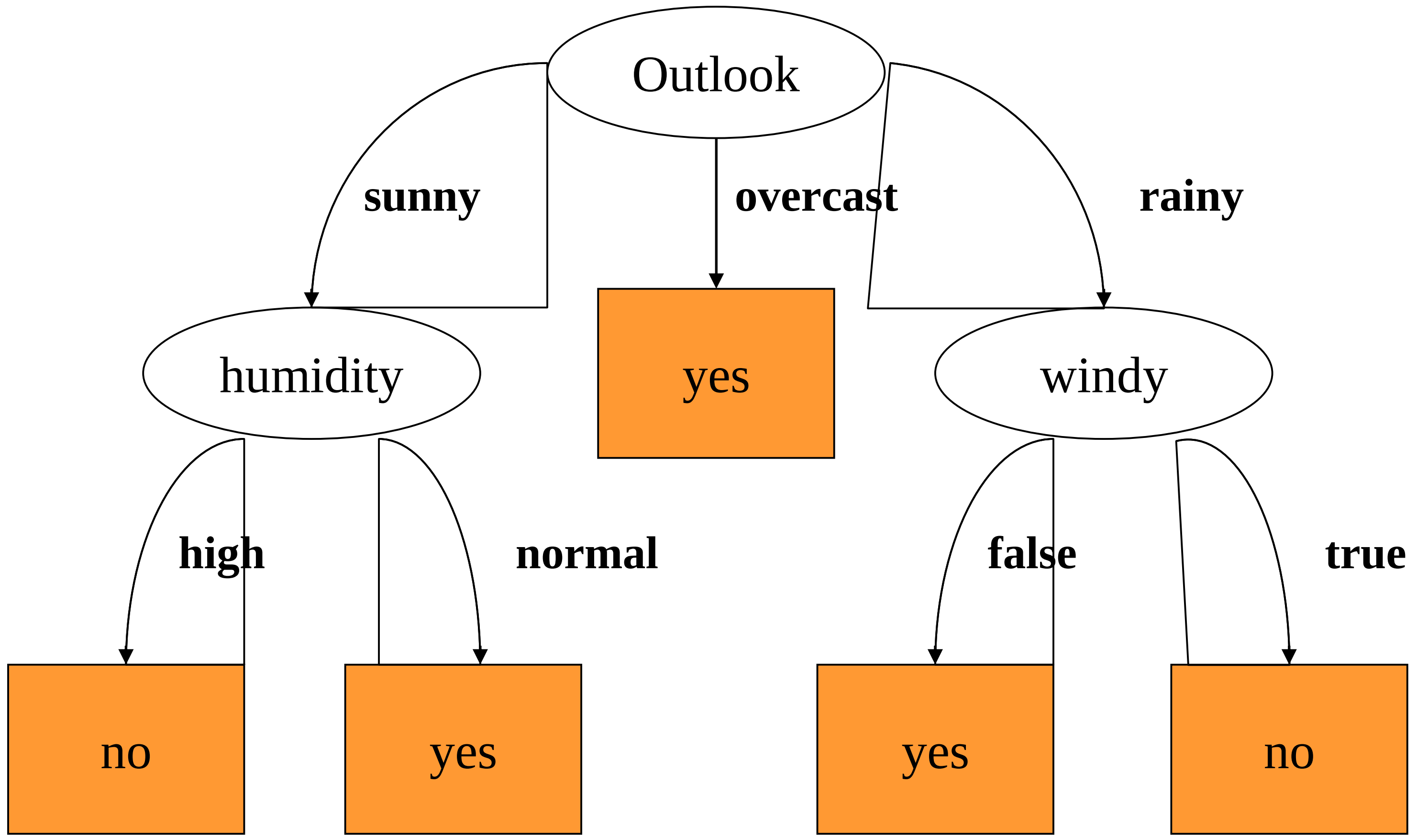
- Similarly, we also evaluate the information gained by applying each of the remaining 3 attributes for the branch “Outlook = rainy”.

  ➢ Gain(Outlook=rainy;Temperature) = 0.971 – 0.951 = 0.02

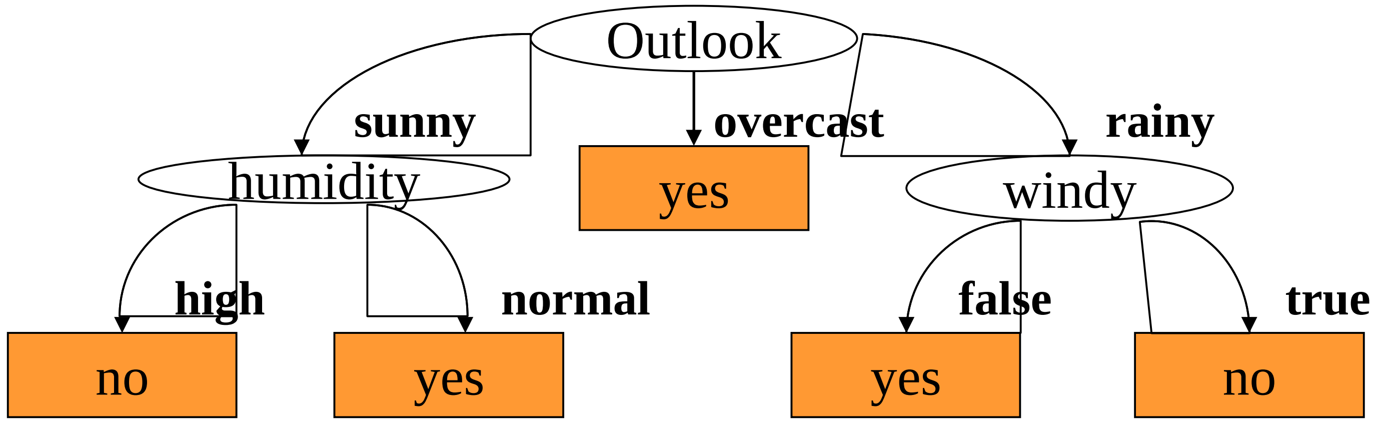
  ➢ Gain(Outlook=rainy;Humidity) = 0.971 – 0.951 = 0.02

