Ensemble Learning

Liang Liang

Notation

- a set of N data points $\{x_1, x_2, x_3, \dots, x_N\}$ and $x_n \in \mathcal{R}^M$
- a data point $x_n \in \mathcal{R}^M$, it is a vector and has M elements
- a set of 'ground-truth' target values $\{y_1, y_2, y_3, \dots, y_N\}$

- Each data point has M features

$$x_n = \left[x_{(n,1)}, x_{(n,2)}, x_{(n,3)}, \dots, x_{(n,m)}, \dots, x_{(n,M)}\right]^T$$

- Drop the index n

$$x = \left[x_{(1)}, x_{(2)}, x_{(3)}, \dots, x_{(m)}, \dots, x_{(M)}\right]^T$$

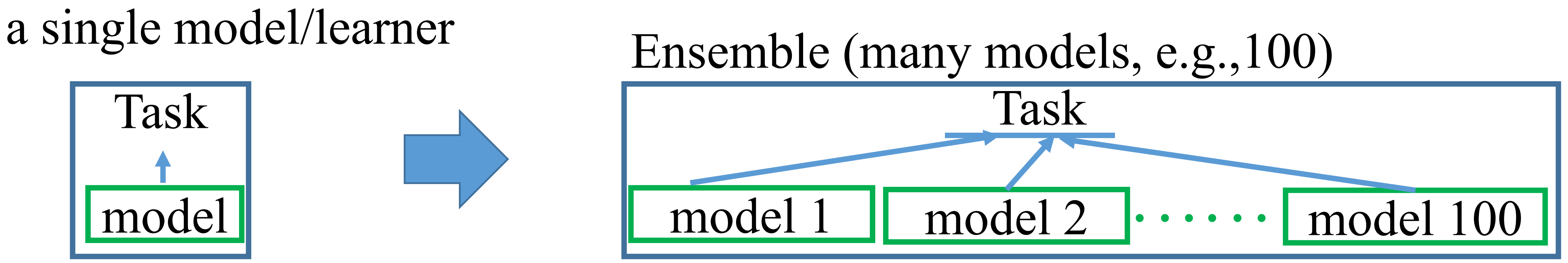
x_1 is a data point/vector

$x_{(1)}$ is a feature component of a vector, it is a scalar

Rationale: Combination of Models

- There is no algorithm/model that is always the most accurate
- We can build many models (e.g., simple classifiers or regressors) and combine them into a single "strong" model
- Different models may use different

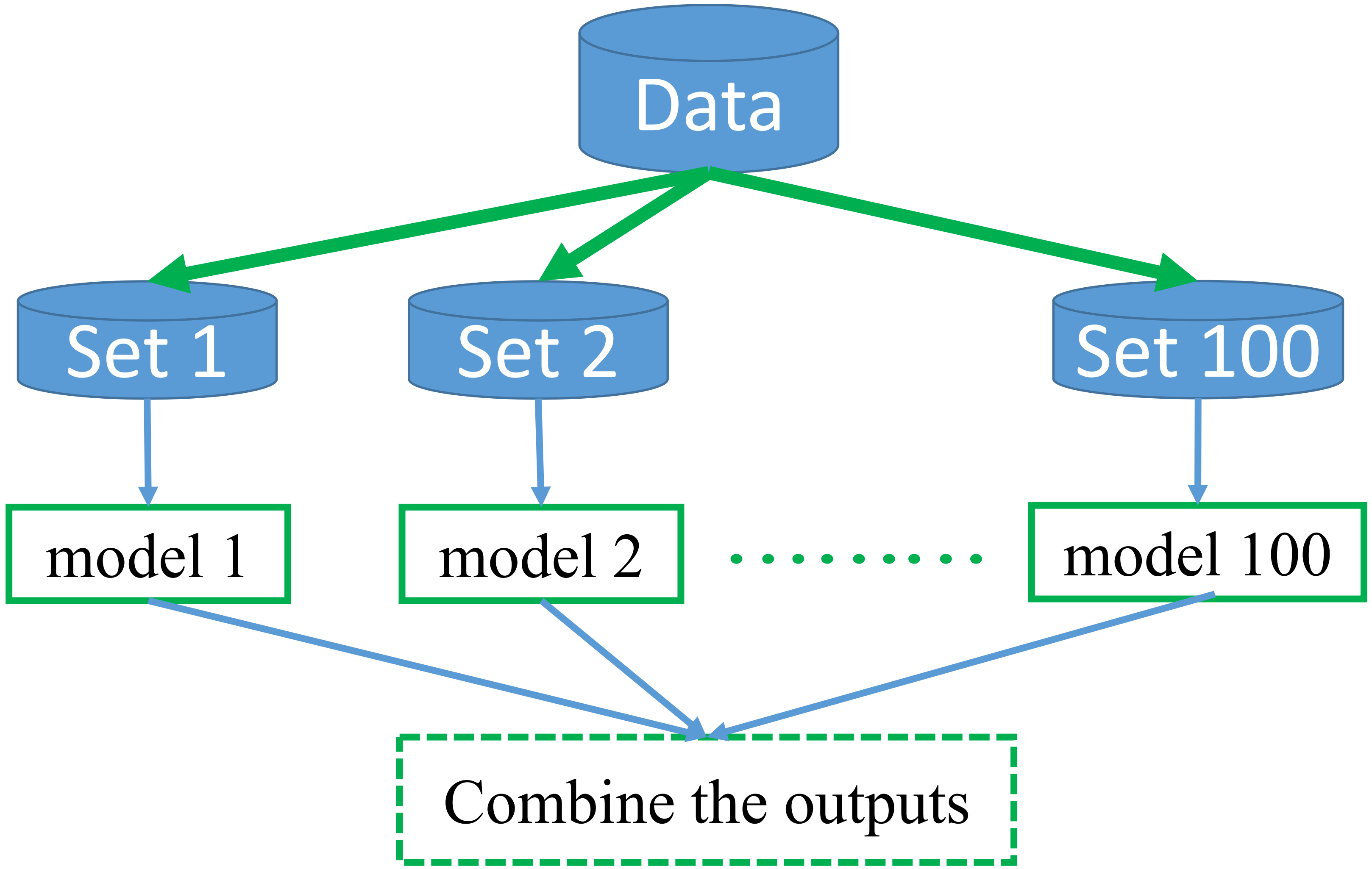
  learning algorithms, hyper-parameters, training sets, model type/structure, etc



Three main strategies: Bagging, Boosting and Stacking

- Bagging
  - Train weak/simple models simultaneously on bootstrap replicates of the training set (randomly get a subset of the training data)
  - Regression: average the outputs from the simple models
  - Classification: take majority vote among the outputs (averaging)

Bagging



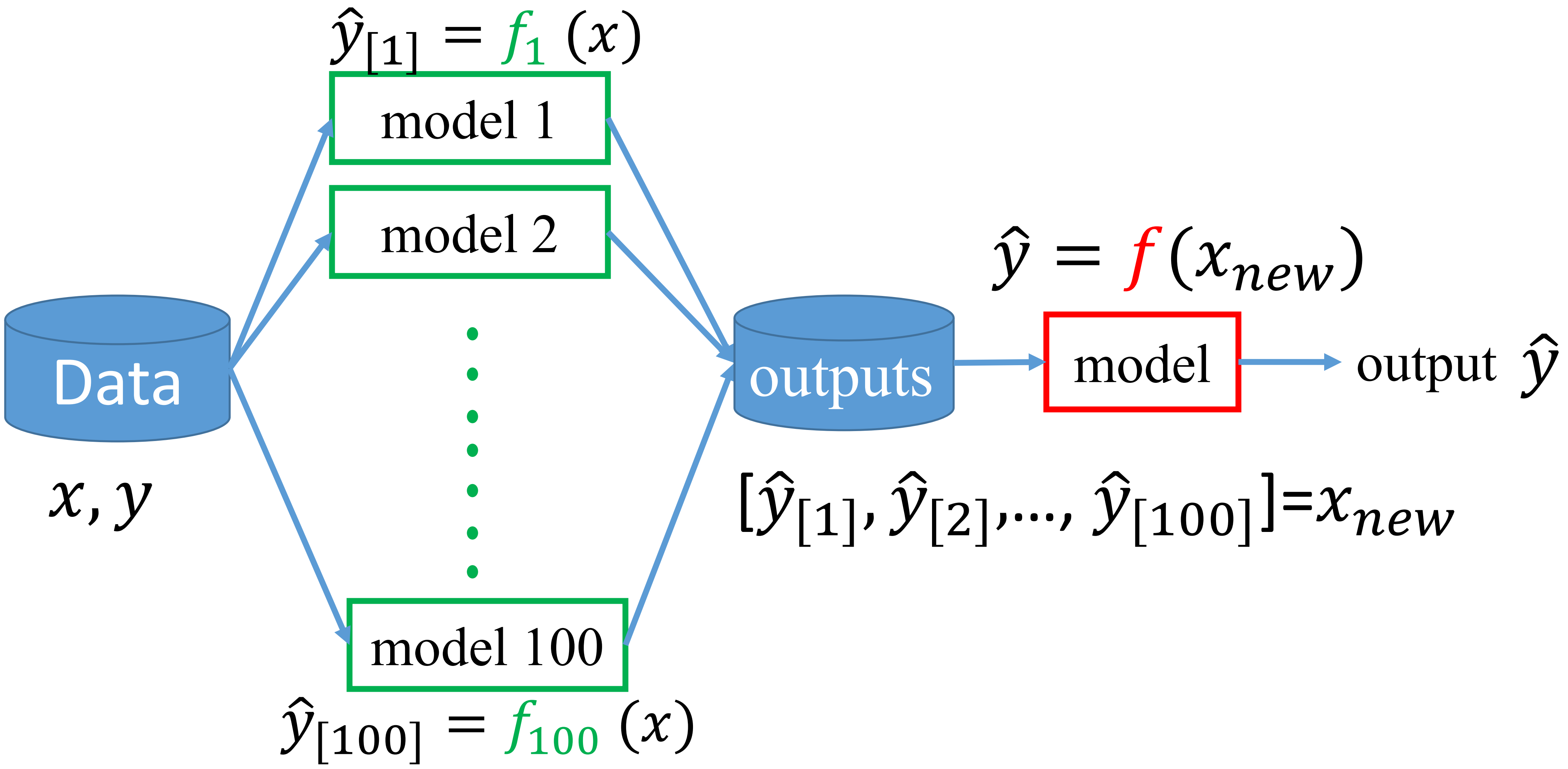
Combine the outputs:
Regression: average the outputs from the weak/simple models
Classification: take majority vote among the outputs (averaging)

Boosting

- Define a Loss function
- Simple models are trained one after another to minimize the loss
    - train model-2 after model-1 is trained
    - train model-3 after model-2 is trained
    - ….
    - add the outputs together

Stacking

- Train a model using the outputs from many other models



Bagging

- Why does it work ?

  averaging the outputs may reduce the variance of estimation

  if the outputs are i.i.d. random variables or weakly correlated.

- A simple example:

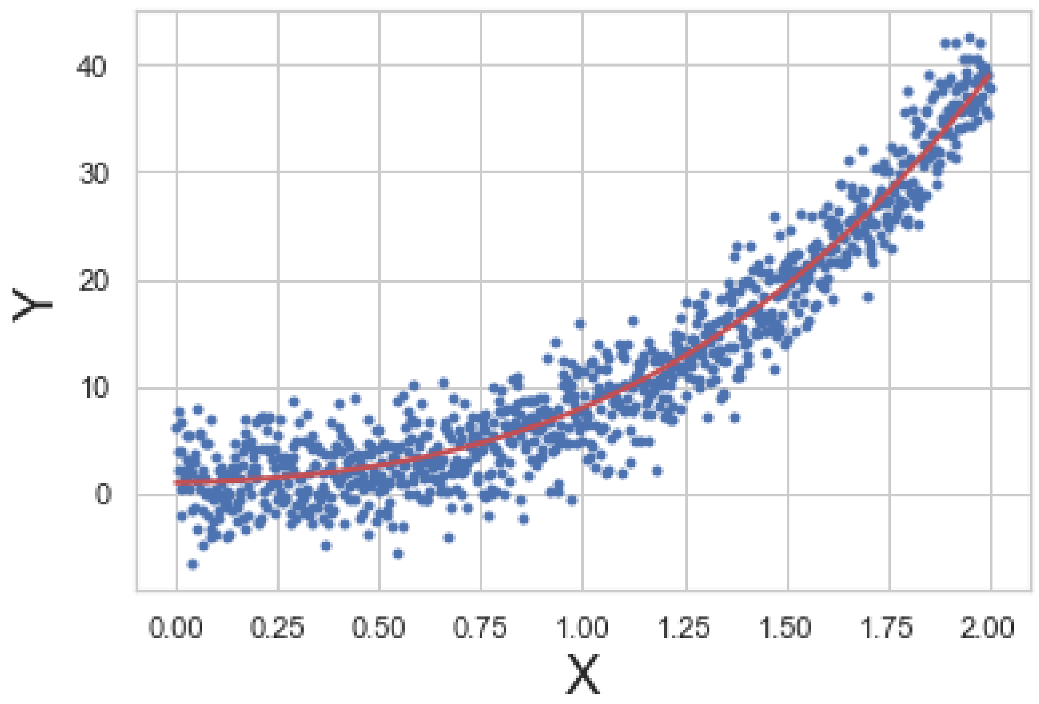
We measure the length of an object *100* times and then get a sequence of samples $r_1, r_2, r_3, \ldots, r_{100}$, and we assume the samples have PDF: $\mathcal{N}(\mu, \sigma^2)$

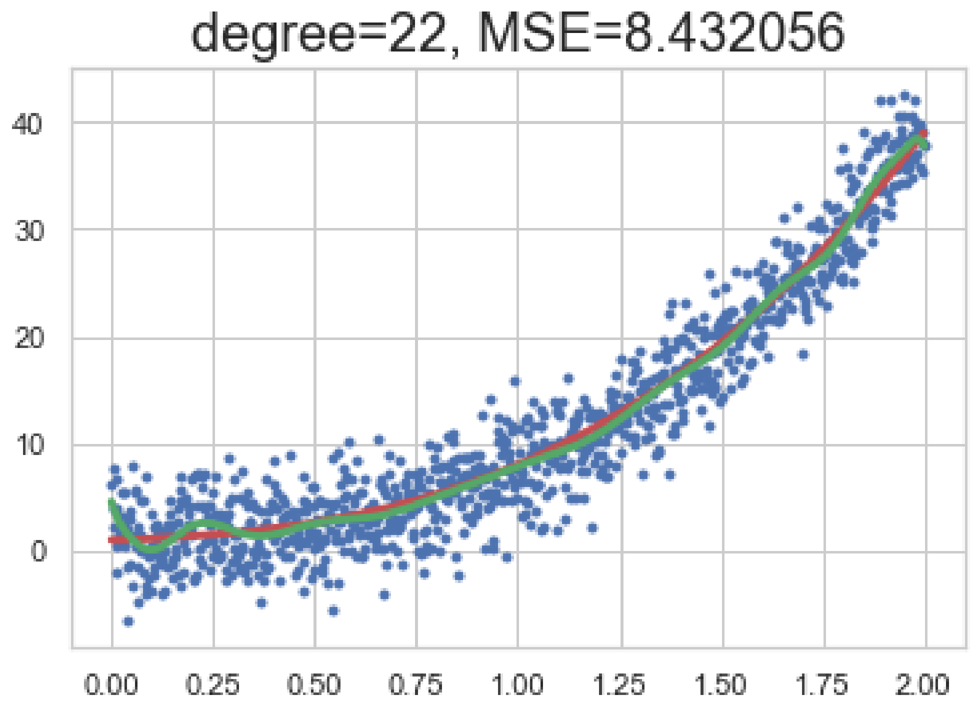
The best estimation of the length is the average $\bar{r} = \frac{1}{100} \sum_{n=1}^{100} r_n$

the variance of the estimation

$$var(\bar{r}) = var\left(\frac{1}{100} \sum_{n=1}^{100} r_n\right) = \frac{1}{100^2} \sum_{n=1}^{100} var(r_n) = \frac{1}{100^2} \sum_{n=1}^{100} \sigma^2 = \frac{\sigma^2}{100}$$

Does Averaging always work ?





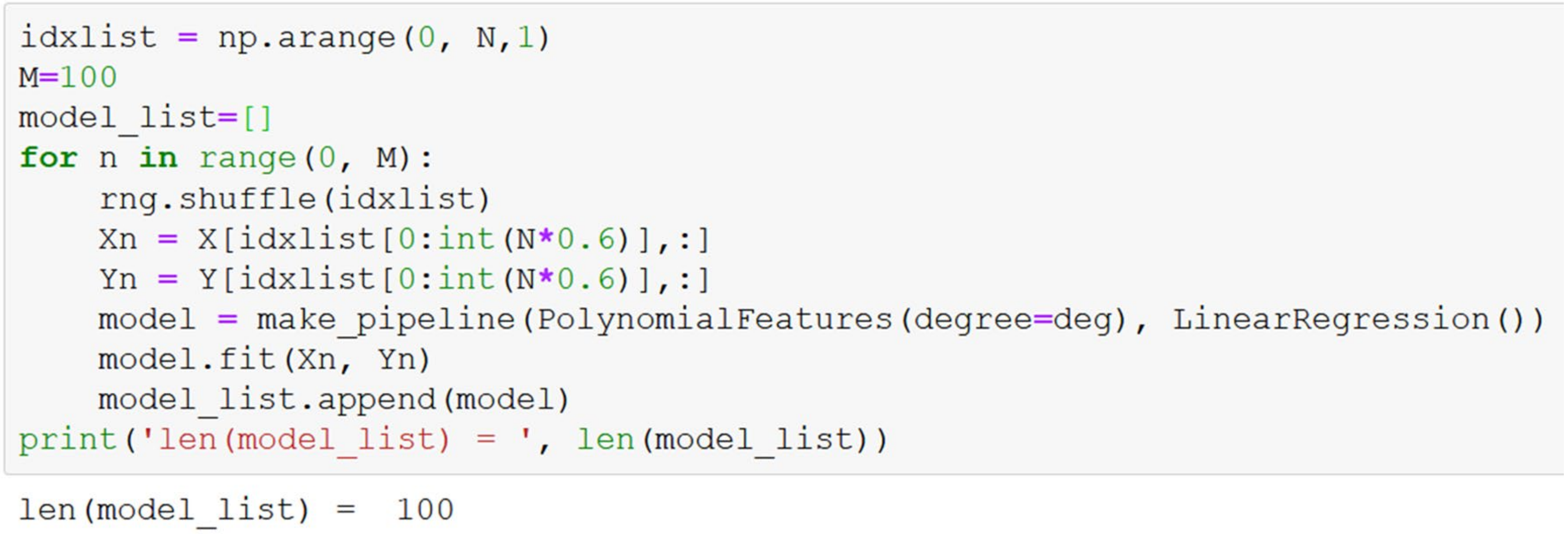
$$\boldsymbol{y_{best}} = f(x) = 1 + x + 2x^2 + 3x^3$$

$$\boldsymbol{y} = y_{best} + \varepsilon \text{ , where } \varepsilon \sim \mathcal{N}(0, \sigma_\varepsilon^2)$$

The polynomial model of degree 22

Does Averaging always work ?

Let's fit 100 polynomial models of degree 22 to different subsets of the data



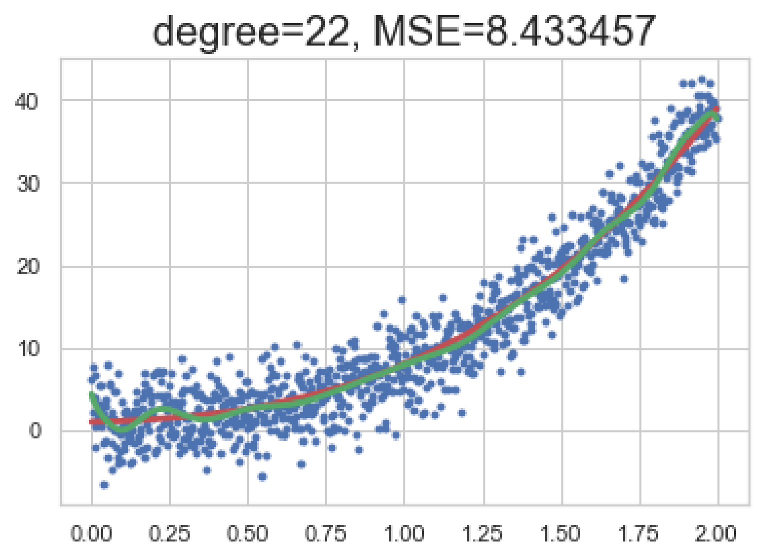
```python
idxlist = np.arange(0, N,1)
M=100
model_list=[]
for n in range(0, M):
    rng.shuffle(idxlist)
    Xn = X[idxlist[0:int(N*0.6)],:]
    Yn = Y[idxlist[0:int(N*0.6)],:]
    model = make_pipeline(PolynomialFeatures(degree=deg), LinearRegression())
    model.fit(Xn, Yn)
    model_list.append(model)
print('len(model_list) = ', len(model_list))
```

```
len(model_list) =  100
```

Does Averaging always work ?

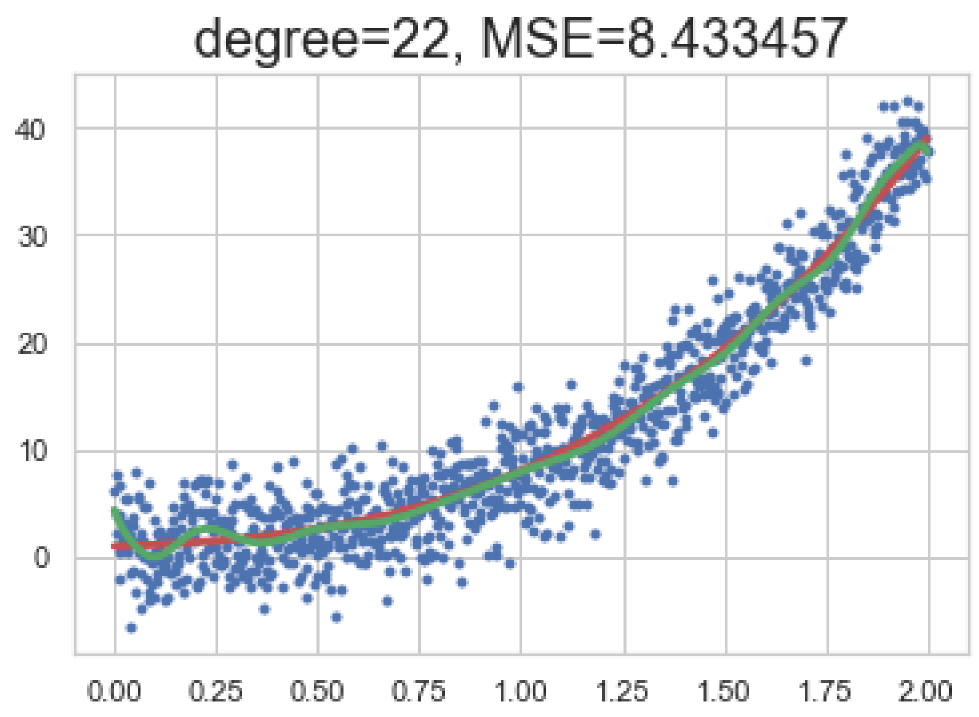
Let's average these 100 polynomial modes of degree 22

```
Yp_avg = np.mean(Yp, axis=0).reshape(N,1)
MSE = np.mean((Y-Yp_avg)**2)
MSE = '{:6f}'.format(MSE)
plt.plot(X, Y,'.b')
plt.plot(X, Y_best,'-r', linewidth=3)
plt.plot(X, Yp_avg, '-g', linewidth=3)
plt.title('degree='+str(deg)+', MSE='+str(MSE), fontsize=20)
```



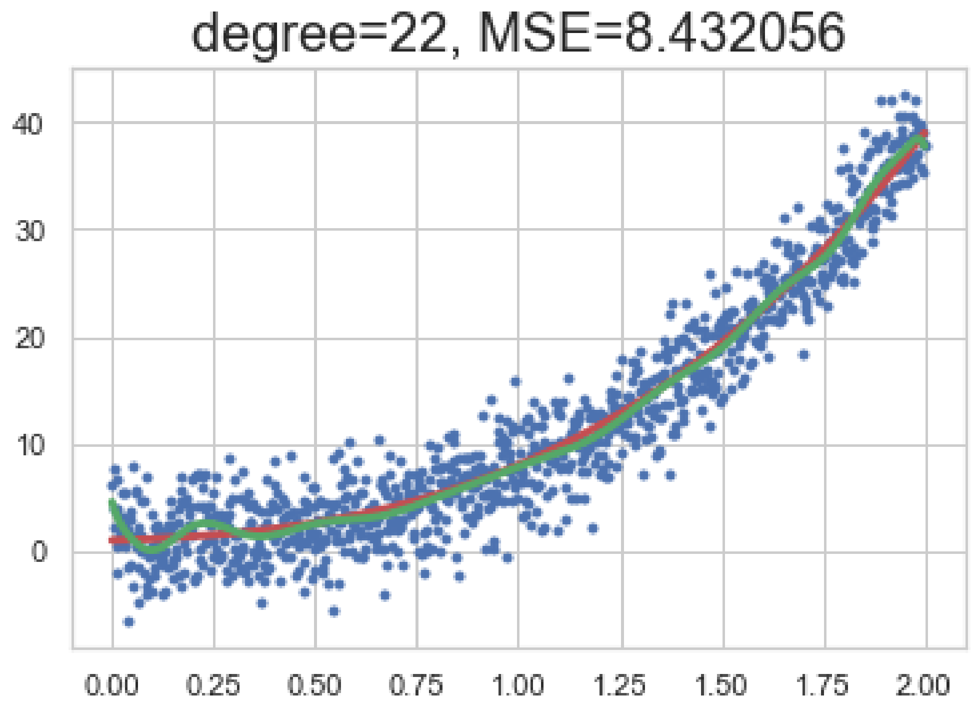
Averaging may not work

the average polynomial model



No improvement, Why ???

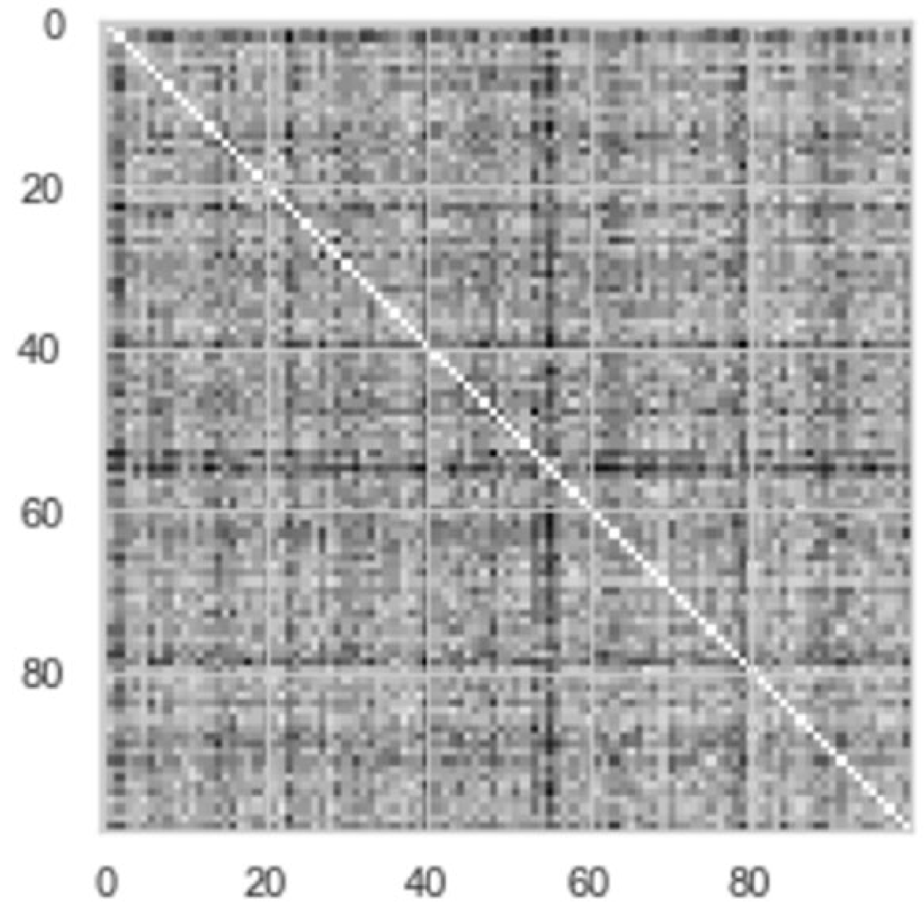
the single polynomial model



Averaging may not work if the models (outputs) are strongly correlated

```
C=np.corrcoef(Yp)
plt.imshow(C, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x1
```



```
C.min()
```

```
0.9976457145926244
```

N random variables, r_1 to r_N

Each one has variance $var(r_n) = \sigma^2$

NOT i.i.d.

nonnegative pairwise correlation ρ in the range of 0 to 1

$\rho = 0$ means no correlation

$\rho = 1$ means the strongest correlation

the average $\bar{r} = \frac{1}{N}\sum_{n=1}^{N} r_n$, N is the number of models

$var(\bar{r}) = \rho\sigma^2 + \frac{1-\rho}{N}\sigma^2$

when $\rho \sim 0$, $var(\bar{r}) \sim \frac{1-\rho}{N}\sigma^2$ increase N => decrease $var(\bar{r})$

when $\rho \sim 1$, $var(\bar{r}) \sim \rho\sigma^2$ independent of N

Decision Tree and Random Forest

- A decision tree partitions the feature space into a set of disjoint regions, and then fit a simple model in each region.

- A random forest is a combination of decision trees by bagging

  reducing variance by

  (1) randomly sampling data points <u>and features</u> when building a tree

  to reduce correlation between trees

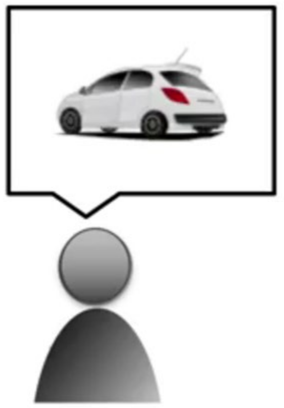
  (2) averaging (bagging) the trees

Decision Tree Example

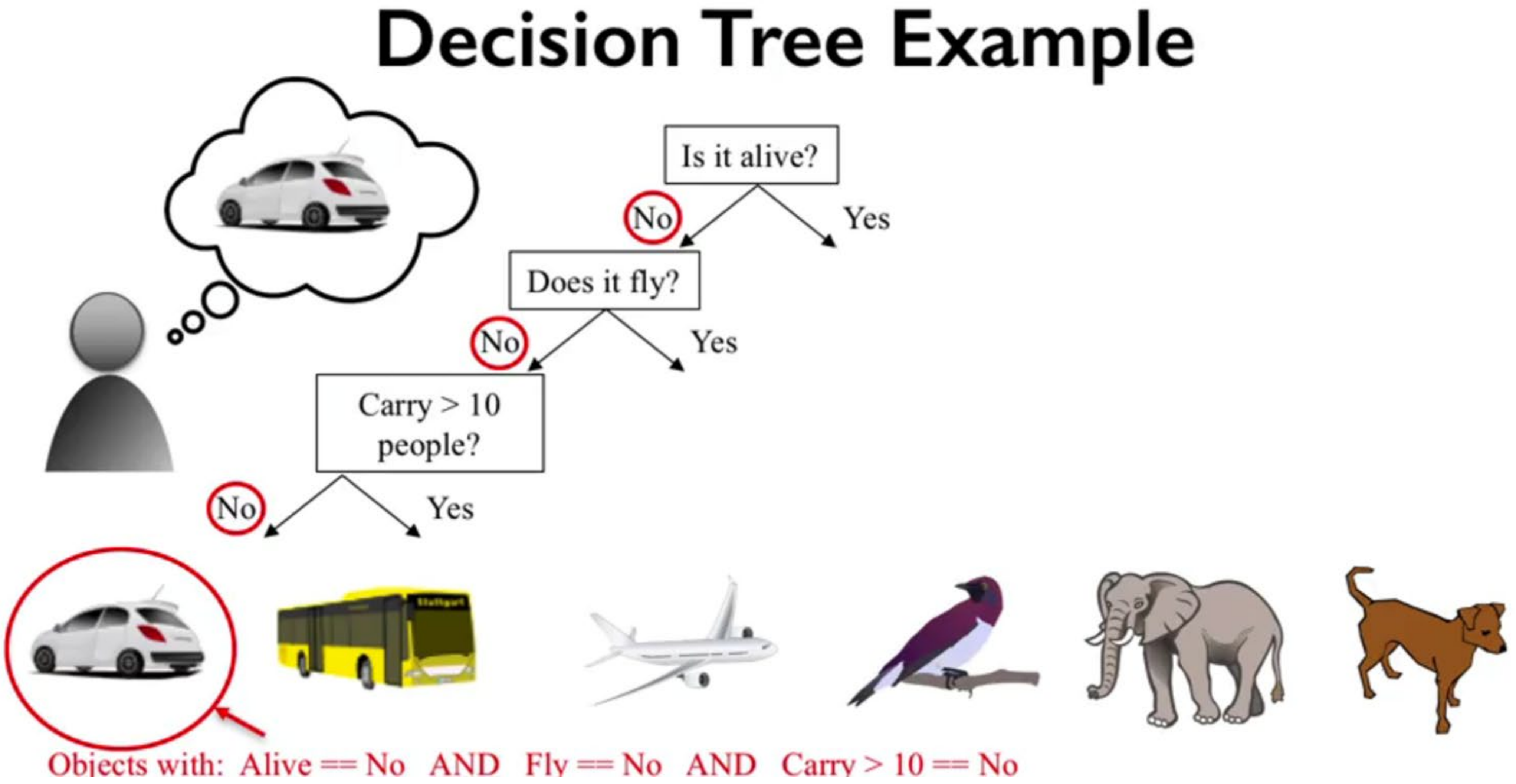
Assume a person only knows the following 6 objects:

Car
Bus
Airplane
Bird
Elephant
Dog

One day, a person saw a picture of an object, then this guy tried to figure out the name of the object using a decision tree.



The person asked many yes-no questions



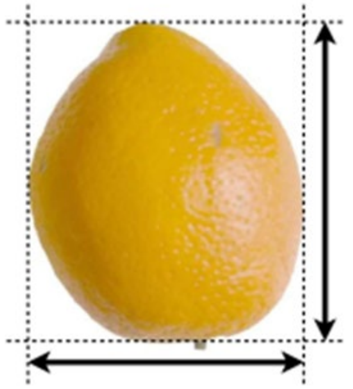
A data sample has three binary features: [alive, fly, carry_more_than_10]

Decision Tree Example

Feature Vector x_n

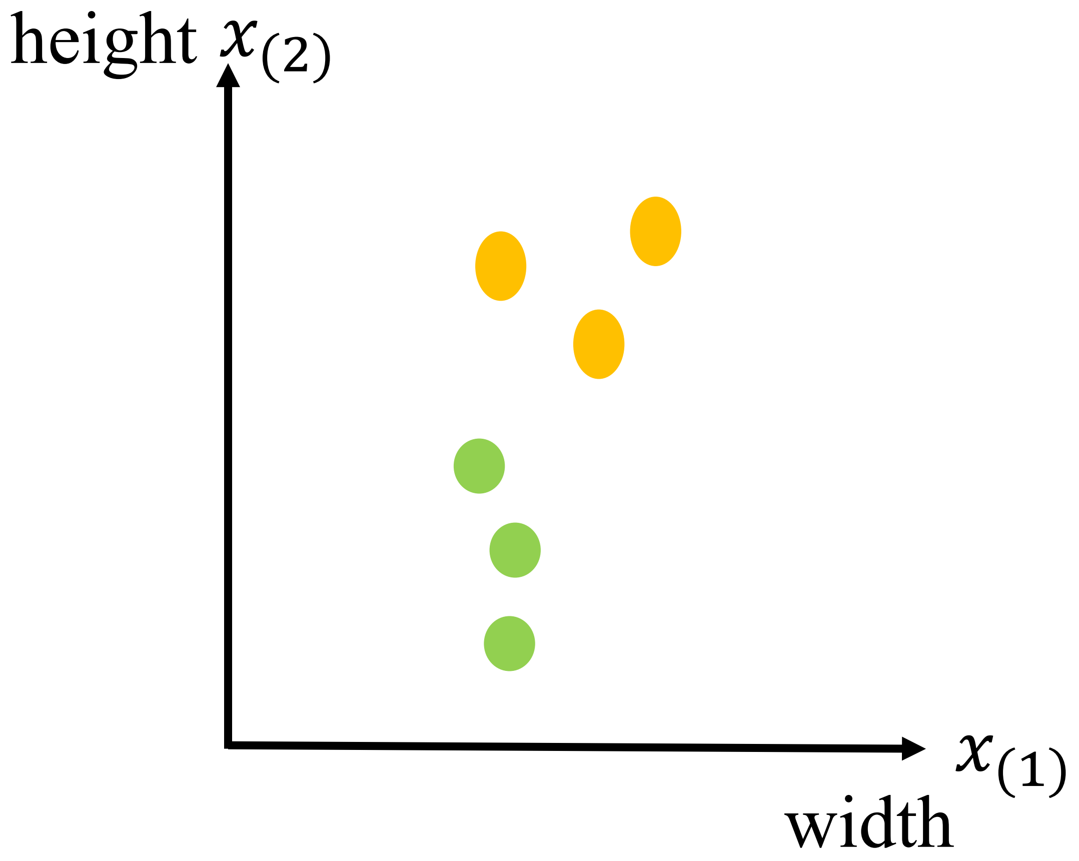
$$x_n = \begin{bmatrix} x_{(n,1)} \\ x_{(n,2)} \end{bmatrix} \begin{matrix} \text{width} \\ \text{height} \end{matrix}$$

$y_n = 1$ It is an apple

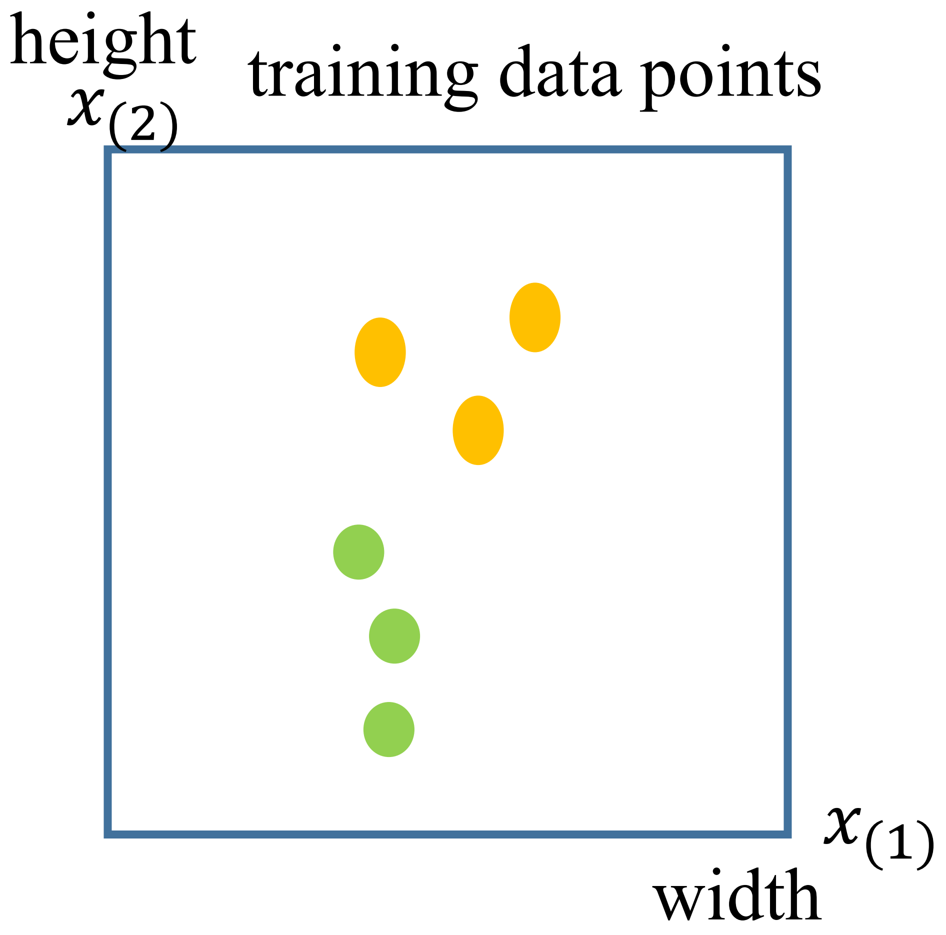


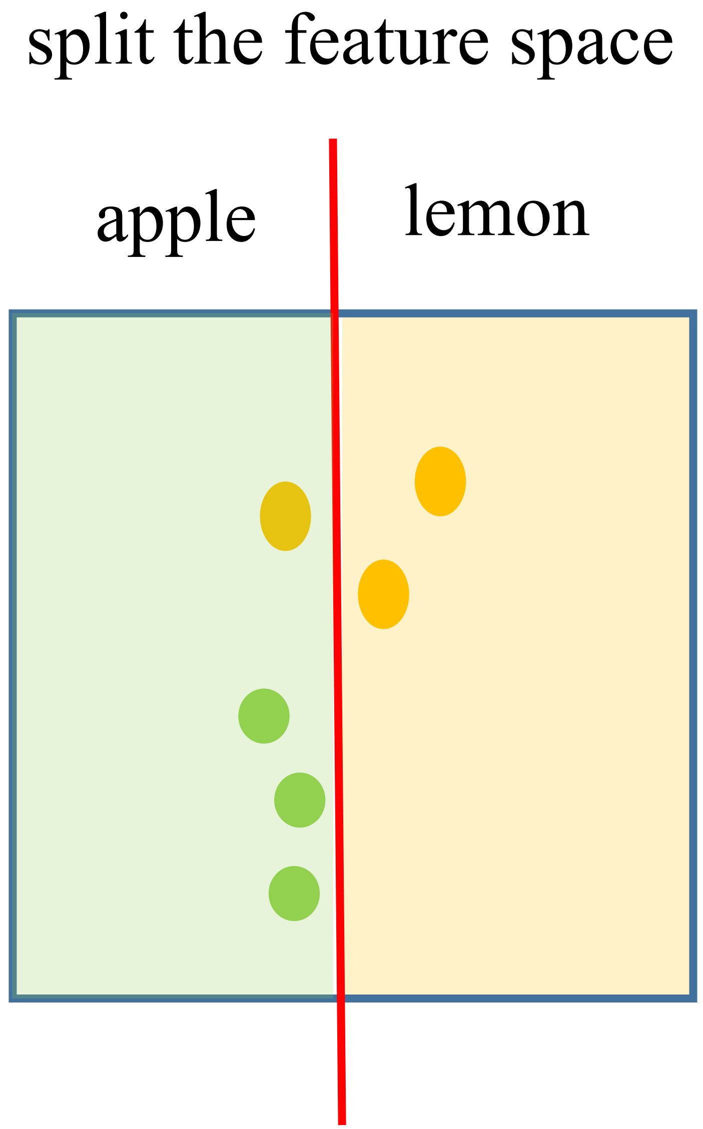
$y_n = 0$ It is a lemon

Feature Space (Input Space)