  ➢ Gain(Outlook=rainy;Windy) =0.971 – 0 = 0.971

The Resulting Tree



DT can be represented as a set of rules



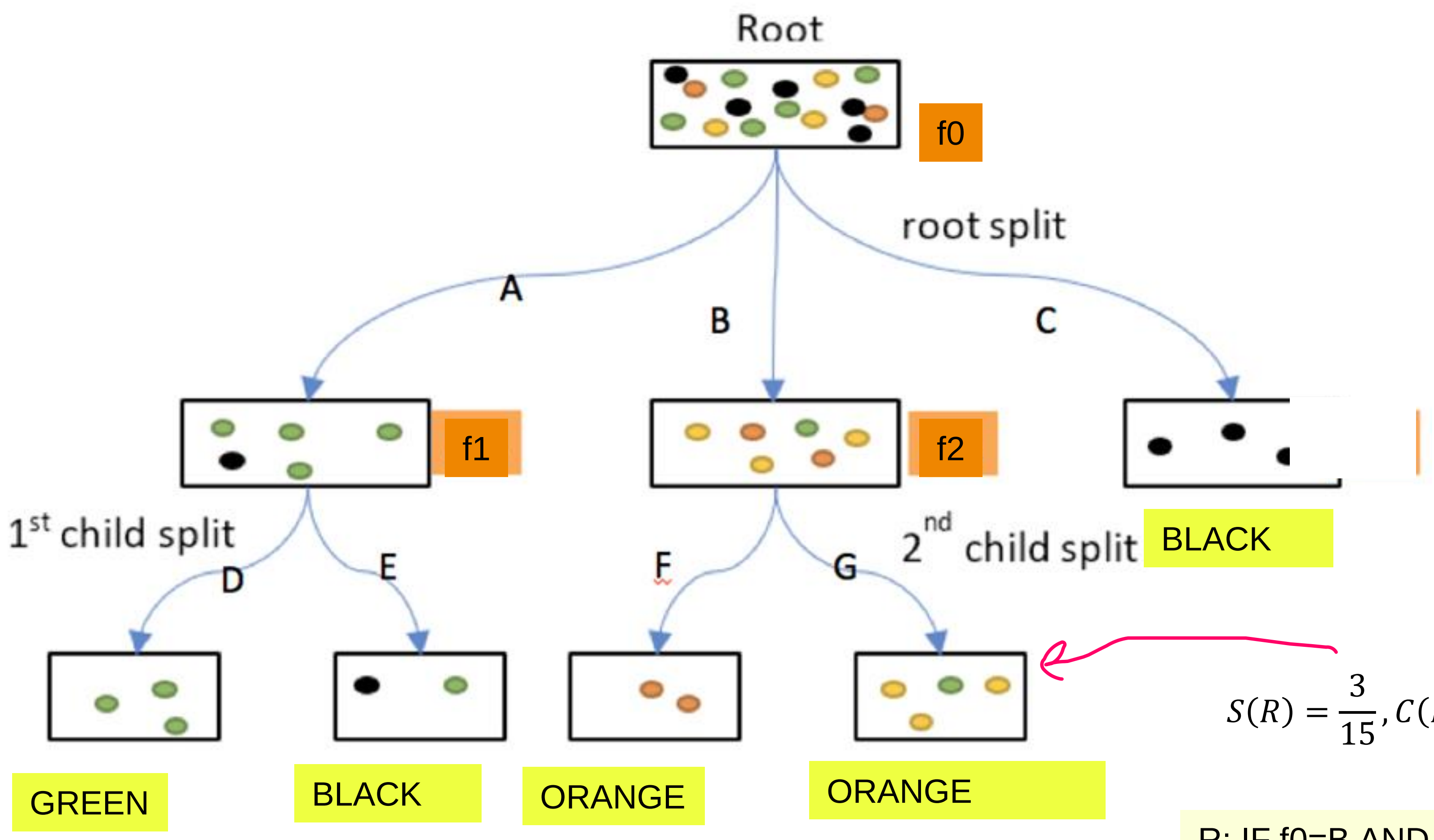
IF Outlook = sunny **AND** humidity = high → no
IF Outlook = sunny **AND** humidity = normal→ yes
IF Outlook = overcast → yes
IF Outlook = rainy **AND** windy = false→ yes
IF Outlook = rainy **AND** windy = true→ no

Support and Confidence: not all rules have the same relevance

- Let n be a root node, and R the rule that can be inferred following the pattern from the root to n. Let D_v be the set of examples matching the left hand side of R, and D'_v be the subset of examples that also match the right hand side (the decision). Note that in general, $D'_v \subseteq D_v$
- Each rule has a **support** $\frac{|D'_v|}{|D|}$ (or "cover") represented by the **% of examples in D that satisfy R.**

- Each rule has also a **confidence** which might or might not be equal to 1. The confidence is the % $\frac{|D'_v|}{|D_v|}$ of examples of the set D_v **which is correctly classified by the rule** (i.e. that match both the RHS and LHS of R)

Computing the support and confidence of DT rules



$$S(R) = \frac{3}{15}, C(R) = \frac{3}{4}$$

R: IF f0=B AND f2=G THEN ORANGE

Support and Confidence

Remember one of the 2 "exit" conditions in the algorithm:

- ***Else if*** *the set of features* ***is empty***, ***return*** *a leaf node with the category label that is the most common in examples.*

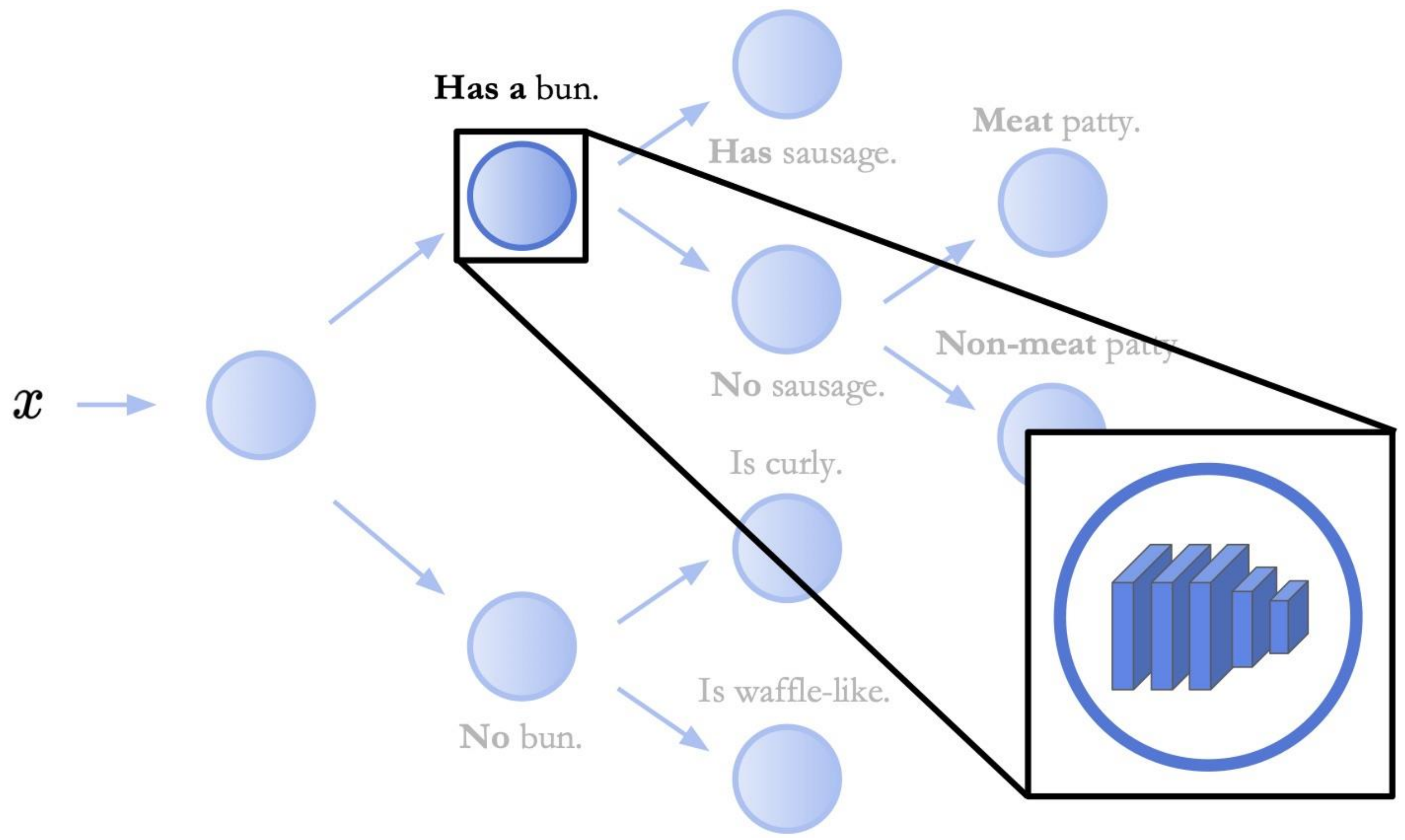
  Hence **if** the set of examples $|D_v|$ **does not** have a uniform classification, but, say, $|D_v+|$ positive and $|D_v-|$ negative, if $|D_v+|>|D_v-|$, we output the **label** "positive" and:
- **support** is $\frac{|D_v+|}{|D|}$
- **confidence** is $\frac{|D_v+|}{|D_v|}$