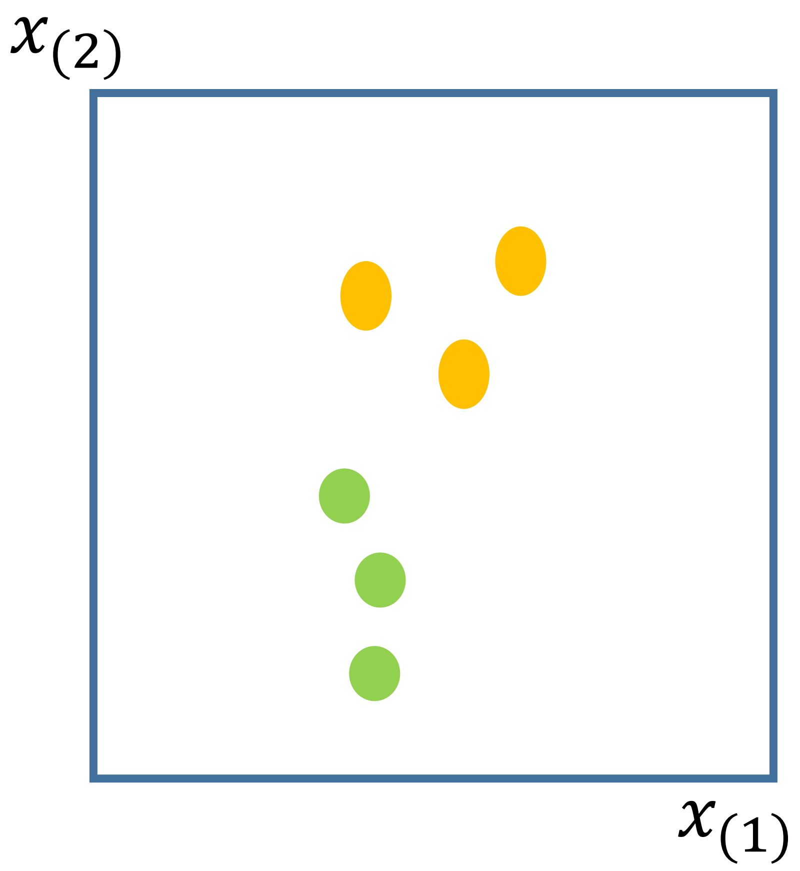


Decision Tree Example

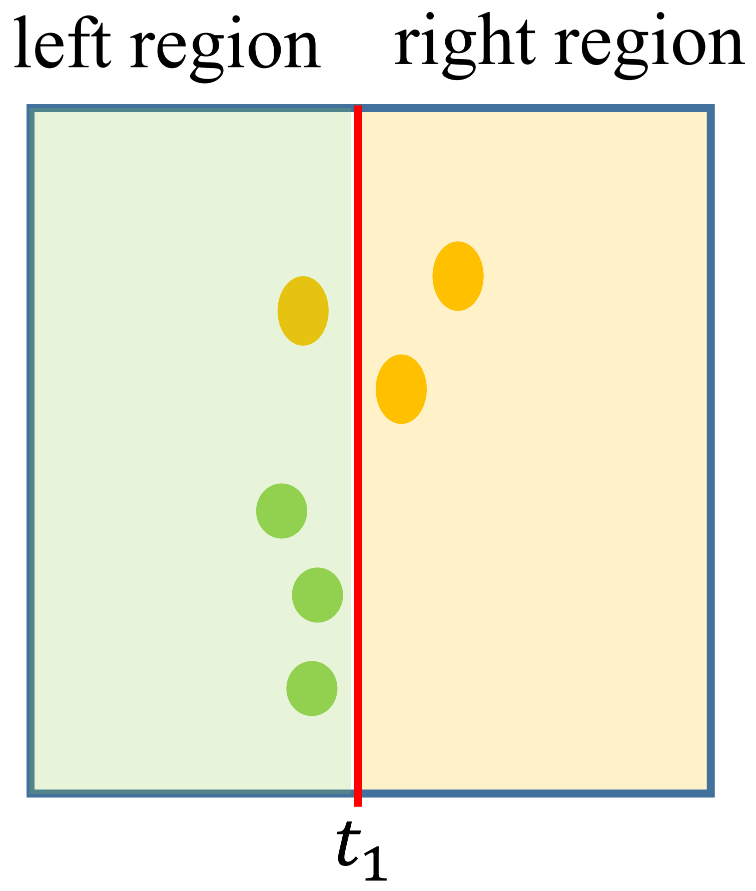




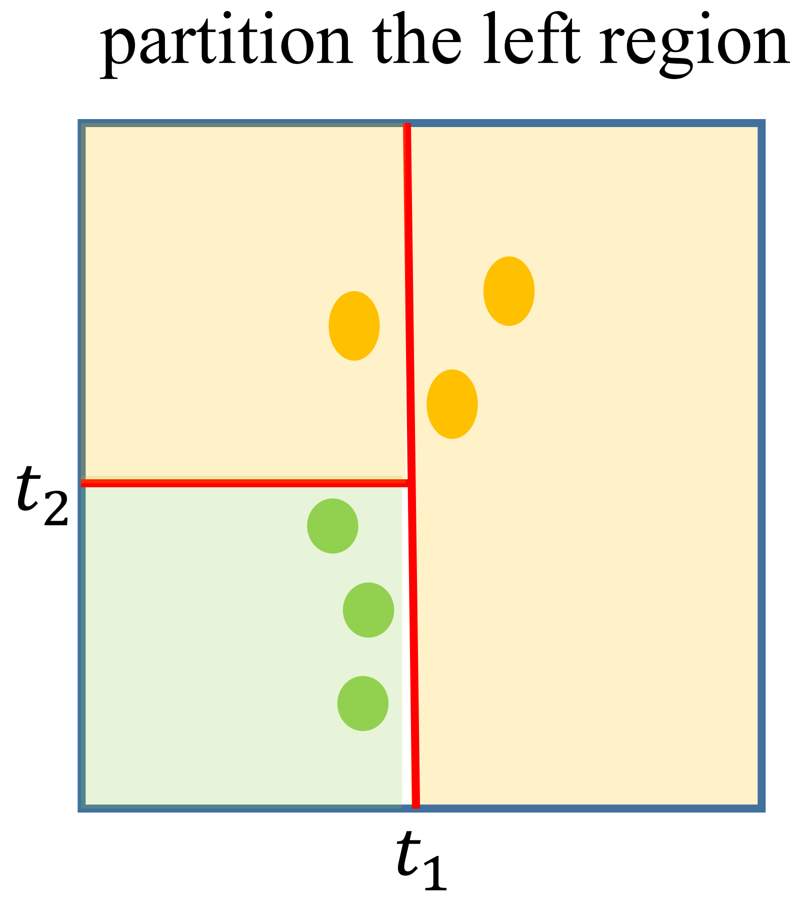
$x_{(1)} \leq t_1$: apple (majority voting)
because there are 3 apples and 1 lemon



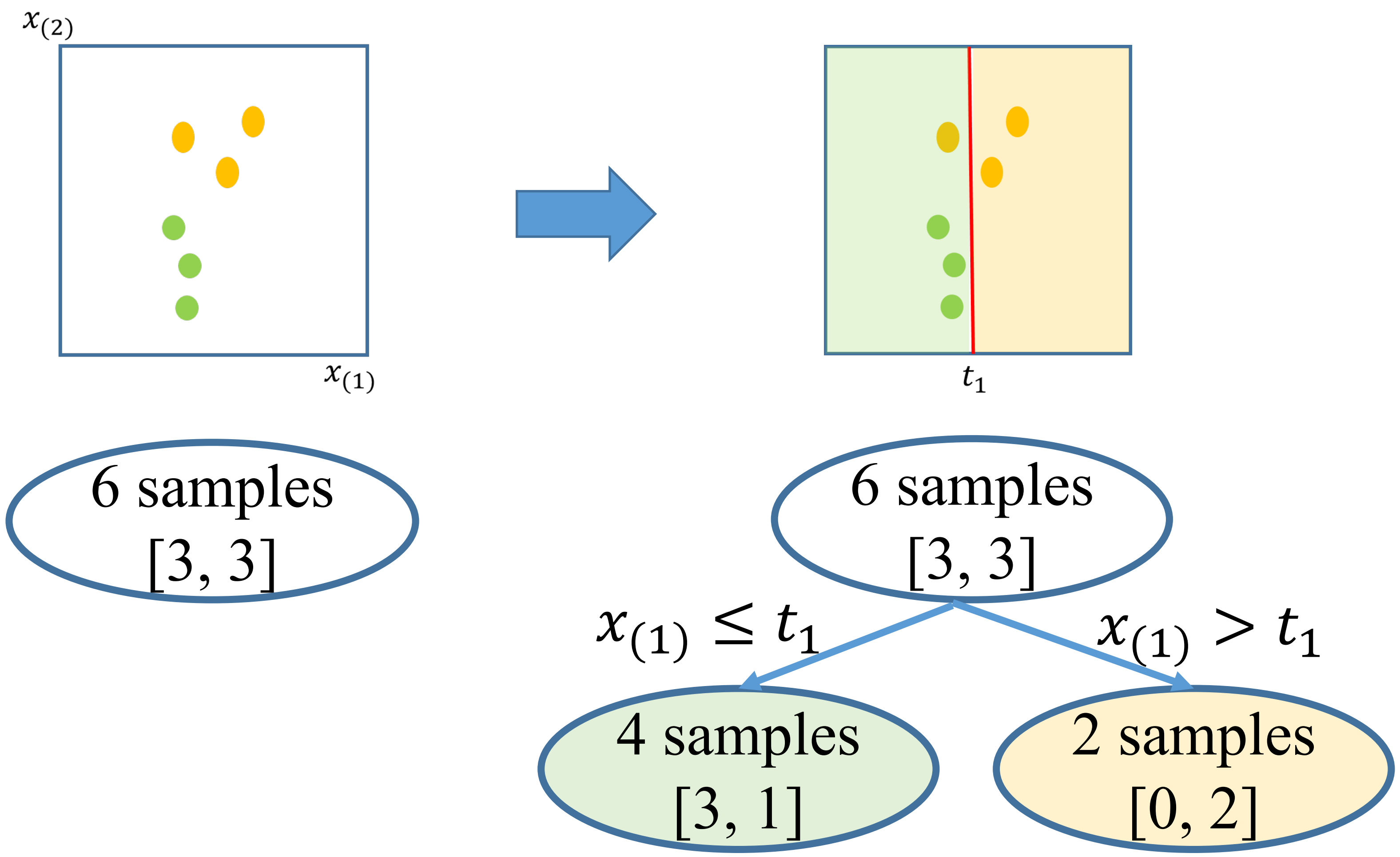
a sample has two features $x_{(1)}$ and $x_{(2)}$



$x_{(1)} \leq t_1$: apple
$x_{(1)} > t_1$: lemon

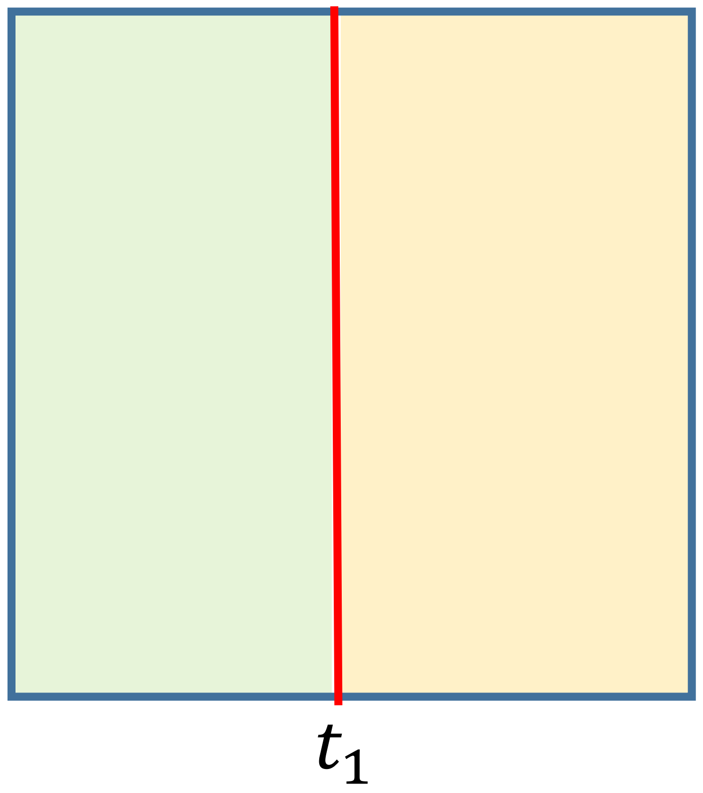


Given $x_{(1)} \leq t_1$,
$x_{(2)} \leq t_2$: apple
$x_{(2)} > t_2$: lemon

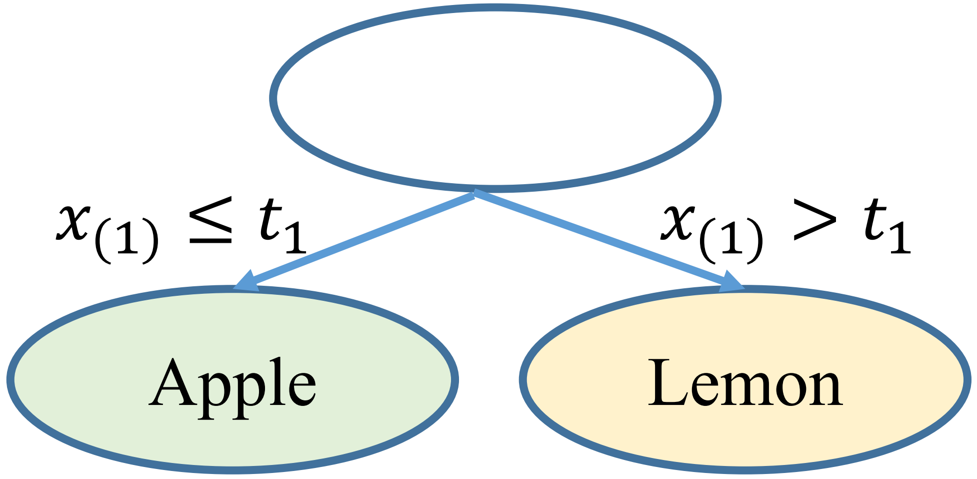


a tree is growing
during training

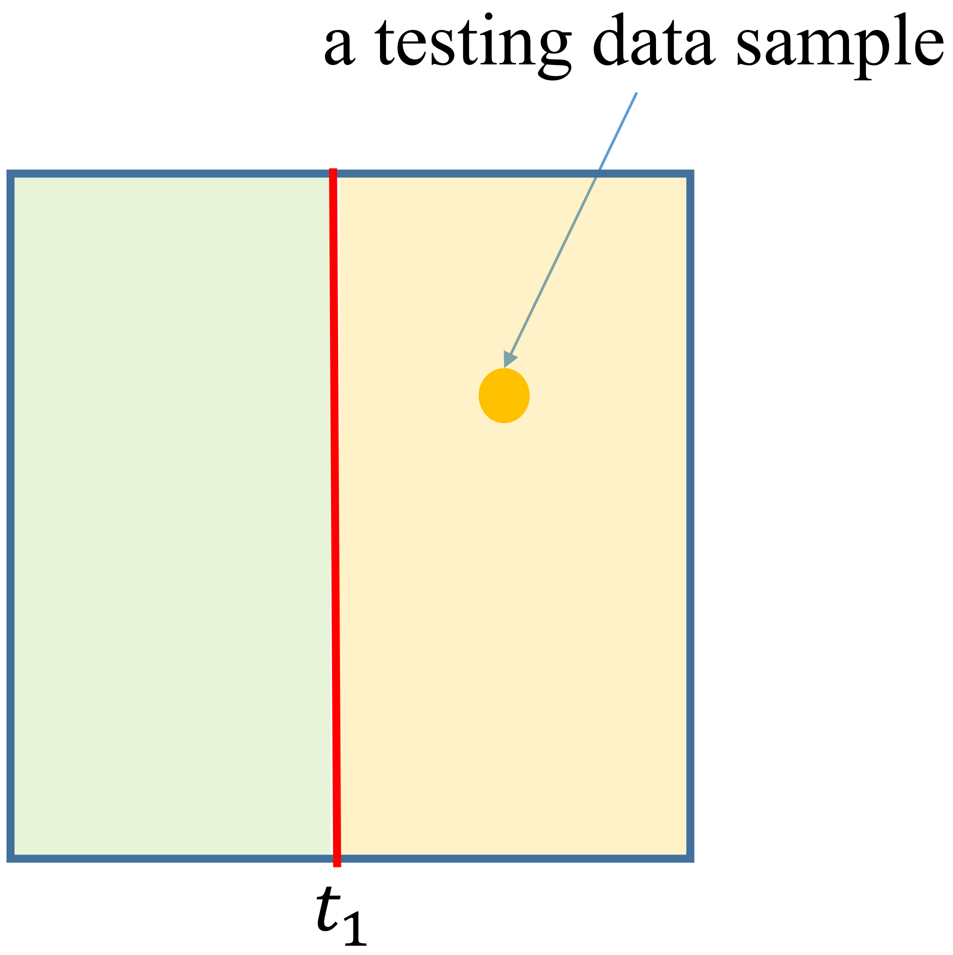
partition of input space

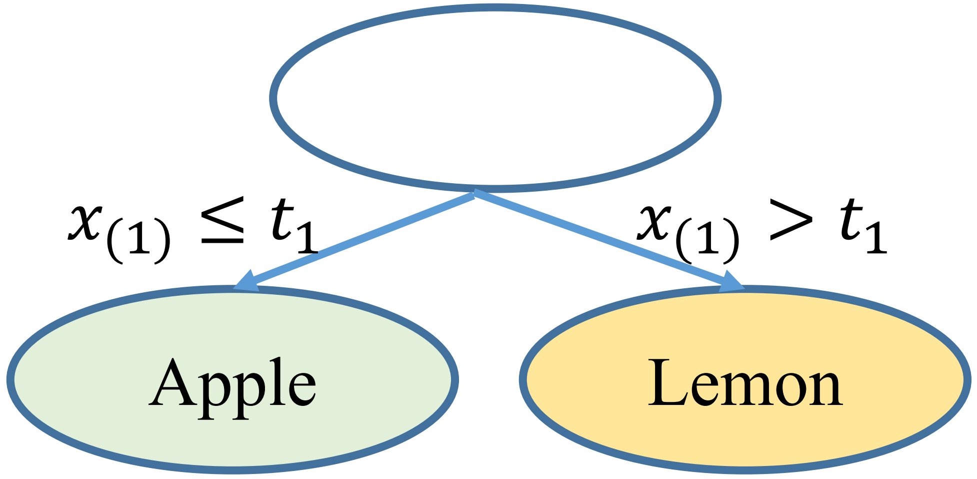


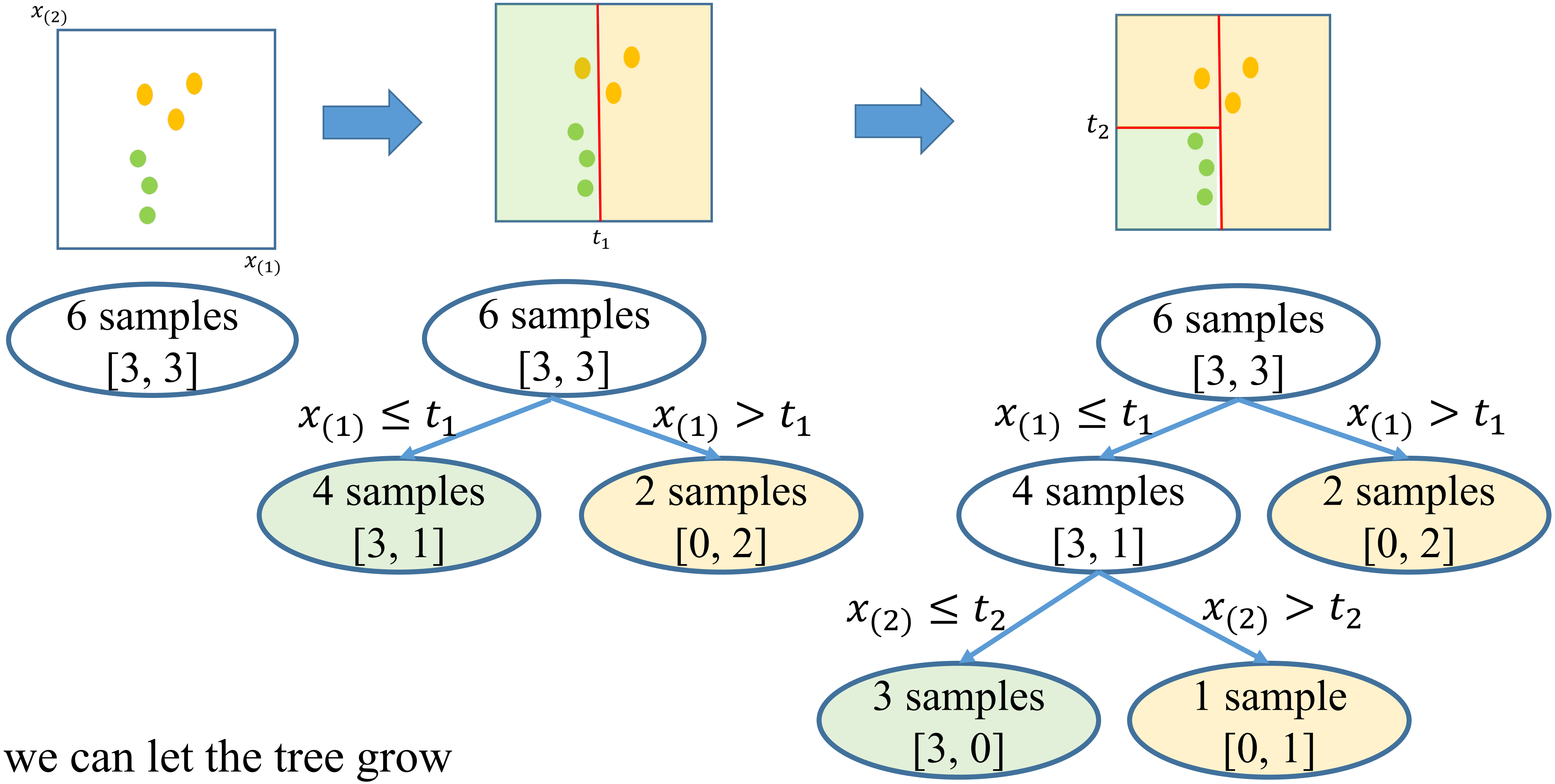
the decision tree T1



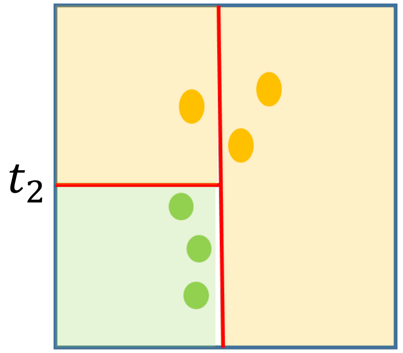
test the decision tree T1





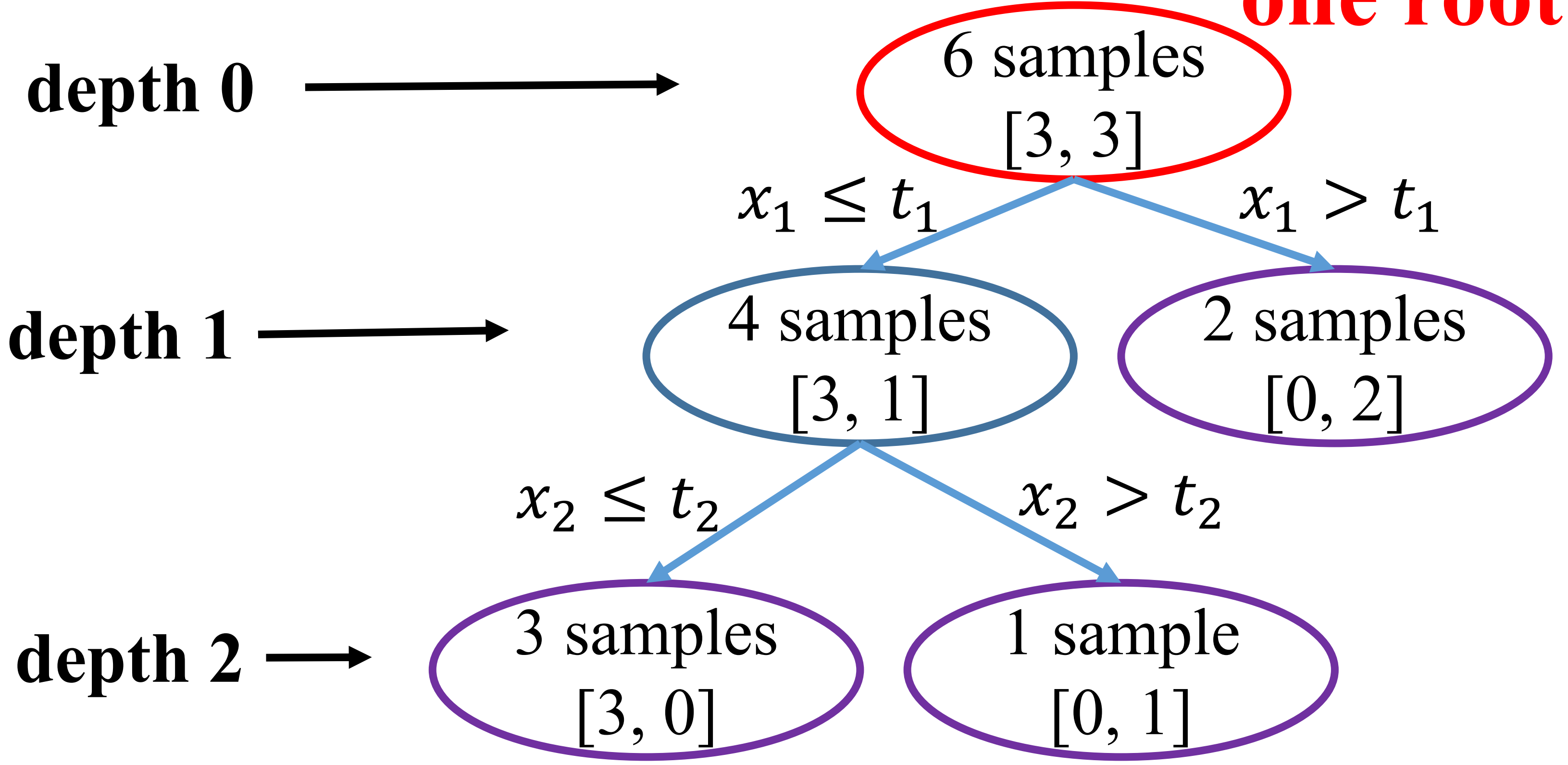


we can let the tree grow another layer



We can further expand this tree such that every leaf node only contains one data sample

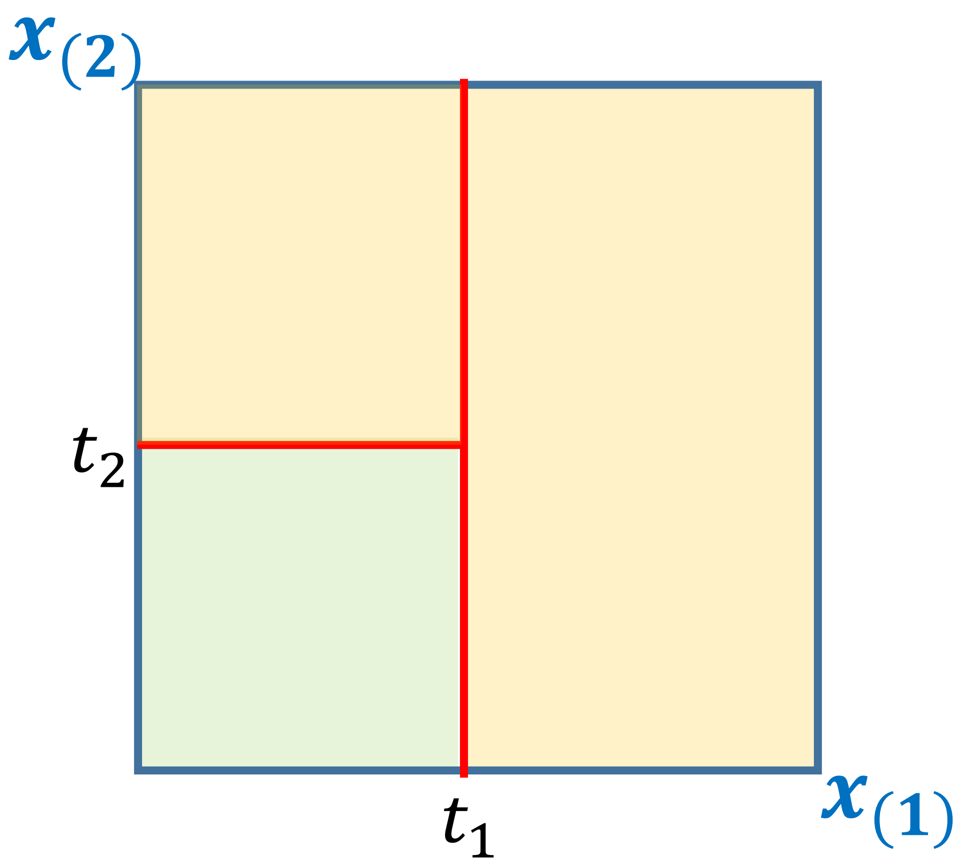
one root node



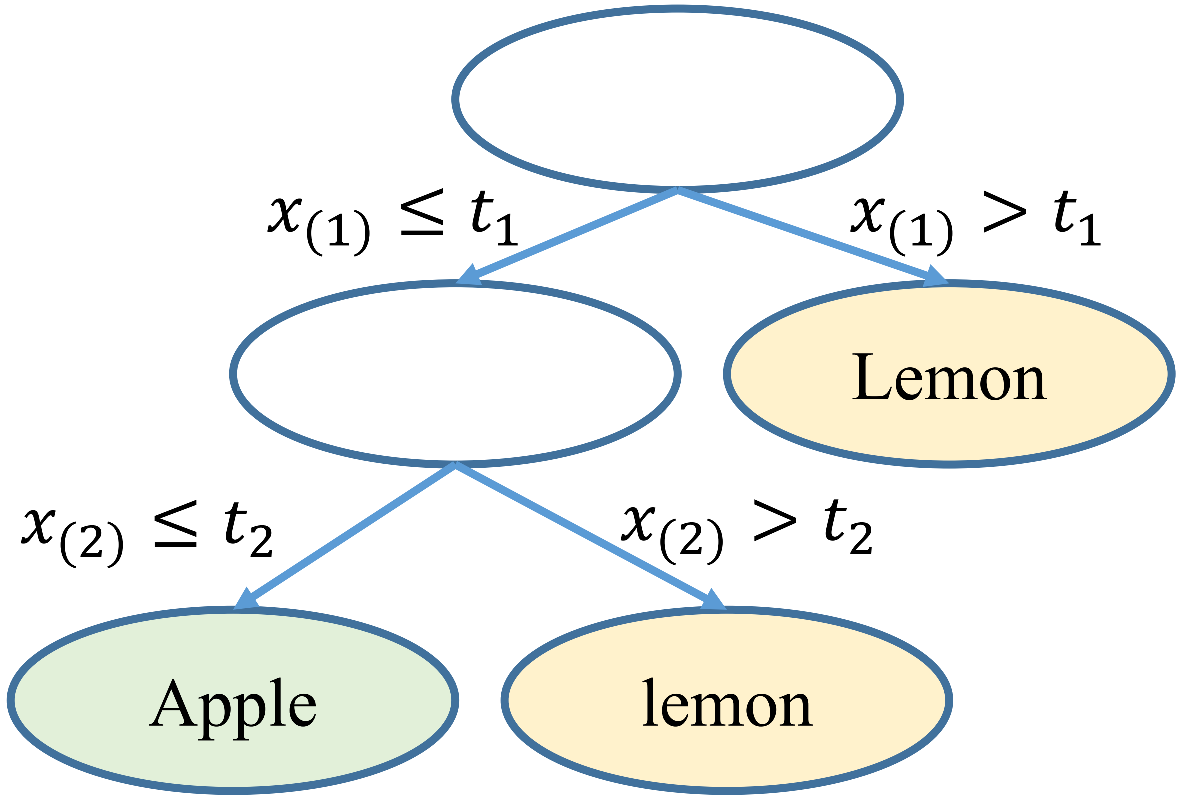
3 leaf (terminal) nodes do not have child nodes

The depth of the tree is 2

partition of input space

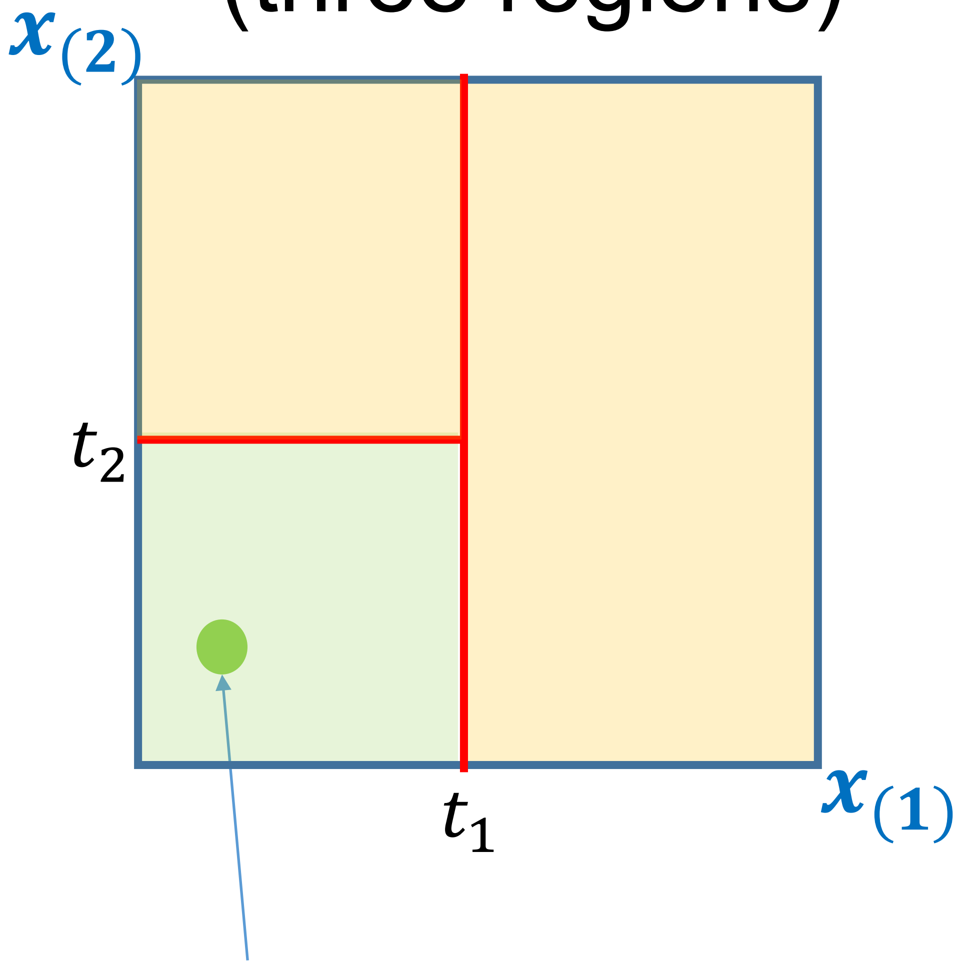


the decision tree T2

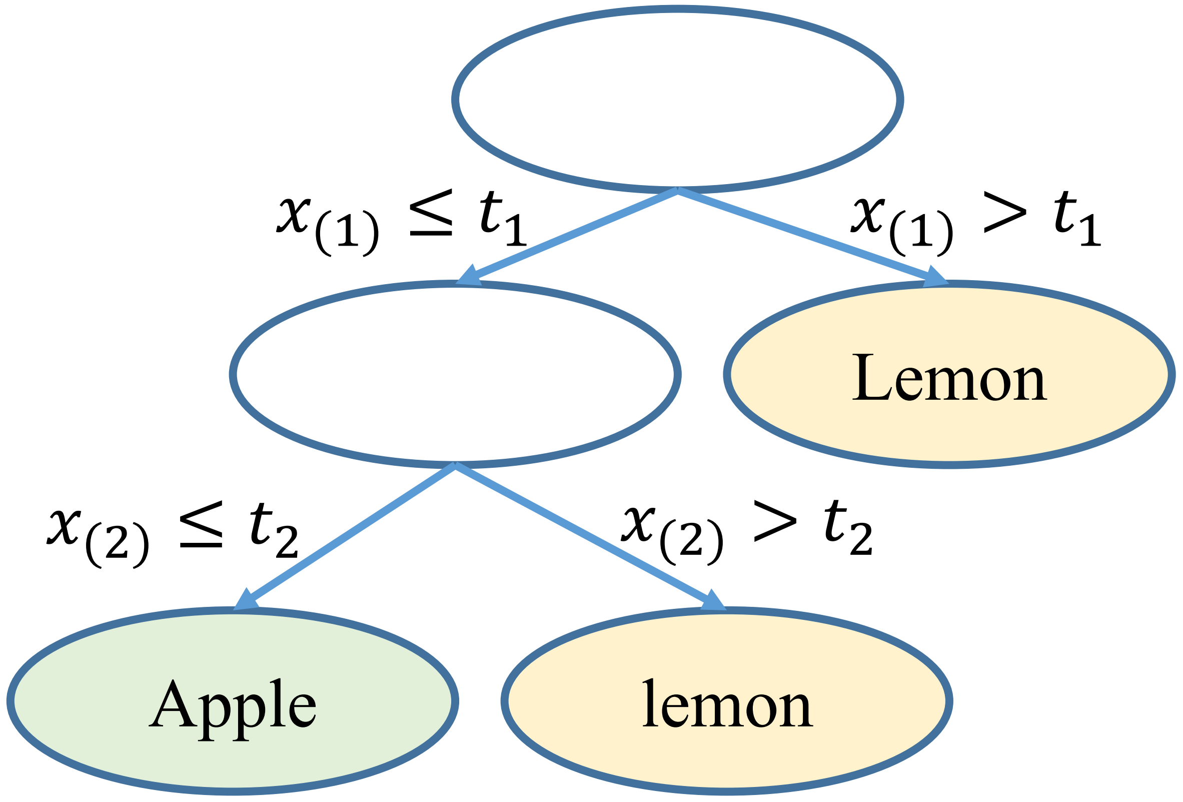


partition of input space (three regions)