For example if we "consume" all features in a branch of the tree and we remain with 5 examples, of which 3 positive and 2 negatives, we append the decision "positive" to the tree branch (and its associated rule), with support 3 (or 3/|D|) and confidence 3/5

Issues of Decision Tree Learning

- Decision trees are among the first types of ML algorithms developed
- Can handle symbolic (discrete) features, continuous features can also be handled, if discretized beforehand by some data preprocessing method;
- The feature to be predicted must be discrete
- The real advantage of Dtrees is explainability (they are currently used as a post-prediction method for adding explainability see e.g. this NEURIPS 22 paper)
- Other recent approaches combine deep neural networks with decision trees to obtain models which are both interpretable and accurate (see e.g., this Berkley's univ. Paper 2020), but other DT+NN based methods have been conceived recently

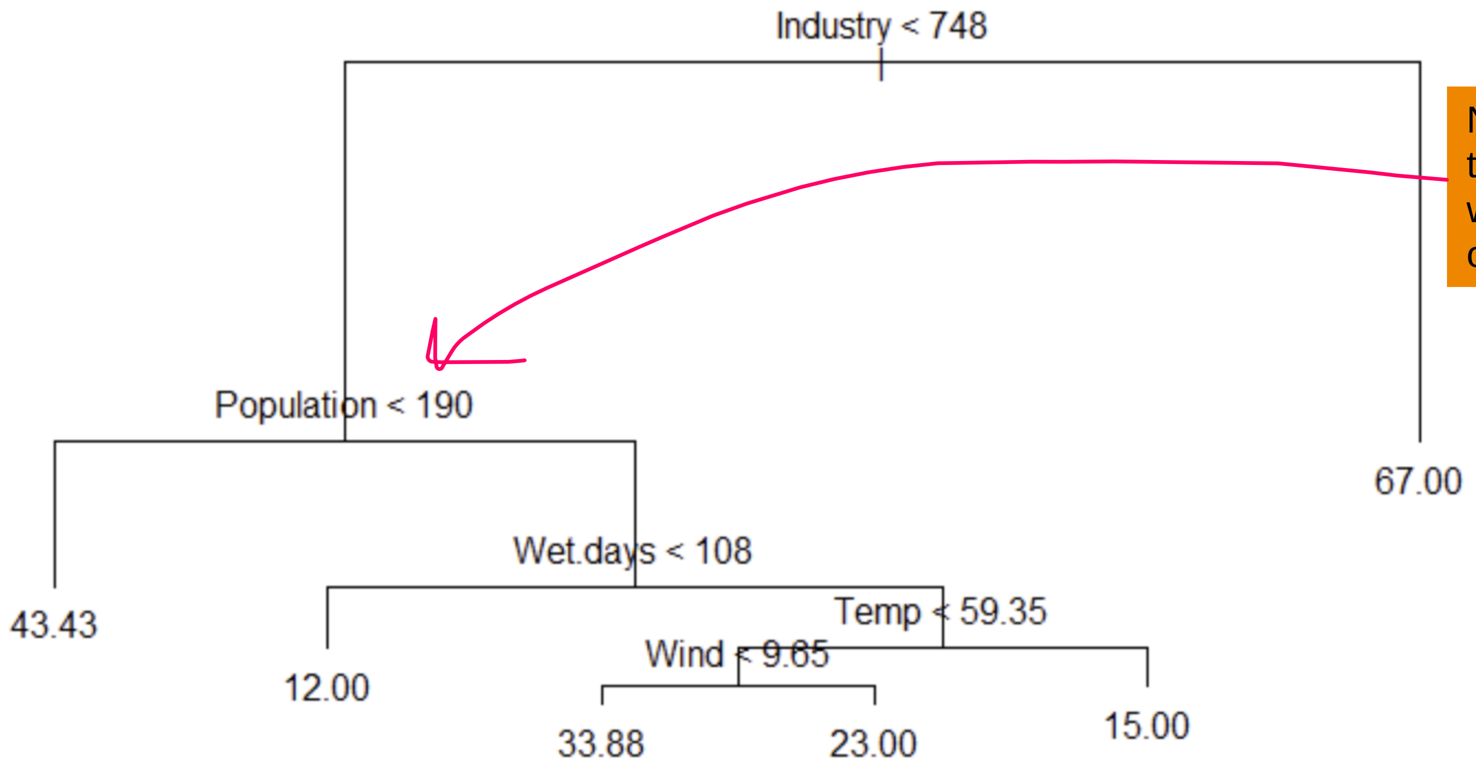
Neural-Backed Decision Trees



Here, predictions are made via a decision tree, preserving high-level interpretability.
However, each node in decision tree is a neural network making low-level decisions.

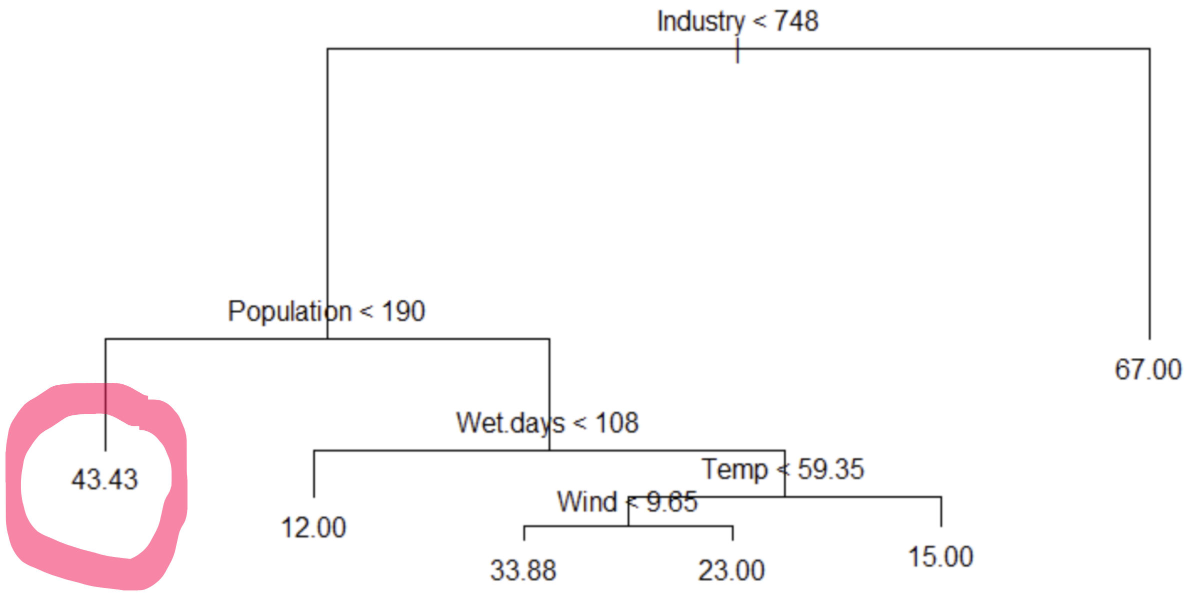
Regression Trees

- Regression trees handle both continuous features and non-categorical classification functions (reading on reg-trees: link)
- Regression trees output values $\boldsymbol{y} \in \mathfrak{R}$, rather than class labels $\boldsymbol{c_i} \in C=\{c_1,c_2,..c_n\}$



Note that at each node related to a continuous feature f, we have a split over the range of its values