use the decision tree T2



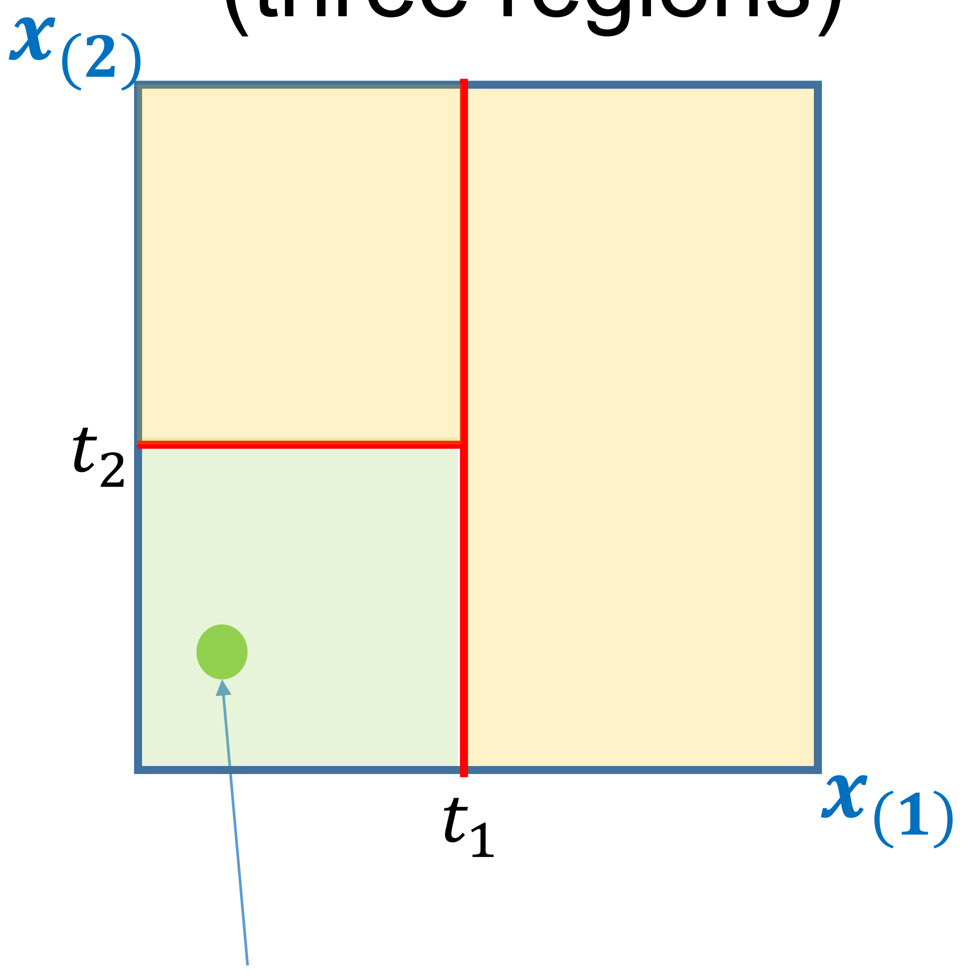
a new data sample x



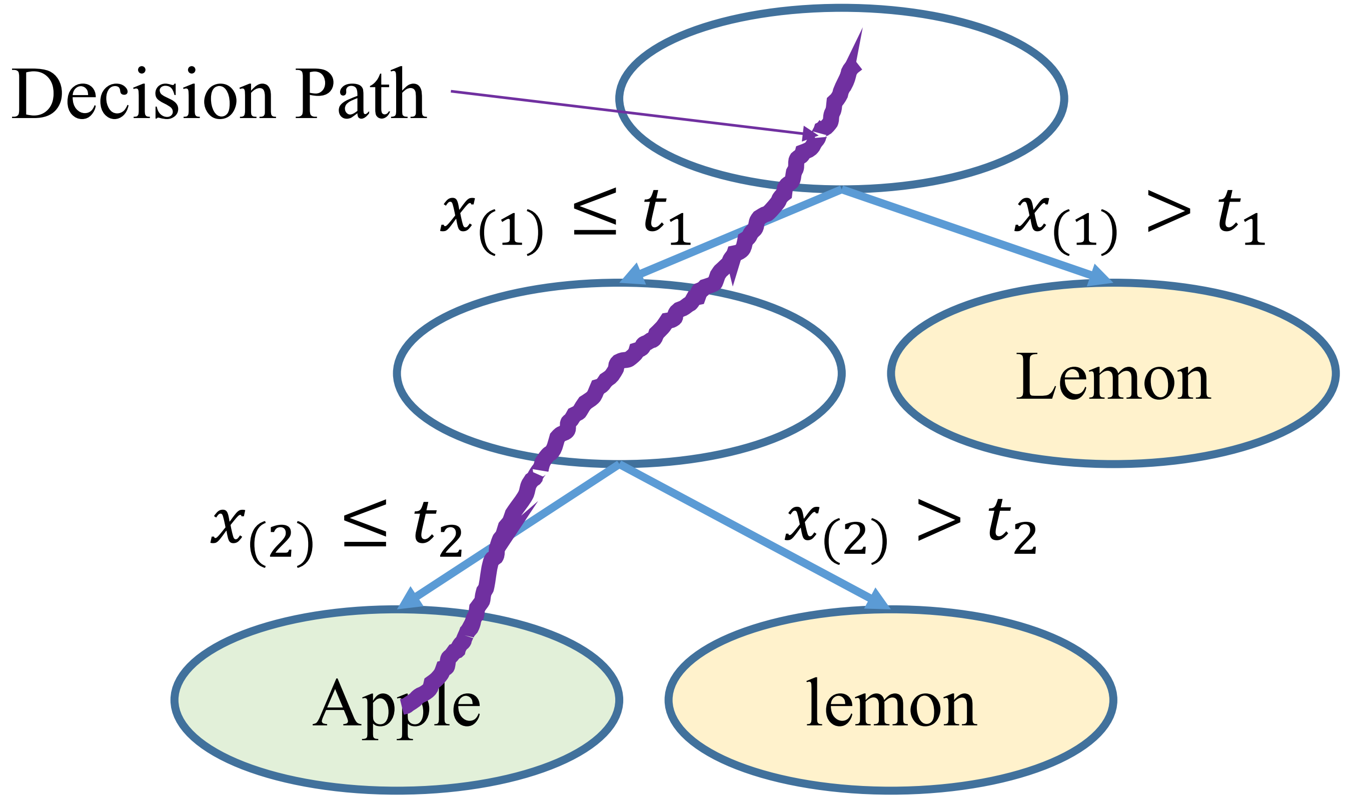
a region corresponds to a leaf node in the tree

partition of input space (three regions)

use the decision tree T2

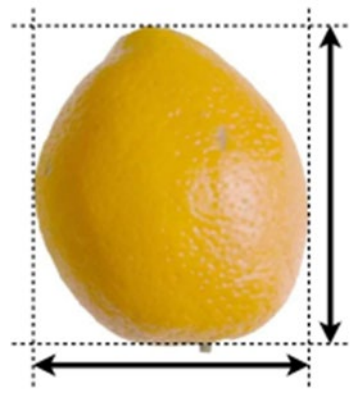


a new data sample x



The data sample x falls into a region/node

Apply Decision Tree to the fruits dataset



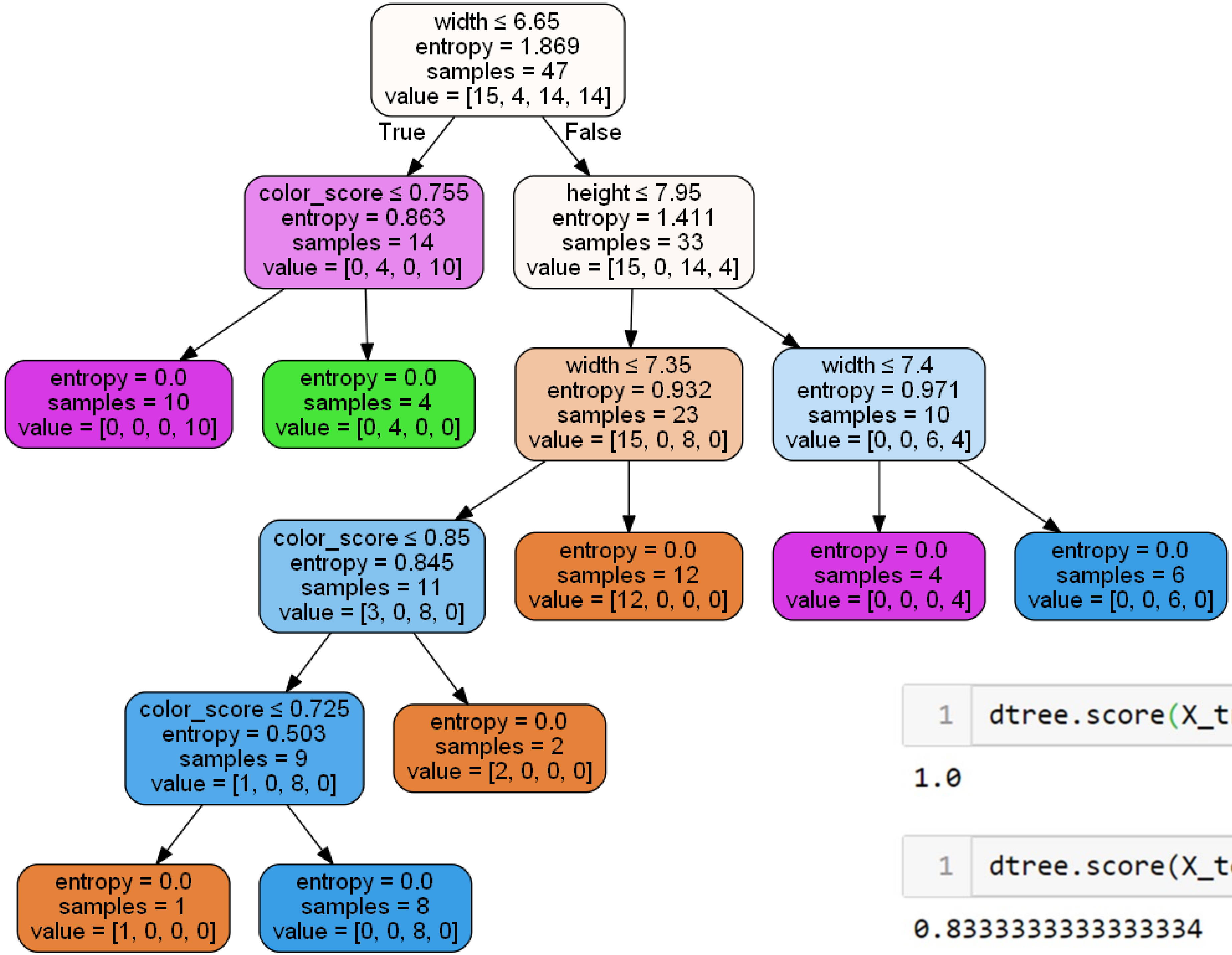
The fruits dataset (a large table)
Each row contains the information of a fruit sample/instance

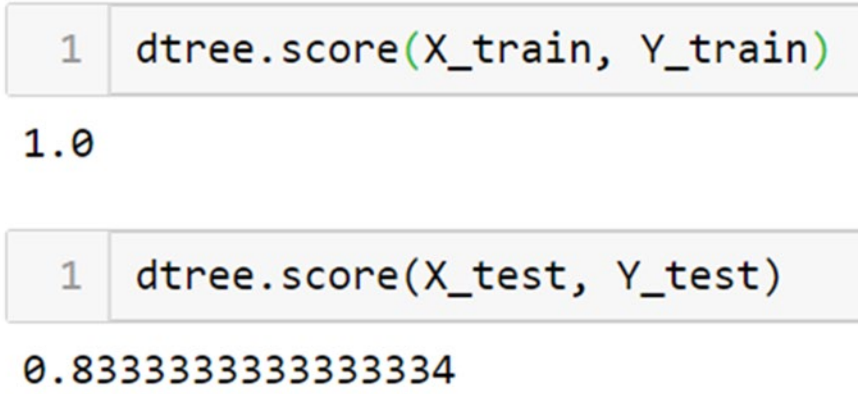
fruit label	fruit_name	subtype	mass (g)	width (cm)	height (cm)	color_score
1	apple	granny_smith	192	8.4	7.3	0.55
4	lemon	spanish_belsan	194	7.2	10.3	0.70

http://usapple.org/the-industry/apple-varieties/

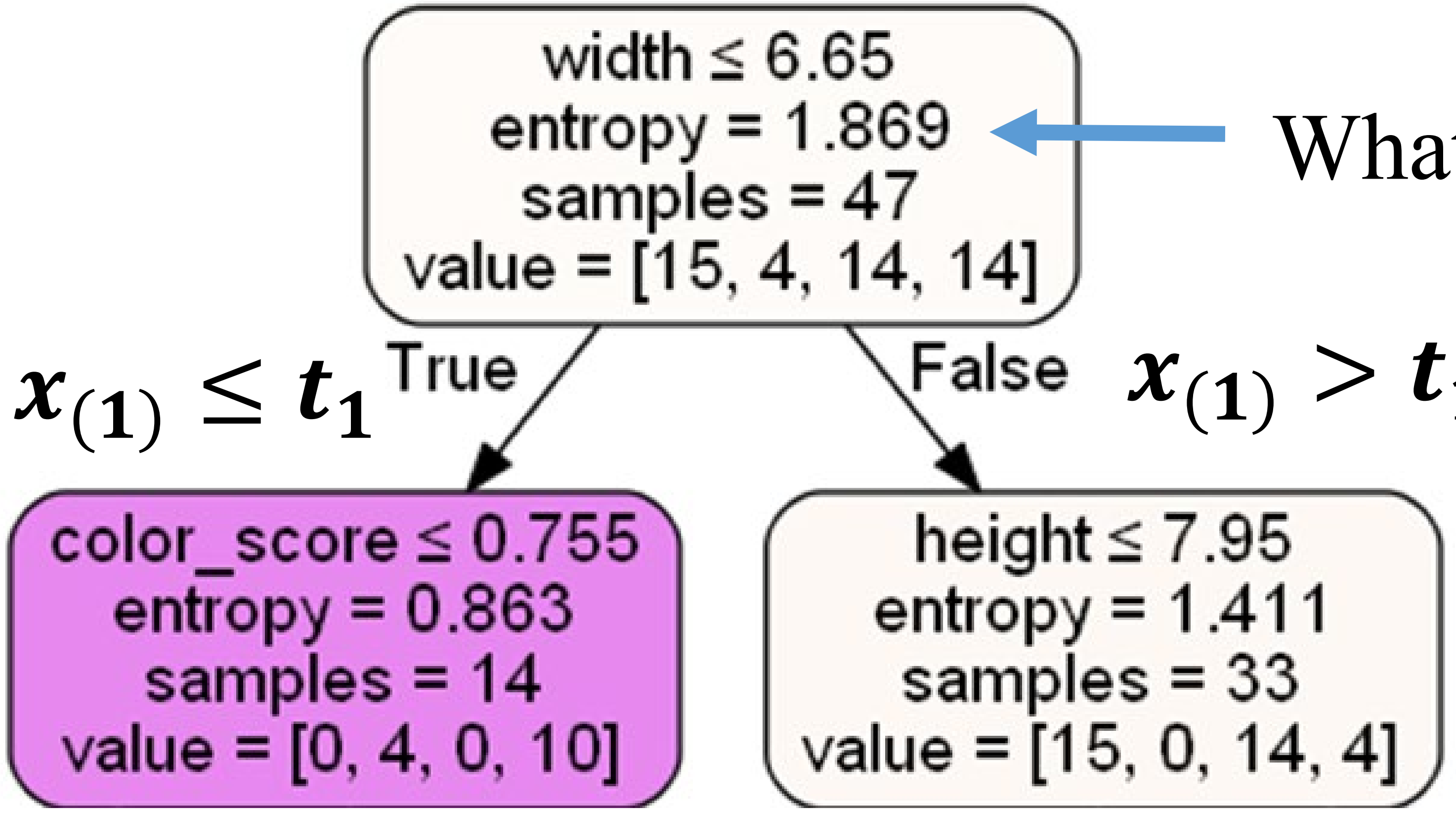
The feature vector of a fruit sample: [width, height, color_score]

1:apple,
2:mandarin
3:orange
4:lemon




```
dtree.score(X_train, Y_train)
```
```
1.0
```
```
dtree.score(X_test, Y_test)
```
```
0.8333333333333334
```

$\boldsymbol{x}_{(1)}$ is width, $\boldsymbol{t}_1 = 6.65$



What is the entropy ?

$\boldsymbol{x}_{(1)} \leq \boldsymbol{t}_1$

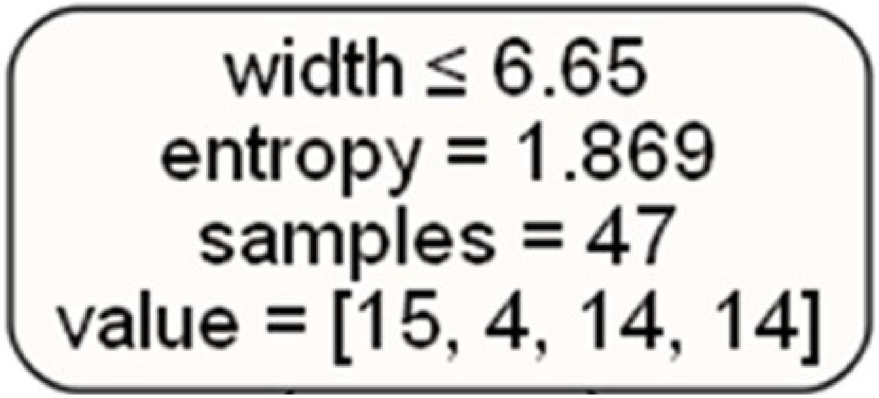
$\boldsymbol{x}_{(1)} > \boldsymbol{t}_1$

$\boldsymbol{x}_{(3)}$ is color_score

$\boldsymbol{t}_3 = 0.755$

$\boldsymbol{x}_{(2)}$ is height

$\boldsymbol{t}_2 = 7.95$



width ≤ 6.65
entropy = 1.869
samples = 47
value = [15, 4, 14, 14]

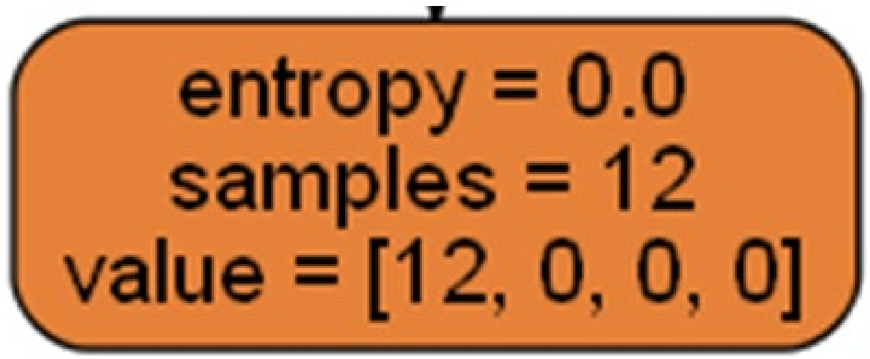
On this node:
the distribution (PMF) over the 4 classes is

$$[p_1 \quad p_2 \quad p_3 \quad p_4]$$

PMF: probability mass function

$$p_1 = \frac{15}{47}, p_2 = \frac{4}{47}, p_3 = \frac{14}{47}, p_4 = \frac{14}{47}$$

Entropy $H(p) = -\sum_{k=1}^{4} p_k \, log_2(p_k) = 1.869$



On this node:
the distribution (PMF) over the 4 classes is

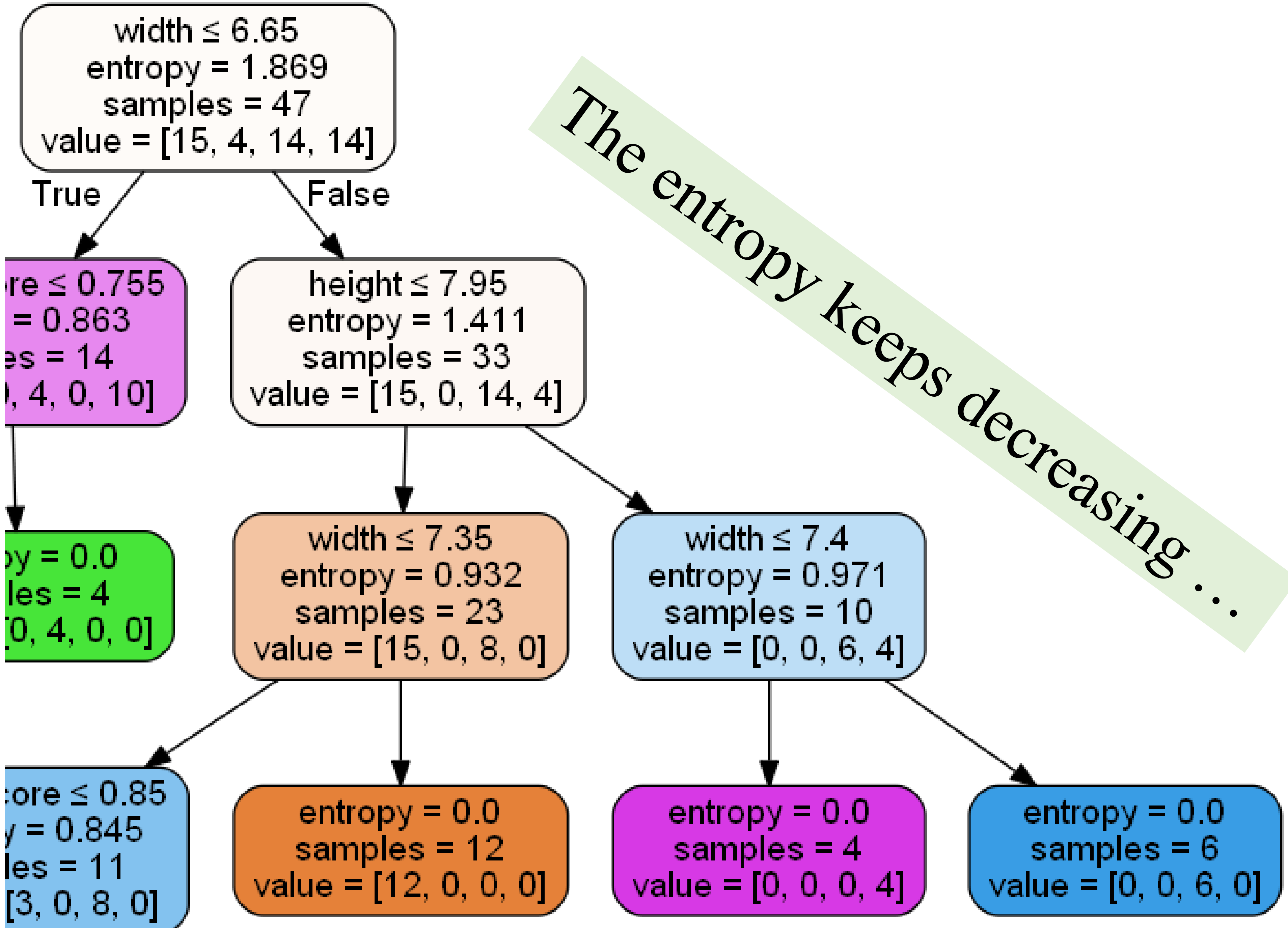
$$[p_1 \quad p_2 \quad p_3 \quad p_4]$$

$$p_1 = \frac{12}{12} = 1 \,, p_2 = 0 \,, p_3 = 0 \,, p_4 = 0$$

Entropy $H(p) = -\sum_{k=1}^{4} p_k \, log_2(p_k) = 0$

Define: $0 \, log_2(0) \equiv 0$

This node only contains apples ($p_1 = 1$).
It is a pure node (of apples)



The entropy keeps decreasing …

If we only have a limited number of samples on node-j, then we only have an estimation of the distribution (PMF) over the 4 classes, which is

$$[\hat{p}_{(j,1)} \quad \hat{p}_{(j,2)} \quad \hat{p}_{(j,3)} \quad \hat{p}_{(j,4)}]$$

$\hat{p}_{(j,k)} = \frac{1}{N_j} \sum_{x_n \in R_j} I_k(y_n)$

$I_k(y_n) = 1$ if $y_n = k$

$I_k(y_n) = 0$ if $y_n \neq k$

k: the index of class k

j: the index of region/node j

N_j is the number of training samples in node/region R_j

p_k is the true probability

$p_k = \hat{p}_k$ when we have a large number of samples

drop the node index $\hat{p}_k = \hat{p}_{(j,k)}$

- So, in previous examples on entropy calculation, we should use

$[\hat{p}_1 \quad \hat{p}_2 \quad \hat{p}_3 \quad \hat{p}_4]$ PMF estimated from data

instead of

$[p_1 \quad p_2 \quad p_3 \quad p_4]$ the 'true' PMF

because we only have a small number of (training) samples on each node

On node-j

The distribution (PMF) over the 4 classes is

$$[\hat{p}_{(j,1)} \quad \hat{p}_{(j,2)} \quad \hat{p}_{(j,3)} \quad \hat{p}_{(j,4)}]$$

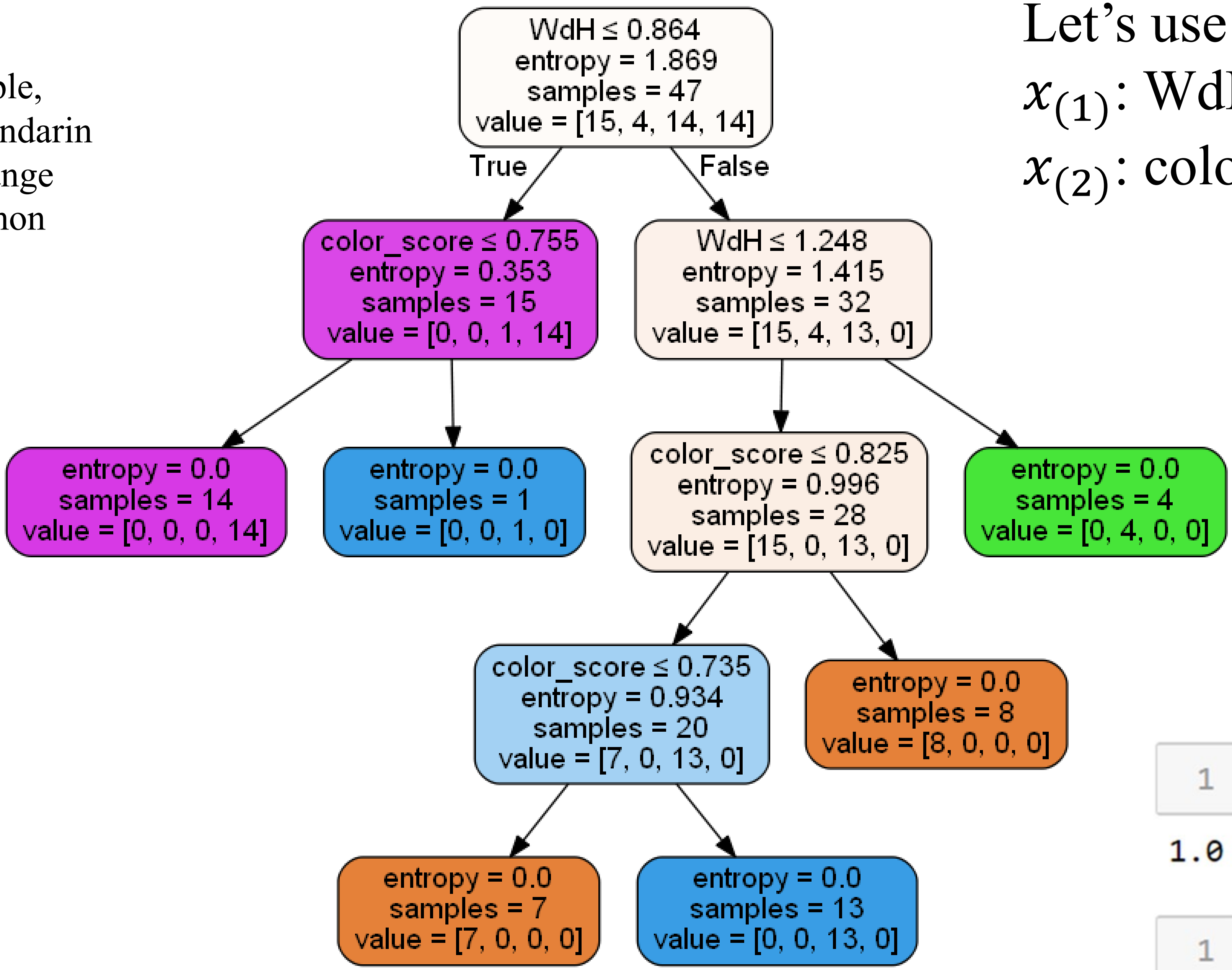
Entropy: $H = -\sum_{k=1}^{K} \hat{p}_{(j,k)} log \hat{p}_{(j,k)}$

If $H = 0$, then the node is a pure node

If $H > 0$, then the node is not pure,
it contains samples from many classes

Thus, Entropy measures the **impurity** of a node.
Impurity: condition of being impure (not pure)

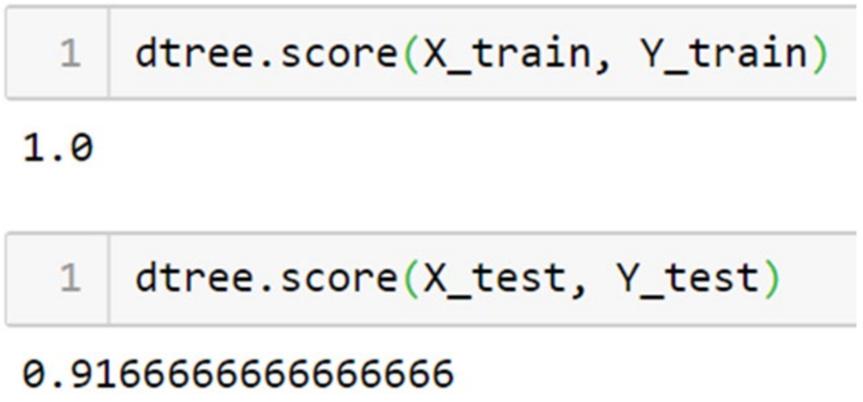
1:apple,
2:mandarin
3:orange
4:lemon



Let’s use two features per sample

$x_{(1)}$: WdH= width/height

$x_{(2)}$: color_score



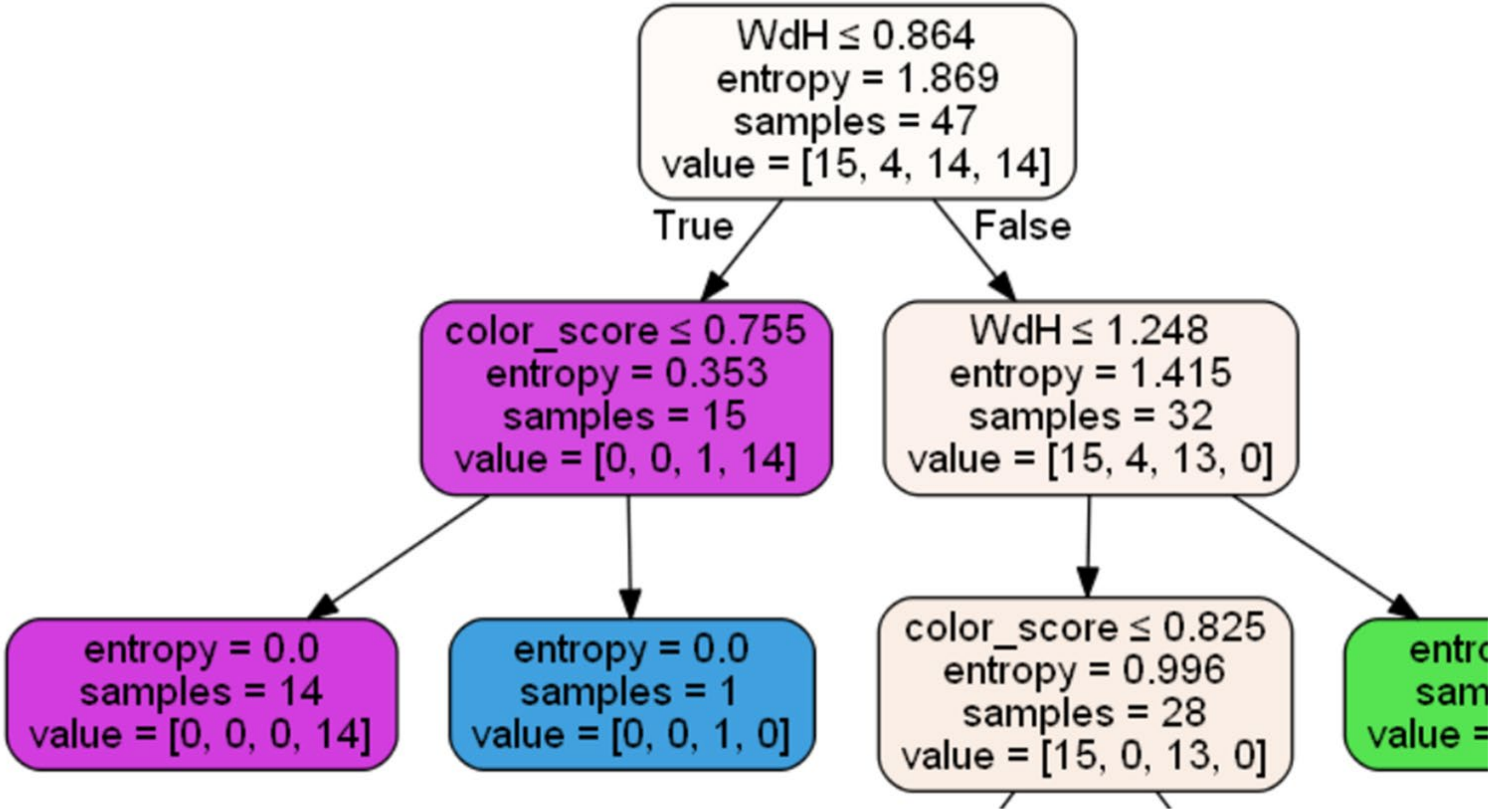
```
dtree.score(X_train, Y_train)
```

```
1.0
```

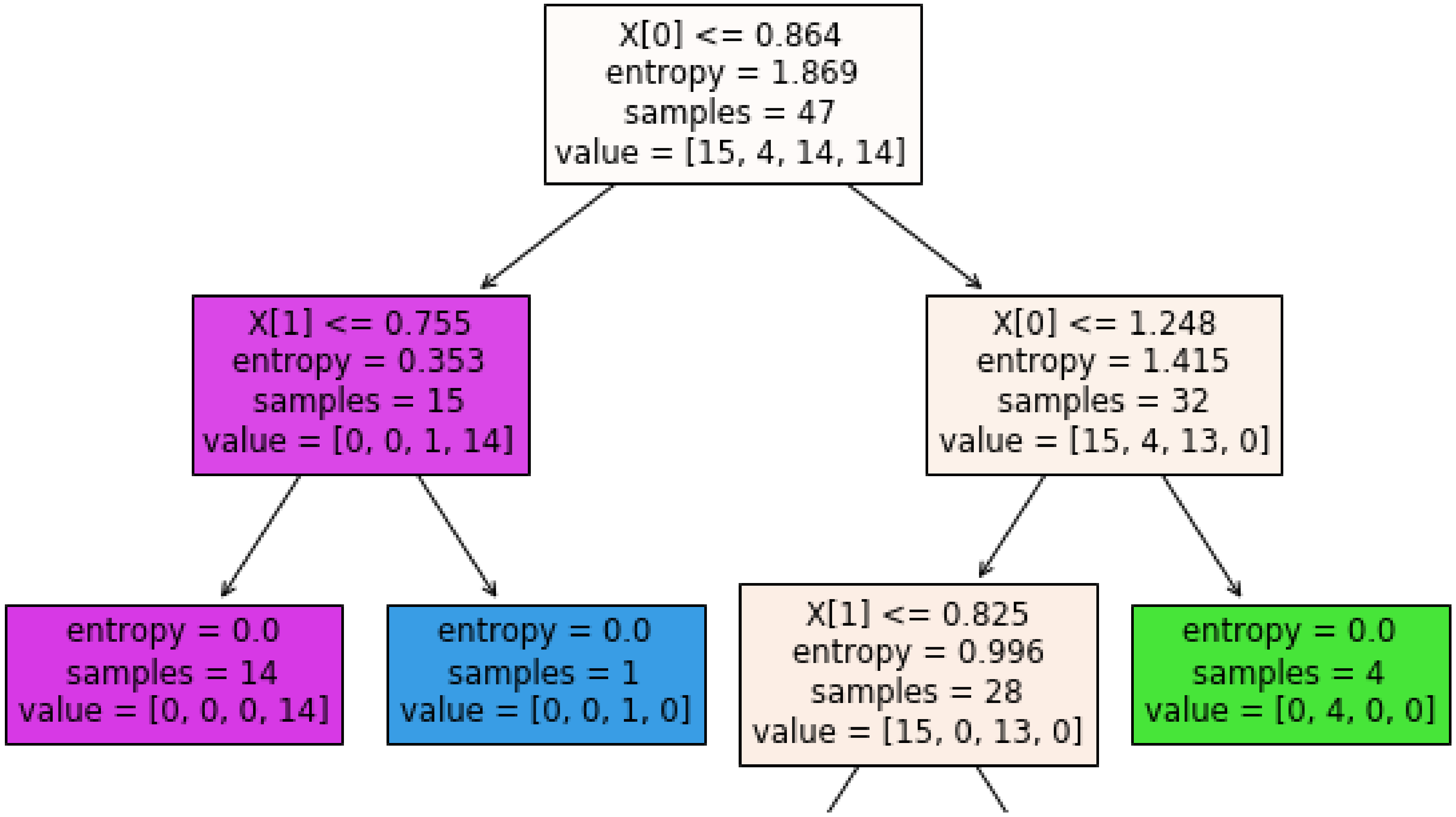
```
dtree.score(X_test, Y_test)
```

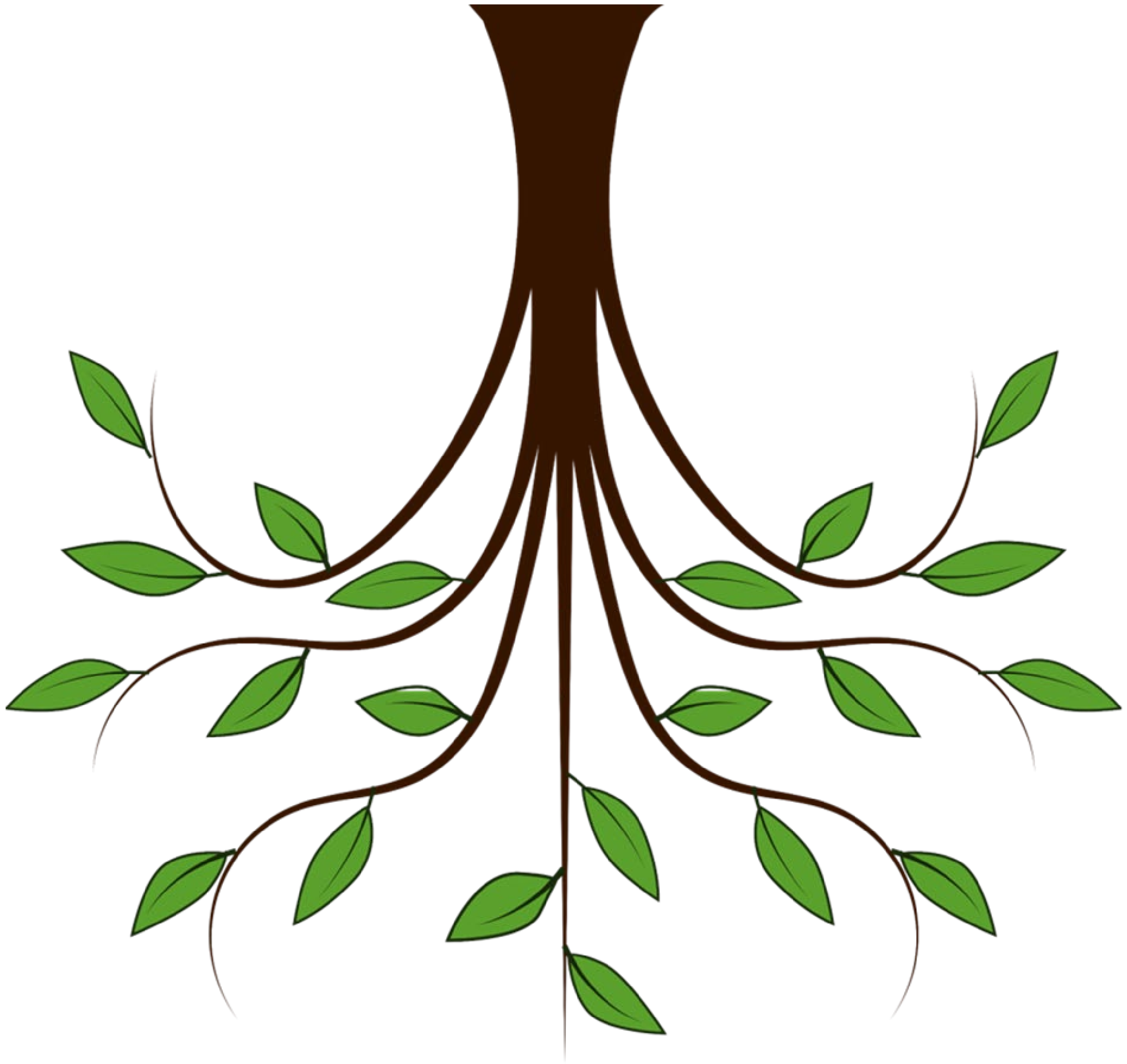
```
0.9166666666666666
```

visualize a decision tree: using graphviz



visualize a decision tree: using plot_tree in sk-learn





Use a trained decision tree for classification

- A training dataset $\{(x_n, y_n), n = 1, \dots, N\}$ and $x_n \in \mathcal{R}^M$

  $y_n = 1,2, \dots, K$, there are K classes
- After training, the feature space $\mathcal{R}^M$ is partitioned into J regions, $R_1, R_2, \dots, R_j, \dots R_J$: A tree is a partition of the feature space
- Each region corresponds to a leaf/terminal node of the tree
- At each node j, we have a distribution $\hat{p}_{(j,k)}$
- The predicted target label $\hat{y}$ of the data point x is given by

$$\hat{y} = \underset{k}{\operatorname{argmax}}\{\hat{p}_{(j,k)}\}$$

if x falls into the node/region-j

Construct/train a decision tree for classification

- A training dataset $\{(x_n, y_n), n = 1, \ldots, N\}$ and $x_n \in \mathcal{R}^M$

  $y_n = 1,2, \ldots, K$, there are K classes

- Let R_0 be the feature space (the input space).