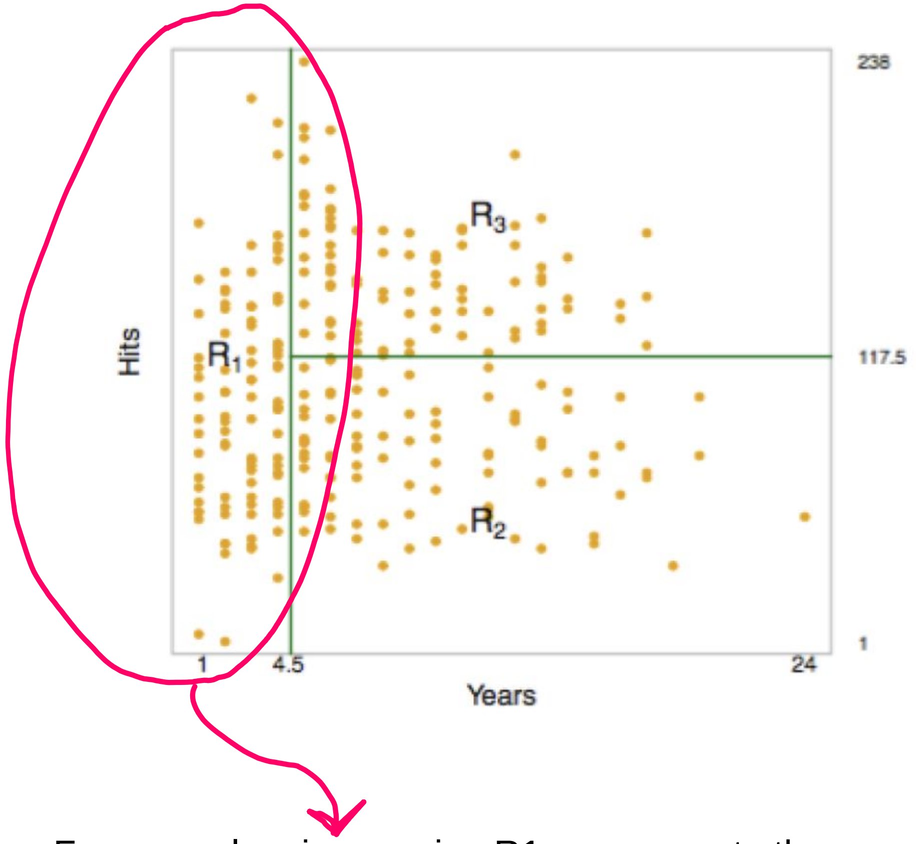
Regression Trees



- In Dtrees we can discretize features, but this is part of **data pre-processing**
- In RT, creating splits on continuous features **is part of the learning process** (RT *parameters*)
- Every branch of the tree defines a region in the multi-dimensional space, and the output *y* (leaf nodes of the tree) is the mean value of the output y of training data D in the defined region

e.g., 43.43 is the *mean value* that the feature y to be predicted has for all examples in the training set for which Industry is <748 and Population < 190

Regression trees (3)



For example, given region R1, we compute the average value of the output function y for all points x_i in the training set that fall into R1 (in simple terms, to minimize RSS, we want thatall the examples in each region have very «close» values of the output variable y)

- These «regions» in theory could have any shape. However, Rtrees divide the feature space into high-dimensional rectangles or «boxes» (for simplicity and ease of interpretation of the resulting predictive model).
- Note here for readability we show only *two-dimensional boxes*, but they can have as many dimensions as the features are (hyper-rectangles)
- Our goal is to find boxes **R1**, . . . , $\mathbf{R}_J$ that minimize the Residual Sum of Squares RSS given by:

$$RSS = \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

where $\widehat{y_{Rj}}$ is the **mean observed value** of the output y_i of training samples x_i lying in the box Rj, and y_i is the value of each single observation in the box Rj

Regression trees (3)

- It is computationally unfeasible to to consider *every possible partition* of the feature space into N boxes.
- Thus, we take a top-down, greedy approach called **recursive binary splitting**, called «top-down» since it begins at the top of the tree (all observations belong to a single region) and then successively splits the feature space.
- Each split corresponds to two new branches further down on the tree (note RT are *binary* trees).
- It is *greedy* since at each step of the tree building process, the best split is made at that particular split (rather than looking ahead and picking a split that will lead to a better tree in a future split).
- It still requires scanning all the observed values of the training set (or region) at each split
- You can learn more on RT algorithm at this link and this second link, but basically same algorithm as for Dtrees, but different optimization criterion (RSS rather than IG)

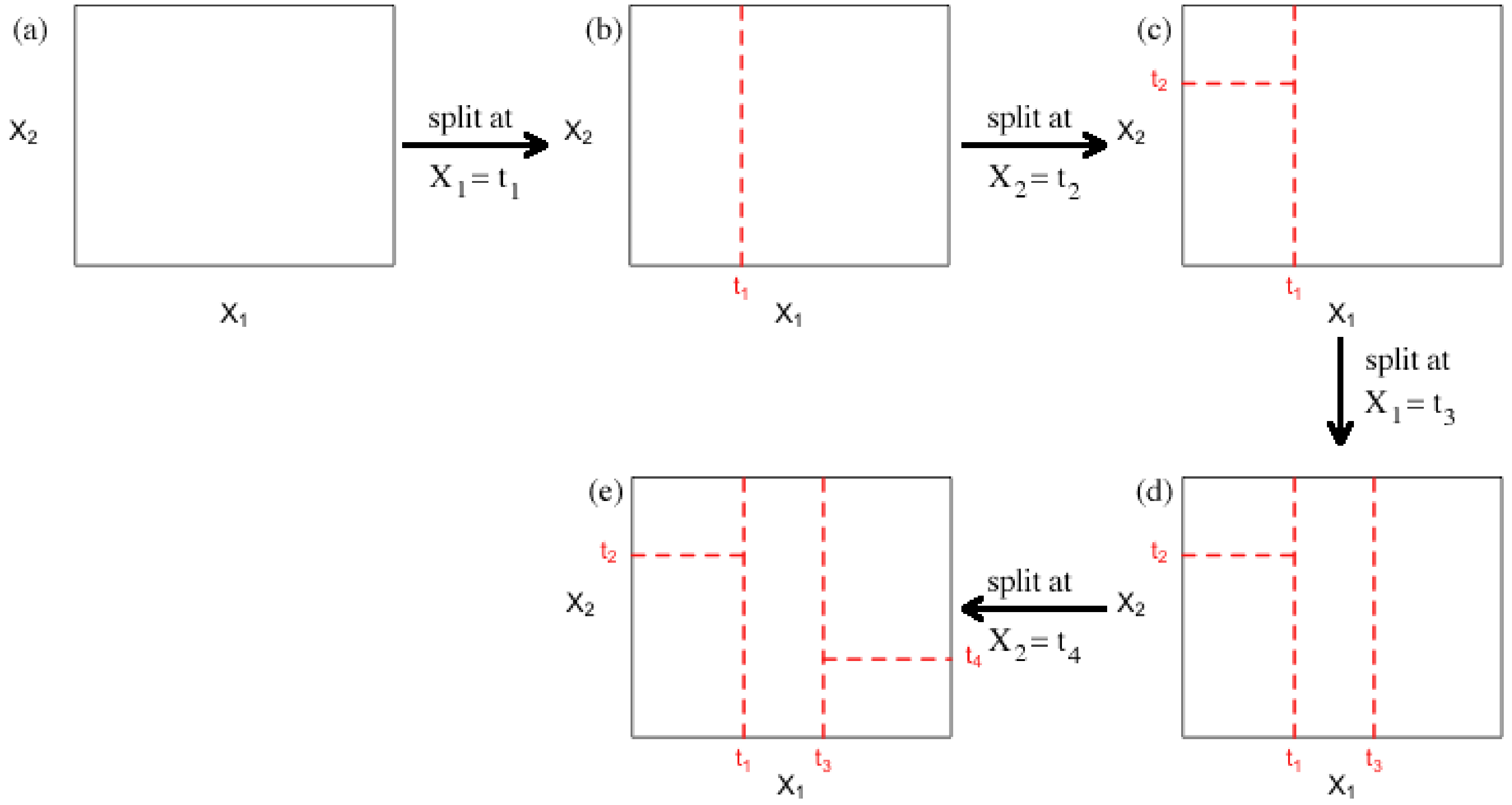
Algorithm

- **Let the initial RSS (no split) be** $\mathrm{RSS}_0 = \sum (y_i - \bar{y})^2$
- **Step 1: Choose a feature *f* and a split point s on its values.**
    - For continuous variables, identify a set of split points *s1..sm* and consider for each of them the generated regions R1 and R2
- **Step 2: For each possible partition calculate:**

$$\begin{aligned}\mathrm{RSS}(\mathrm{split}) &= \mathrm{RSS}_1 + \mathrm{RSS}_2 \\ &= \sum_{R_1} (y_i - \bar{y}_1)^2 + \sum_{R_2} (y_i - \bar{y}_2)^2\end{aligned}$$