  N_0 is the number of data points in the region R_0

- We will partition the space into two regions R_1 and R_2

  N_i is the number of data points in the region R_i

- step-1: **randomly** select **a subset of features**

$$\{x_{(1)}, x_{(2)}, \ldots\}$$

Construct/train a decision tree for classification

- $Q\left(R_j\right)$ measures the impurity of the node j
  - entropy $H = -\sum_{k=1}^{K} \hat{p}_{(j,k)} log \hat{p}_{(j,k)}$
  - Gini index: $\sum_{k=1}^{K} \hat{p}_{(j,k)}\left(1 - \hat{p}_{(j,k)}\right)$ very similar to entropy

Construct/train a decision tree for classification

- step-2: for each candidate feature $x_{(s)}$ in the subset, find the best split $t_{(s)}$ such that this function is <u>maximized</u>:

$$E = Q(R_0) - \frac{N_1}{N_0} Q(R_1) - \frac{N_2}{N_0} Q(R_2)$$

  - $R_1 = \{x_{(s)} \leq t_{(s)}\}$ and $R_2 = \{x_{(s)} > t_{(s)}\}$
  - $Q(R_j)$ measures the impurity of the node j
  - E *is* maximized when $Q(R_1)$ and $Q(R_2)$ are minimized
- step-3: use the best feature and split (leading to the <u>maximum</u> of E) to divide the region R_0 into R_1 and R_2

- Repeat the above steps until we get many regions/nodes

a closer look at the step 2

$$E(t_{(s)}) = Q(R_0) - \frac{N_1}{N_0} Q(R_1) - \frac{N_2}{N_0} Q(R_2)$$

$R_1 = \{x_{(s)} \leq t_{(s)}\}$ and $R_2 = \{x_{(s)} > t_{(s)}\}$

Assume that three features are randomly selected from the M features:

feature	$x_{(1)}$	$x_{(2)}$	$x_{(11)}$
the best split	$t_{(1)}$	$t_{(2)}$	$t_{(11)}$
objective	$E(t_{(1)})$	$E(t_{(2)})$	$E(t_{(11)})$

if $E(t_{(11)}) = \max\{E(t_{(1)}), E(t_{(2)}), E(t_{(11)})\}$
then $x_{(11)}$ and $t_{(11)}$ will be used for splitting the region R_0

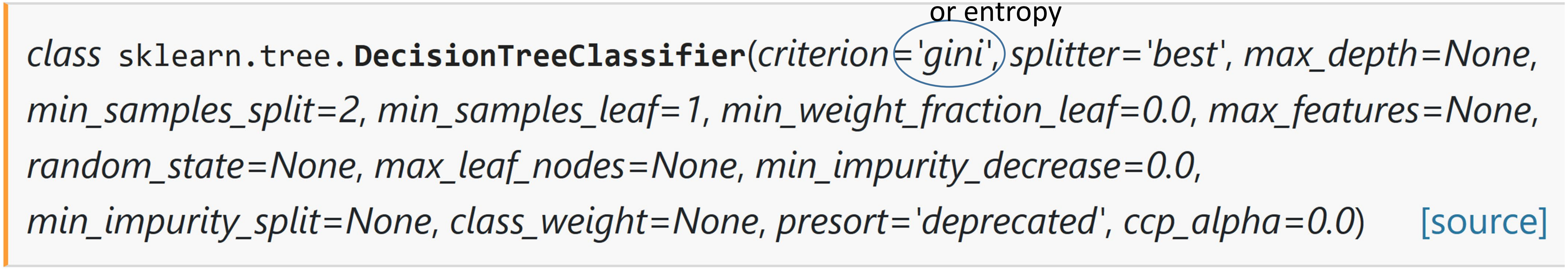
Construct/train a decision tree for classification

- After training, the feature space (input space) is partitioned into many regions, R_1, R_2, …, R_j,…

- A tree is a partition of the feature space

When will the algorithm stop growing the tree ?

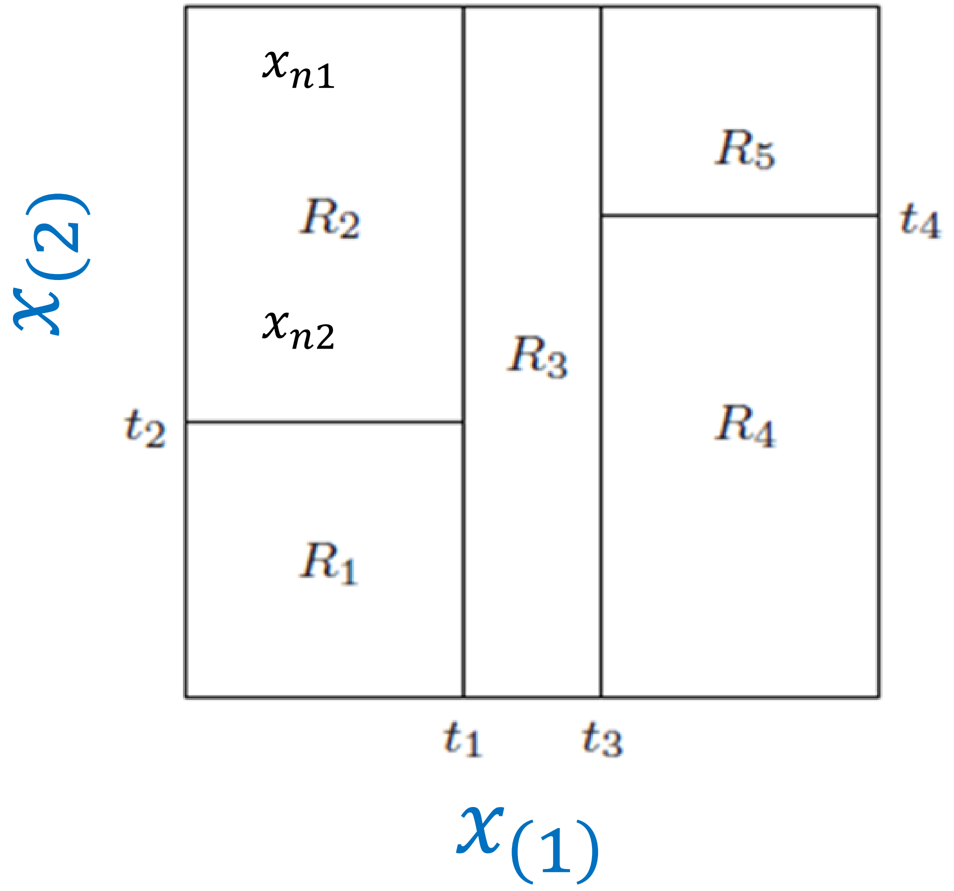
The algorithm can grow a deep tree such that every leaf node is a pure node (i.e., entropy=0). But a deep tree may not be good for your application.

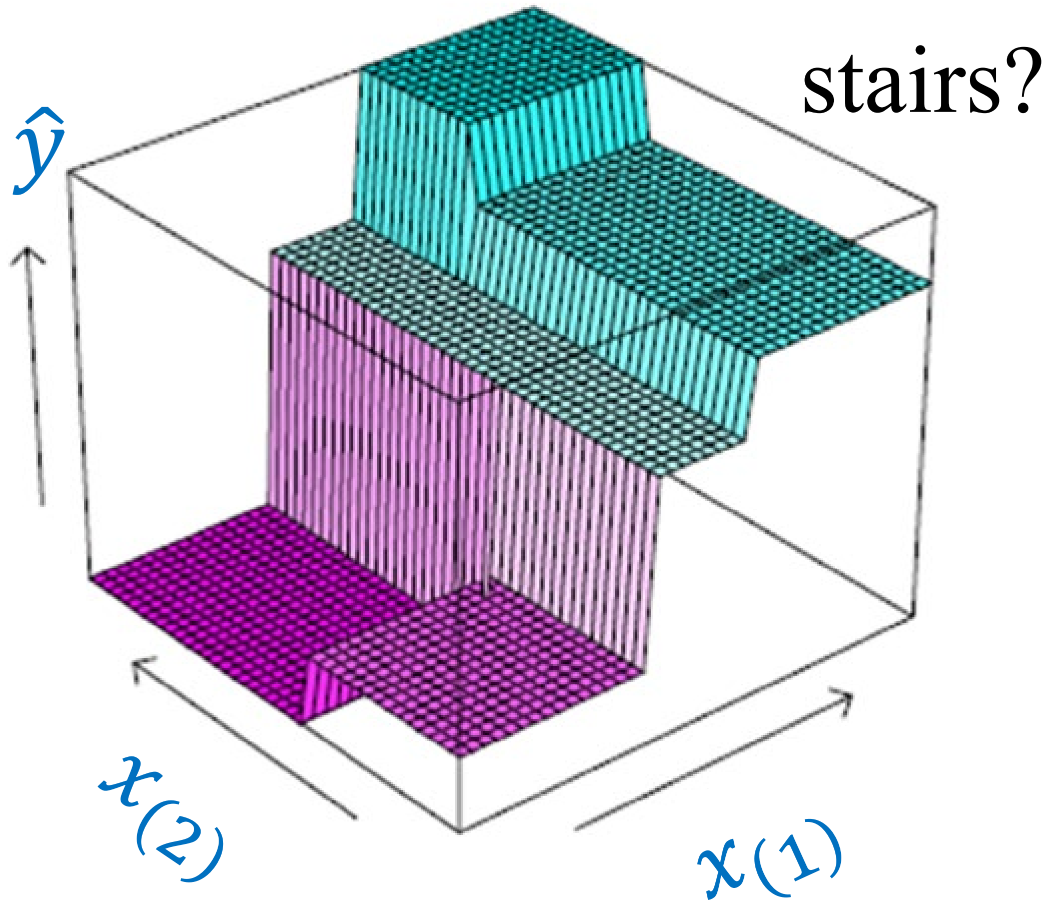


or entropy

```
class sklearn.tree.DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None,
random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0,
min_impurity_split=None, class_weight=None, presort='deprecated', ccp_alpha=0.0)     [source]
```

A decision tree for regression $\hat{y} = f(x_{(1)}, x_{(2)})$

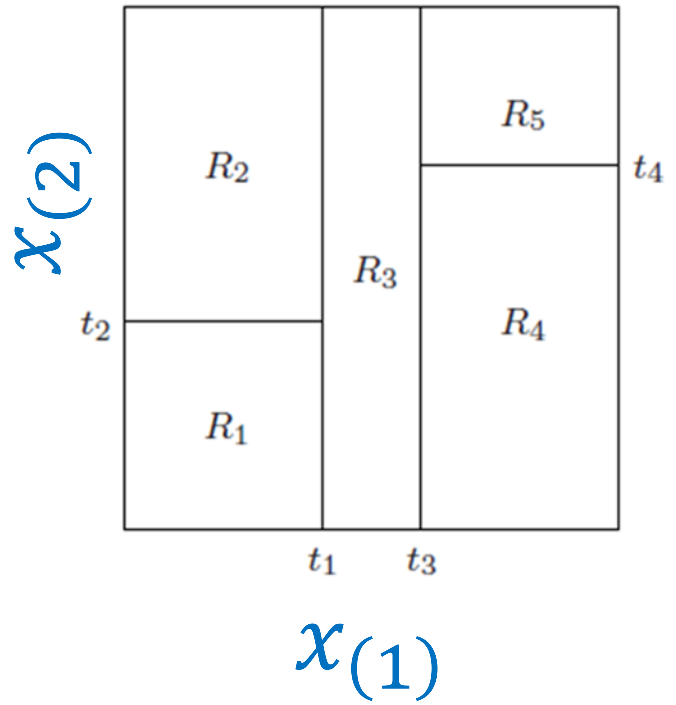


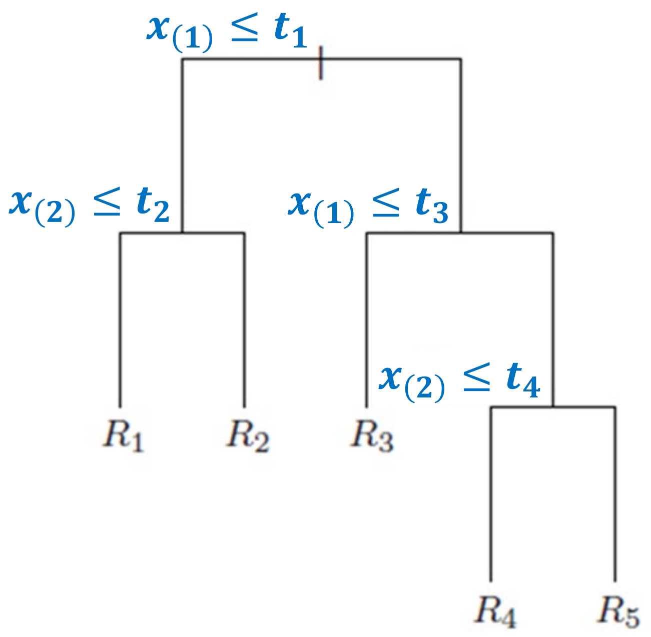


A tree is a partition of the input space

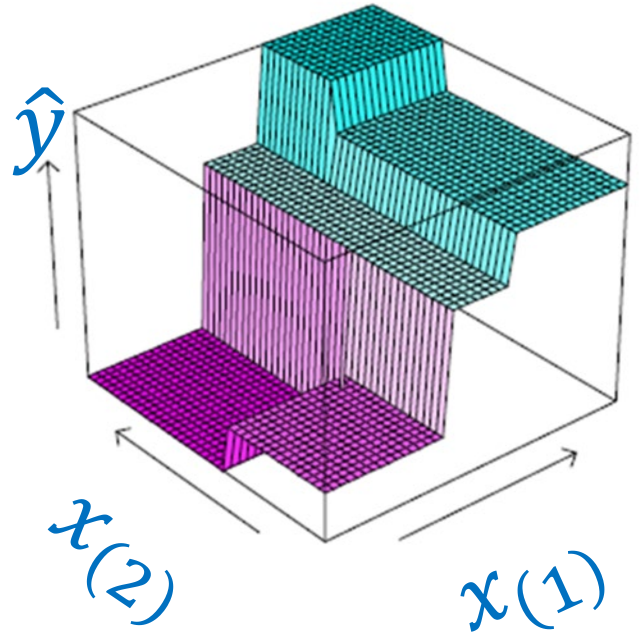
the predicted target values of the data points in the same region are the same.
e.g., two data points x_{n1} and x_{n2} in region R2, then $\hat{y}_{n1} = \hat{y}_{n2}$ from the tree

A decision tree for regression $\hat{y} = f(x_{(1)}, x_{(2)})$





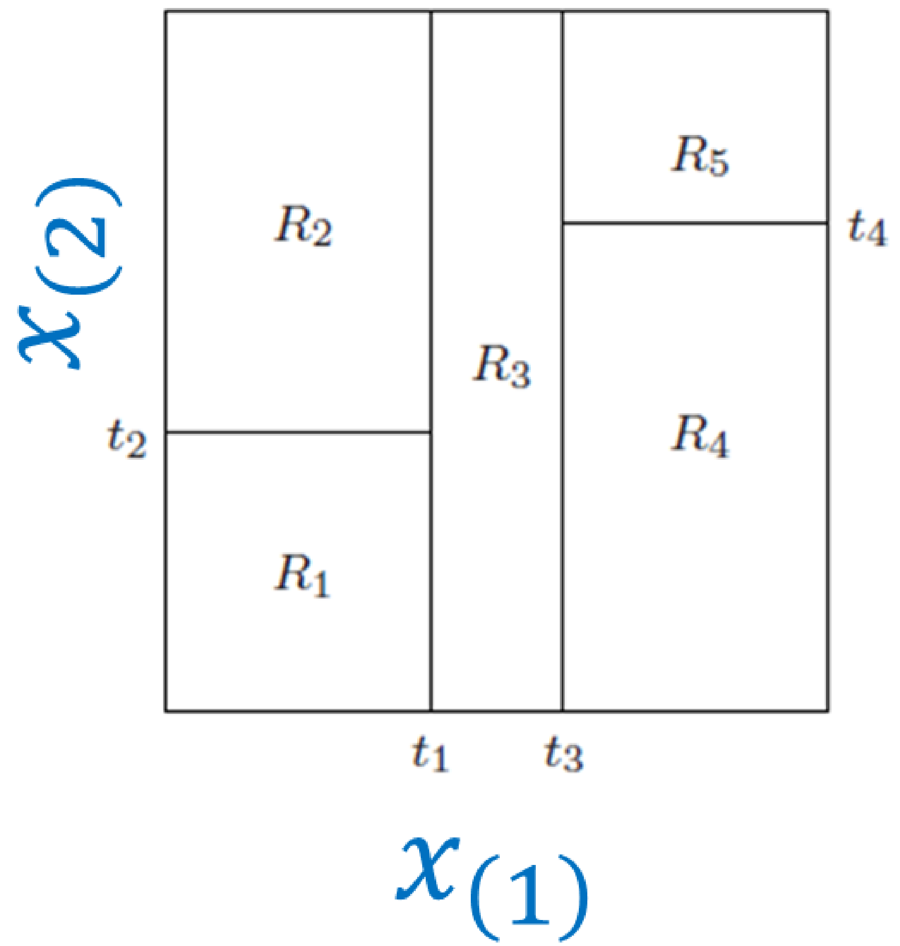
Stairs !



A tree is a partition of the input space

the predicted target values of the data points in the same region are the same. e.g. two data points x_a and x_b in region R2, then $\hat{y}_a = \hat{y}_b$ from the tree

Use a trained decision tree for regression $\hat{y} = f(x_{(1)}, x_{(2)})$



Here is the rule for regression:
If the input data sample $x = (x_{(1)}, x_{(2)})$
falls into the region R_j,
Then the predicted target value is c_j.

R_j is associated with c_j
c_j could be a vector

Construct/train a decision tree for regression (1)

- A training dataset $\{(x_n, y_n), n = 1, \ldots, N\}$ and $x_n \in \mathcal{R}^M$
- Let R_0 be the input space.
- We will partition the space into two regions R_1 and R_2 in the following steps:
- Step-1: **randomly** select **a subset of features**

$\{x_{(1)}$(e.g., width), $x_{(2)}$ (e.g., height),…$\}$

Construct/train a decision tree for regression (1)

- Step-2: for each selected feature $x_{(s)}$, find the best split t,

  such that this function is <u>minimized</u>:

  $$SSE = \sum_{x_n \in R_1} (y_n - c_1)^2 + \sum_{x_n \in R_2} (y_n - c_2)^2$$

  $$c_1 = average(y_n | x_n \in R_1) \; , \; c_2 = average(y_n | x_n \in R_2)$$

  $$R_1 = \{x_{(s)} \le t\}, \qquad R_2 = \{x_{(s)} > t\}$$

- Step-3: use the best feature and split (leading to the <u>minimum</u> of SSE) to divide the region R_0 into two regions R_1 and R_2,

- keep dividing the new regions until we get many regions/nodes

Construct/train a decision tree for regression (1)

Impurity of a node j:

$$Q(R_j) = MSE = \frac{1}{N_j} \sum_{x_n \in R_j} (y_n - c_j)^2$$

It is a pure node if the MSE is 0

We can also use MAE

Next, Let’s revisit the process of growing a tree

Construct/train a decision tree for regression (2)

- A training dataset $\{(x_n, y_n), n = 1, \ldots, N\}$ and $x_n \in \mathcal{R}^M$
- Let R_0 be the input space.
- We will partition the space into two regions R_1 and R_2 in the following steps:
- Step-1: **randomly** select **a subset of features**

$\{x_{(1)}$(e.g., width), $x_{(2)}$ (e.g., height),…$\}$

Construct/train a decision tree for regression (2)

- Step-2: for each selected feature $x_{(s)}$, find the best split $t_{(s)}$,

  such that this function is <u>maximized</u>:

$$E = Q(R_0) - \frac{N_1}{N_0} Q(R_1) - \frac{N_2}{N_0} Q(R_2)$$

$$c_1 = average(y_n | x_n \in R_1) \text{ , } c_2 = average(y_n | x_n \in R_2)$$

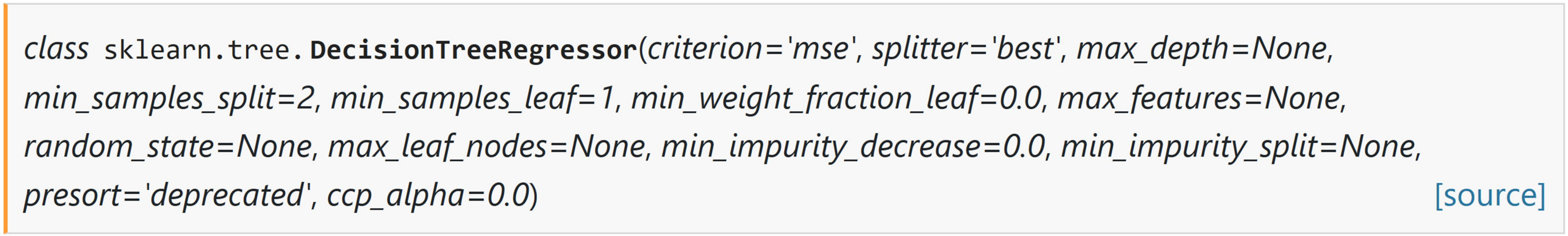
$$R_1 = \{x_{(s)} \leq t_{(s)}\}, \qquad R_2 = \{x_{(s)} > t_{(s)}\}$$

- Step-3: use the best feature and split (leading to the <u>maximum</u> of E) to divide the region R_0 into two regions R_1 and R_2,

- keep dividing the new regions until we get many regions/nodes

When will the algorithm stop growing the tree ?