- **Step 3** examine each feature f_j and all possible split points *s* and choose the one for which RSS_0 – RSS(split) is the largest.
- **Step 4:** iterate on each new generated region

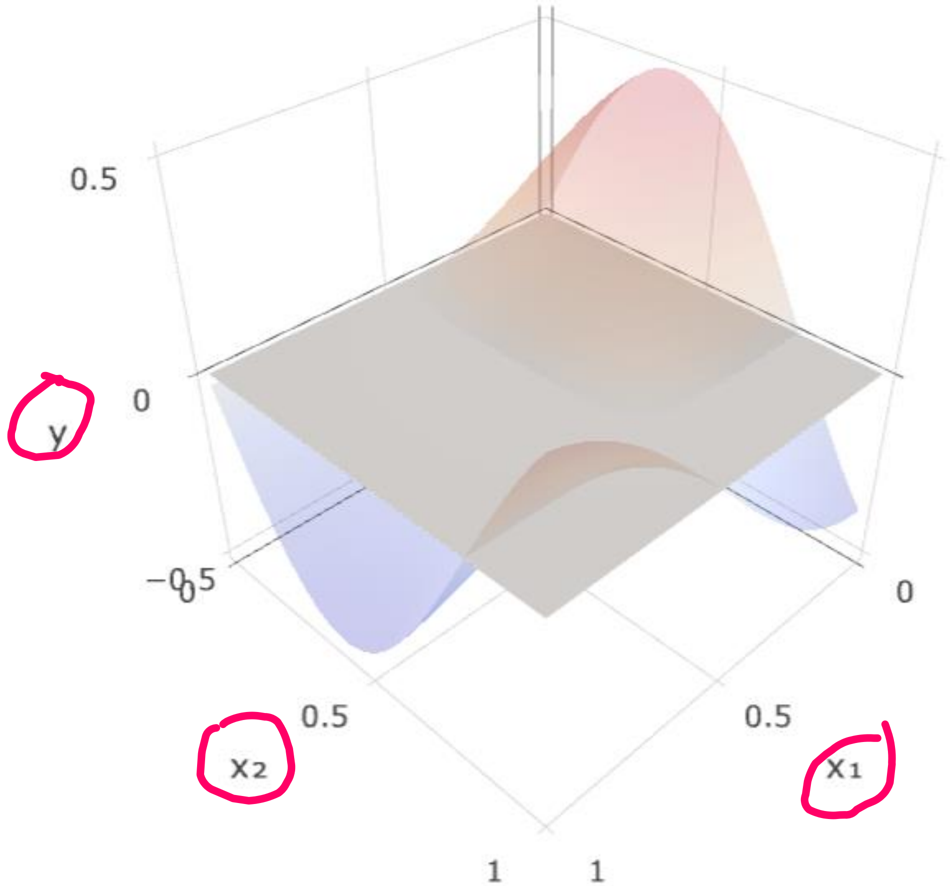
Recursive binary splitting (1)



- Remember, we said that learning implies tuning the model **parameters** and establishing an optimization problem.
- Which parameters we tune in DT and RT?
- What is the optimization function?

Regression trees visualized

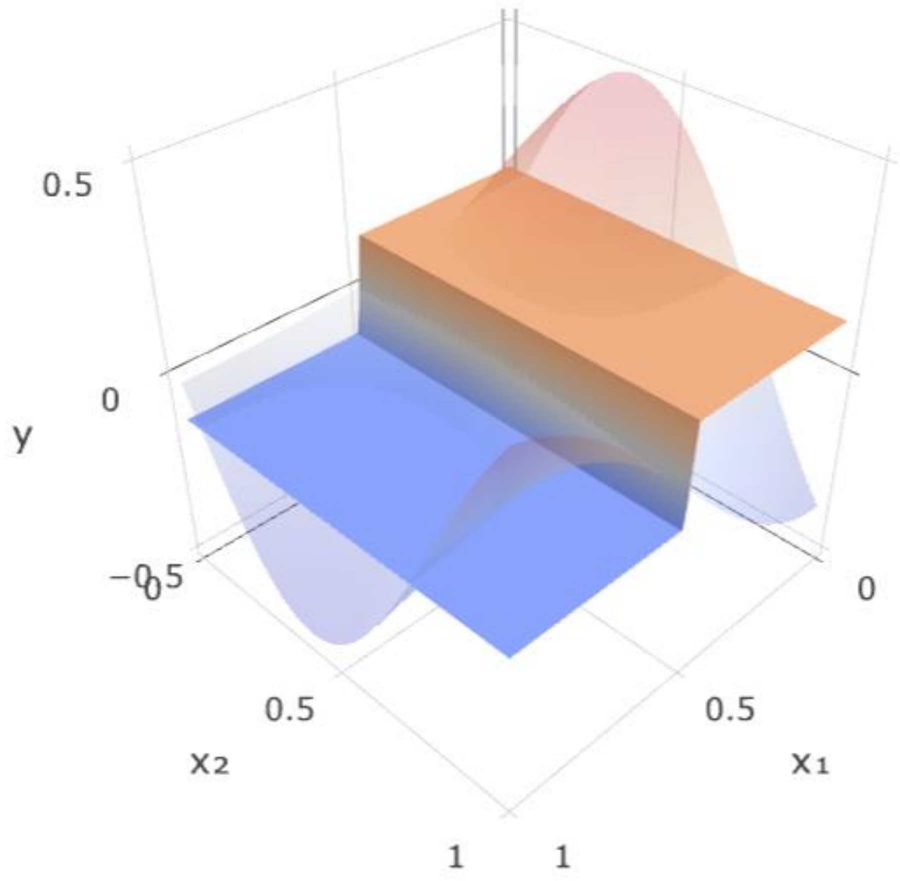
semi-transparent target function $f(\mathbf{x})$ and tree prediction $d_{\text{tree}}(\mathbf{x})$



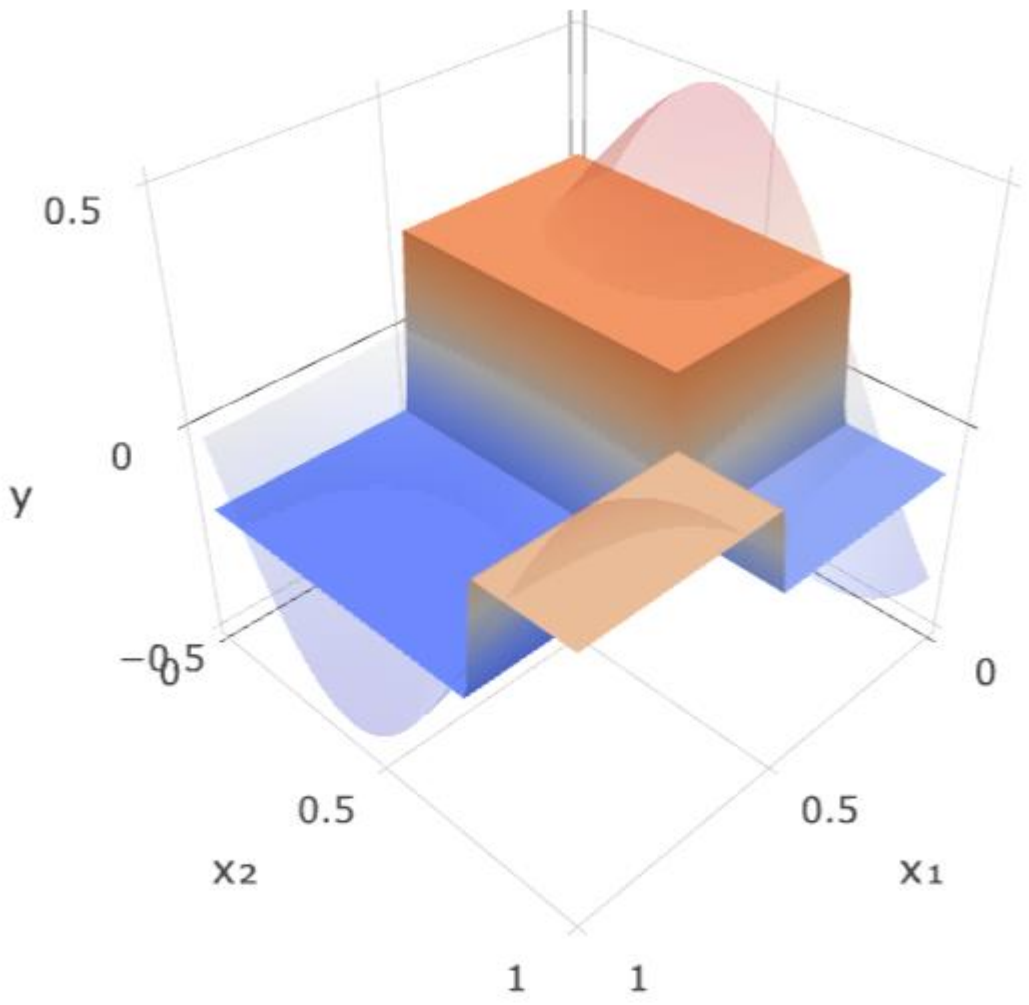
Depth of the tree: 0

http://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html

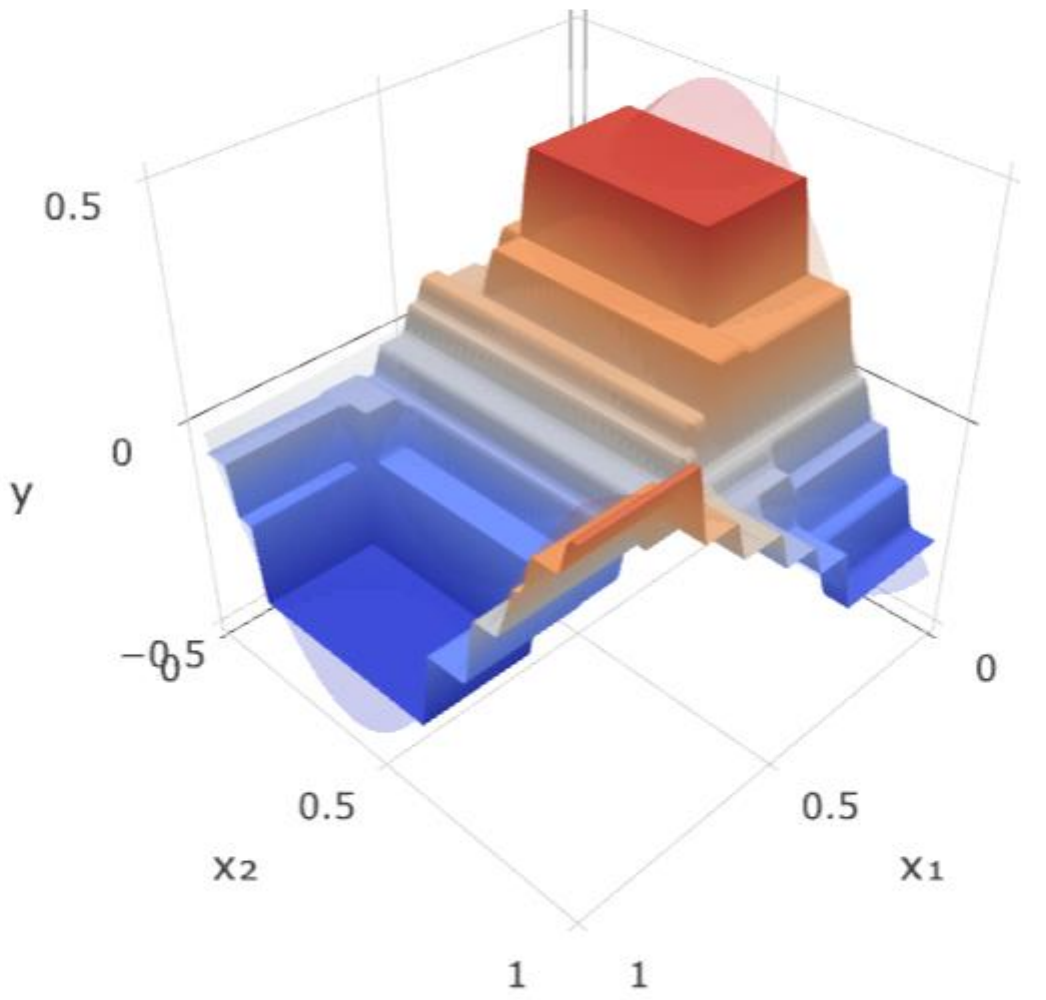
Tree depth: 1



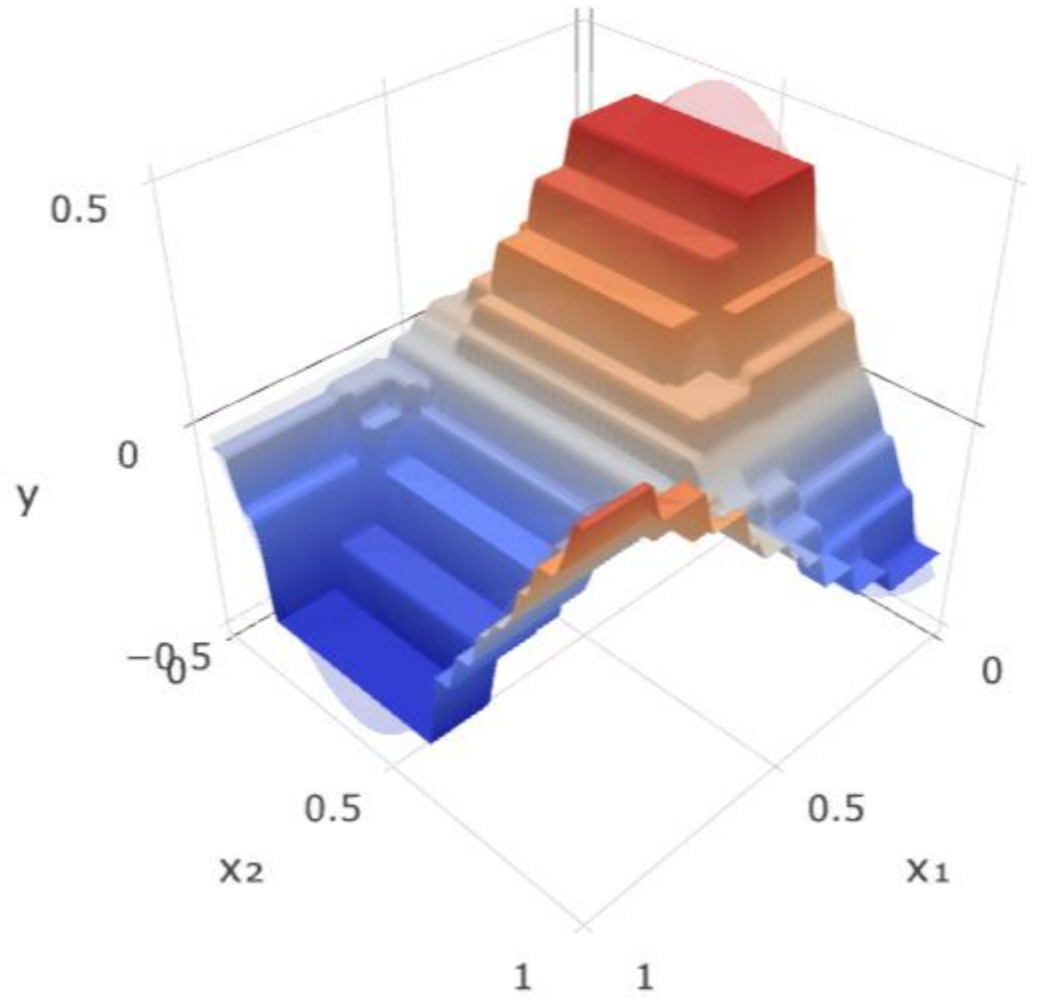
Tree Depth: 2



Tree depth 5



Tree depth: 6



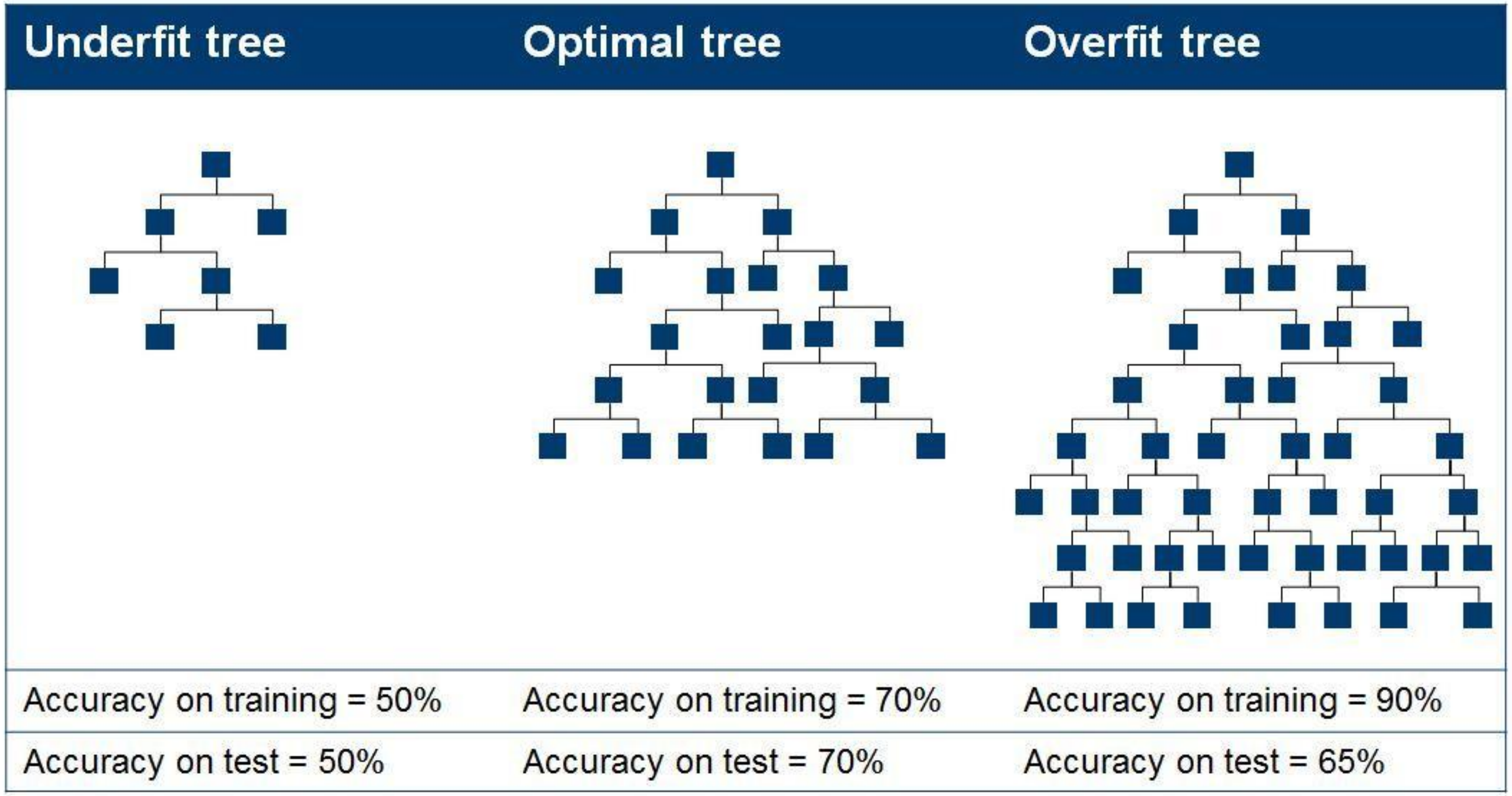
Summary so far

- Decision tree algorithm
- Ordering nodes (Information Gain)
- Regression Trees (Residual Sum of Squares)
- **Fine-tuning the tree**

Fine-tuning the tree

- Decision or regression trees are **not optimal**, they are obtained as the result of a *greedy process*
- *Remember: ML systems commonly approximate optimal solutions with greedy search*
- A common problem is that the resulting tree might be excessively *bushy* – this is a general problem (for all types of ML algorithms) denoted as OVERFITTING
- Overfitting happens when a model learns too many details and even noise in the training data, causing a negative impact on the ability of the model to **generalize**. This means that the model may perform poorly on new data.

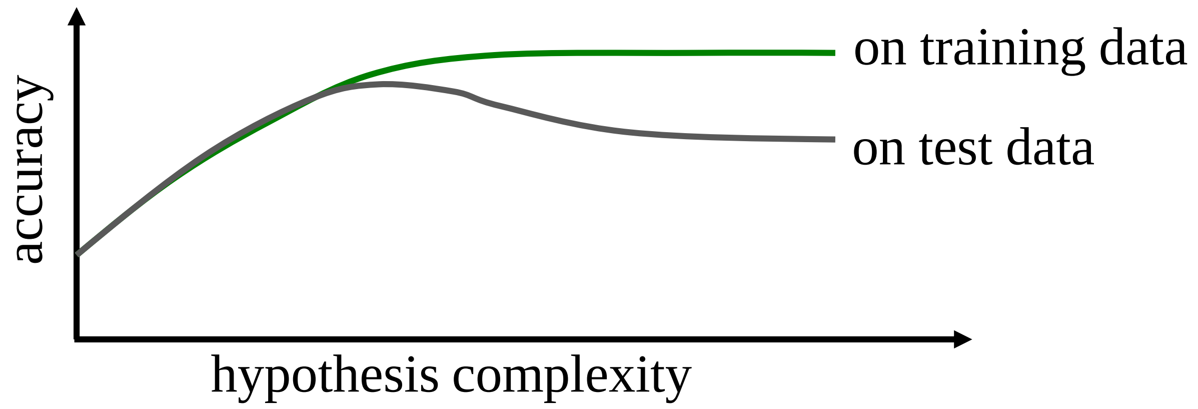
Overfitting decision trees



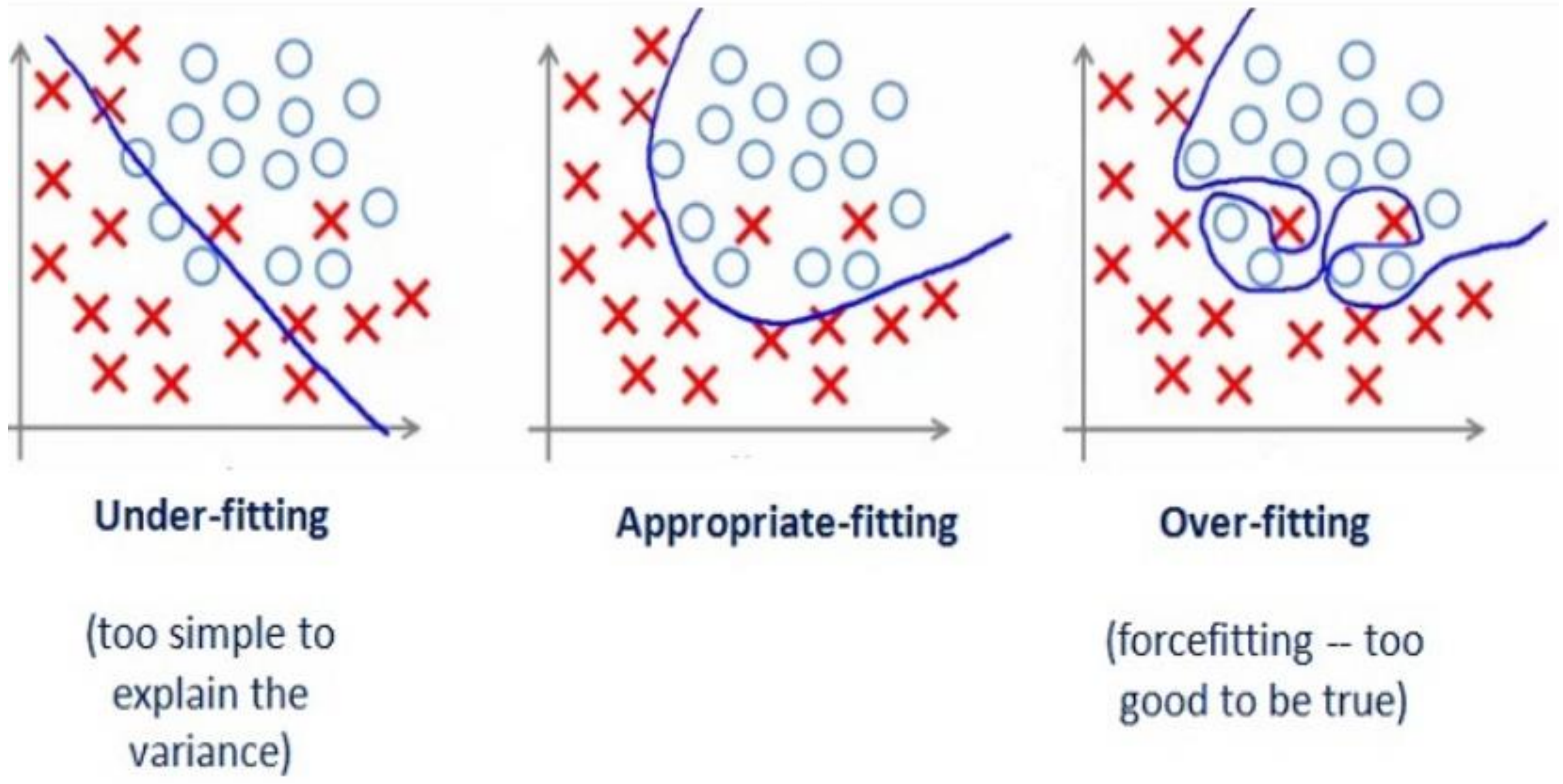
Underfit tree	Optimal tree	Overfit tree
Accuracy on training = 50%	Accuracy on training = 70%	Accuracy on training = 90%
Accuracy on test = 50%	Accuracy on test = 70%	Accuracy on test = 65%

Overfitting

- Learning a tree that classifies the training data perfectly may not lead to the tree with the best generalization to unseen data.
  - ➢ There may be noise in the training data that the tree is erroneously fitting.
  - ➢ The algorithm may be making poor decisions towards the leaves of the tree that are based on very little data and may not reflect reliable trends (e.g. **reliable rules should be supported by "many" of examples, not just a handful**).
- A hypothesis, *h*, is said to **overfit** the training data is there exists another hypothesis, *h´*, such that *h* has less error than *h´* on the training data but a greater error on independent test data.



Overfitting more in general

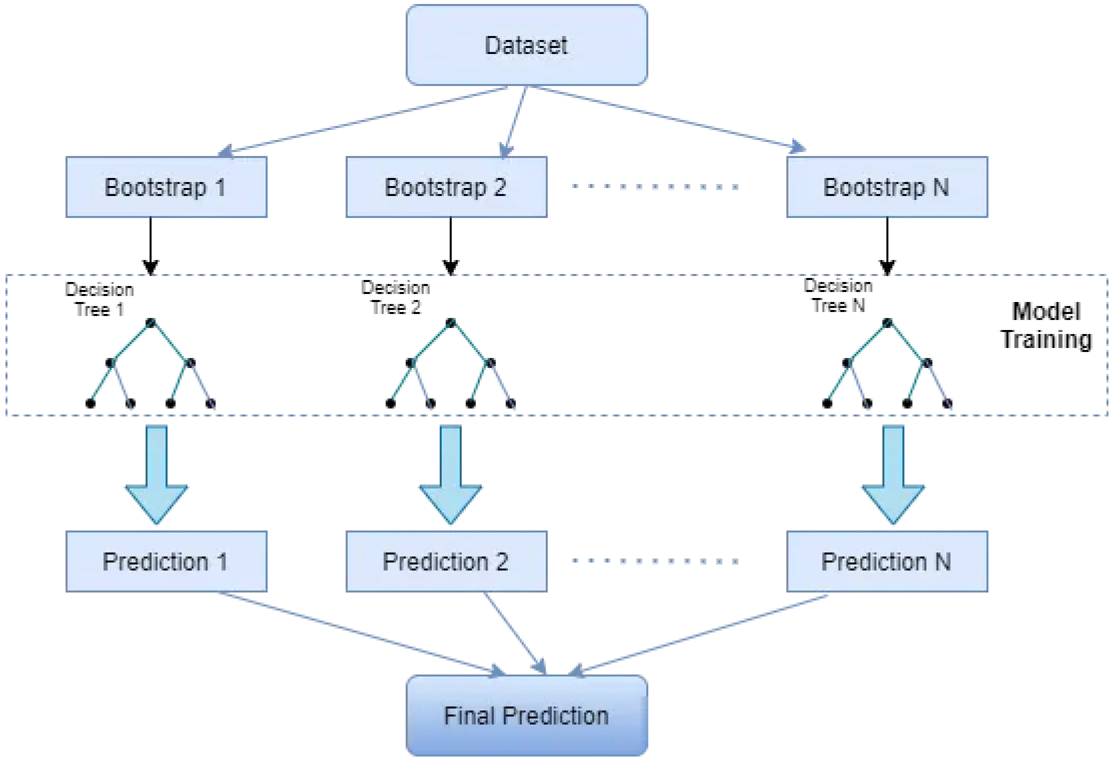

Under-fitting
(too simple to explain the variance)
Appropriate-fitting
Over-fitting
(forcefitting -- too good to be true)

Overfitting Prevention in Dtrees: (Pruning) Methods

- Two basic approaches for decision trees:
  - **Pre-pruning:** Stop growing tree as some point during top-down construction when there is no longer sufficient data to make reliable decisions (e.g. $|D_v|<k$).
  - **Post-pruning**: Grow the full tree, then remove sub-trees that do not have sufficient evidence (support).
- Label leaf resulting from pruning **with the majority class** of the remaining data, or a class probability distribution.
- This step has the effect of reducing confidence

Will see more on decision and regression trees..

… when we will introduce Ensamble methods


Dataset
Bootstrap 1
Bootstrap 2
Bootstrap N
Decision Tree 1
Decision Tree 2
Decision Tree N
Model Training
Prediction 1
Prediction 2
Prediction N
Final Prediction