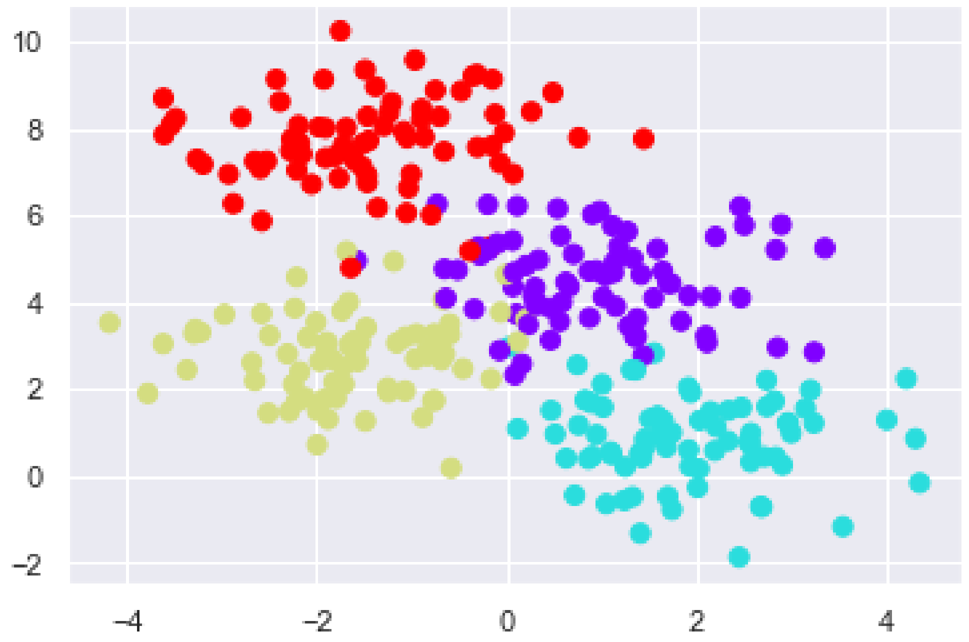
The algorithm can grow a deep tree such that every leaf node is a pure node (i.e., MSE=0). A deep tree may not be good for your application.



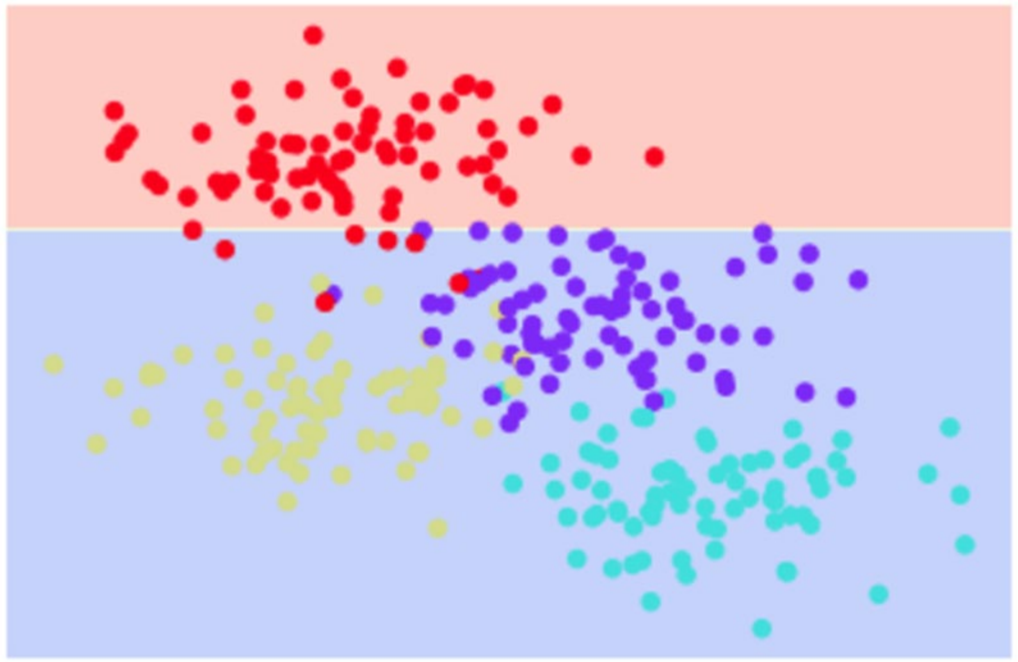
```
class sklearn.tree.DecisionTreeRegressor(criterion='mse', splitter='best', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None,
random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None,
presort='deprecated', ccp_alpha=0.0)
```

[source]

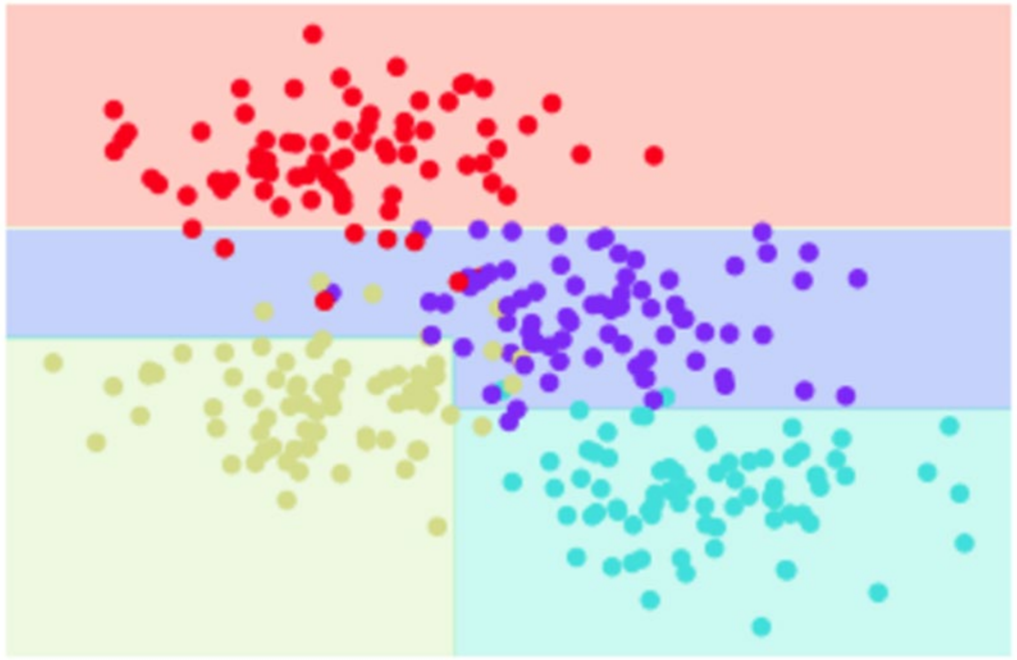
see the demo:
Decision_Trees_and_Random_Forests.ipynb



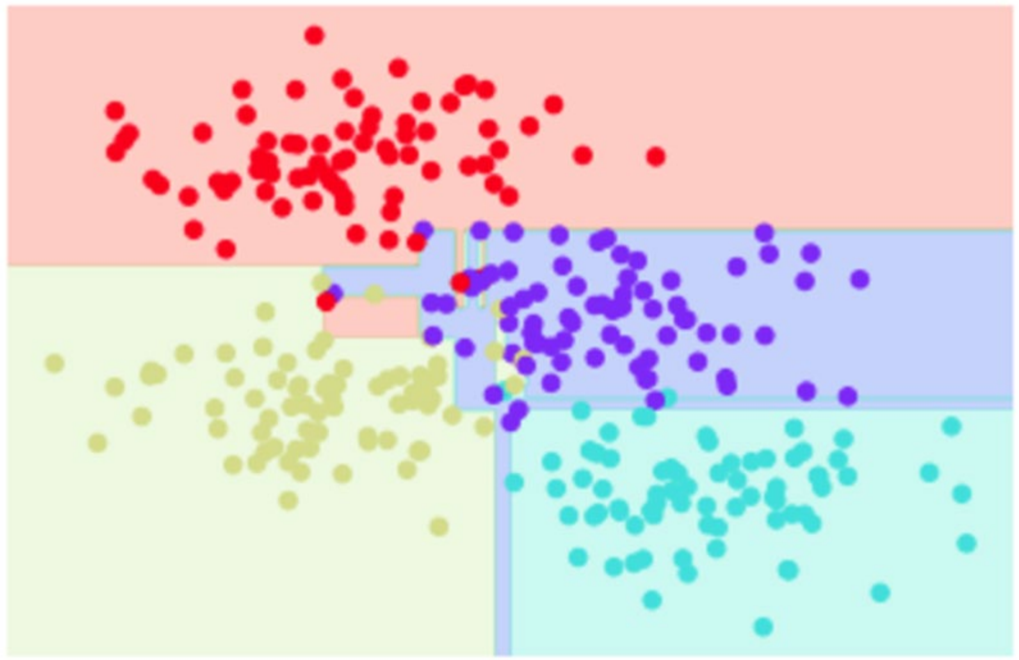
```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=1).fit(X, y)
```



```python
tree = DecisionTreeClassifier(max_depth=3).fit(X, y)
```

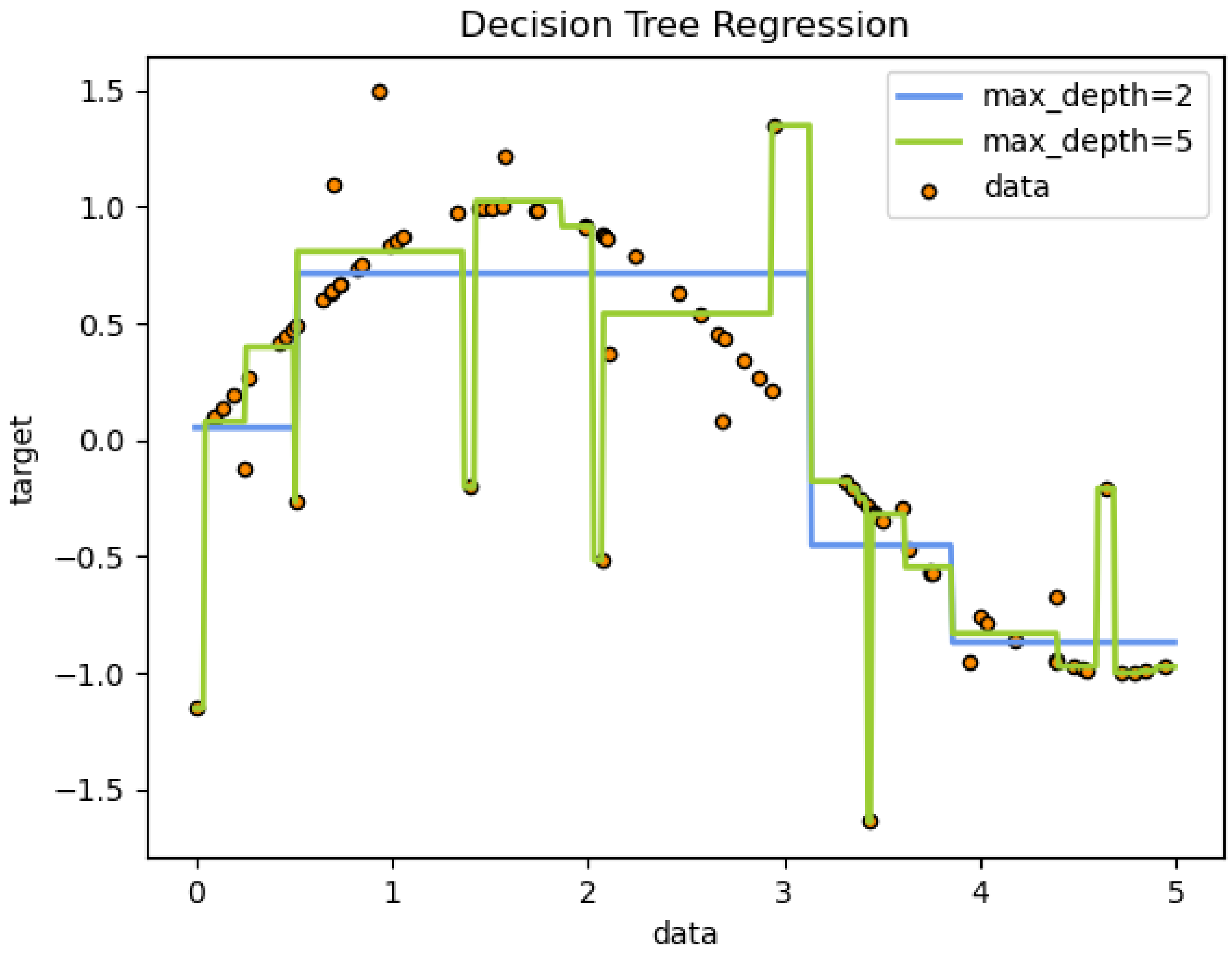


```python
tree = DecisionTreeClassifier(max_depth=100).fit(X, y)
```



A very deep tree: the decision boundary is “noisy”

decision tree regression example

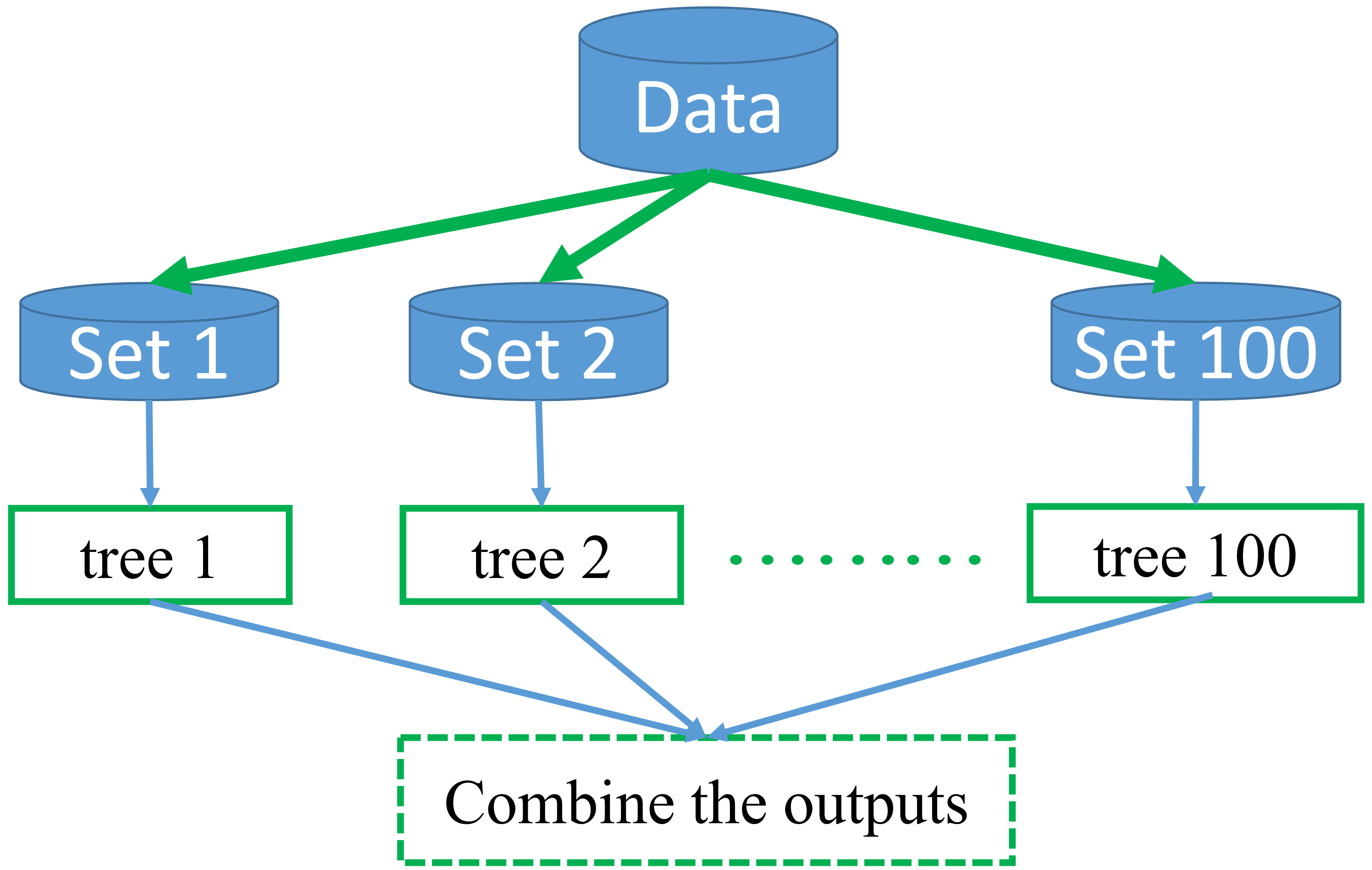


overfitting:
The tree with depth=5 fits to the outliers

Random Forest (a bag of trees)

- A random forest is a combination of many decision trees.
- Each tree is trained on a randomly selected subset of the training data
- The trees (i.e., the outputs) are weakly correlated (in theory)
  because of random selection of features when building a tree
- The output of a random forest is average for regression or majority vote for classification

Bagging



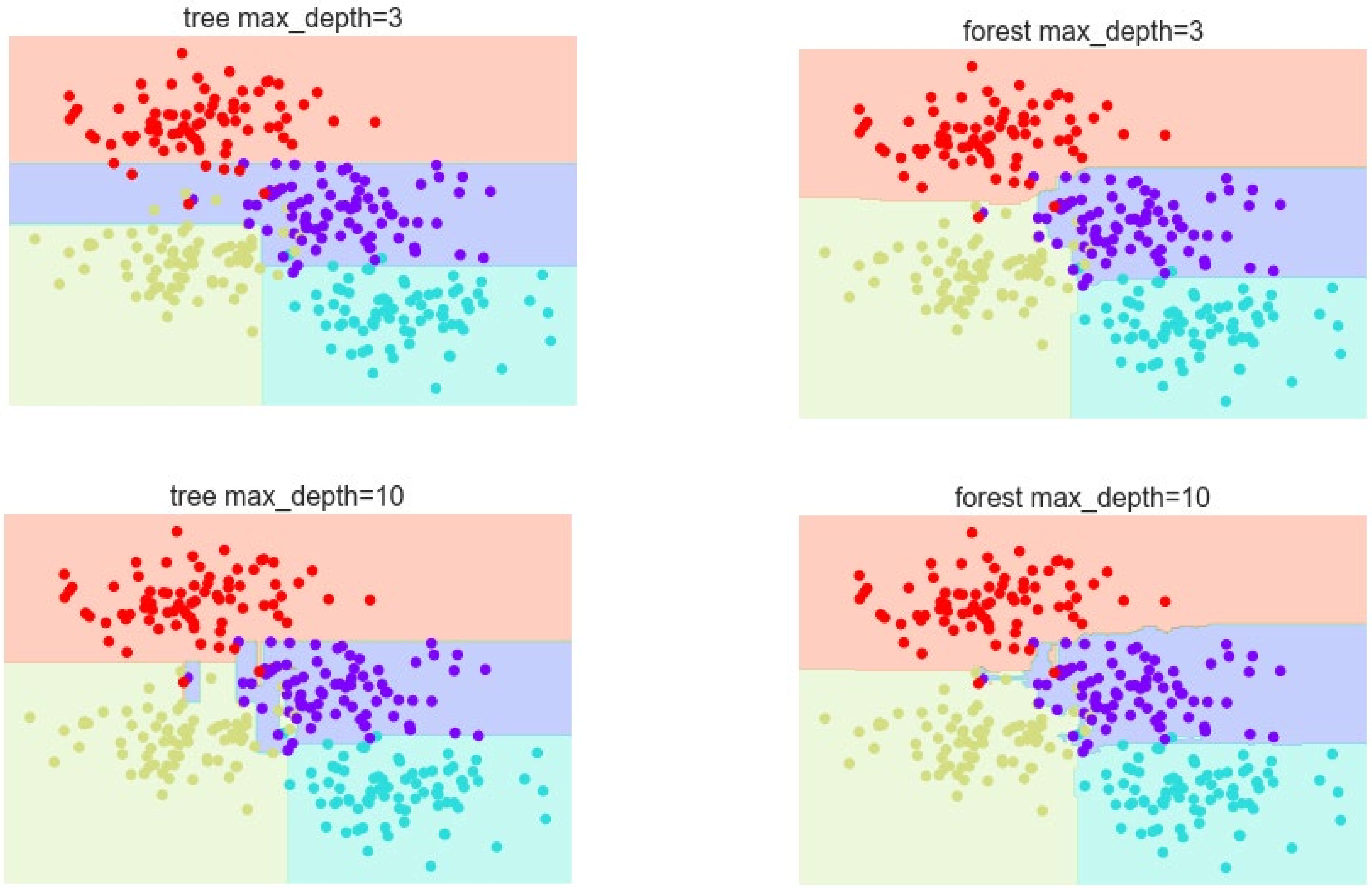
Combine the outputs:
Regression: average the outputs from the trees
Classification: take majority vote among the outputs

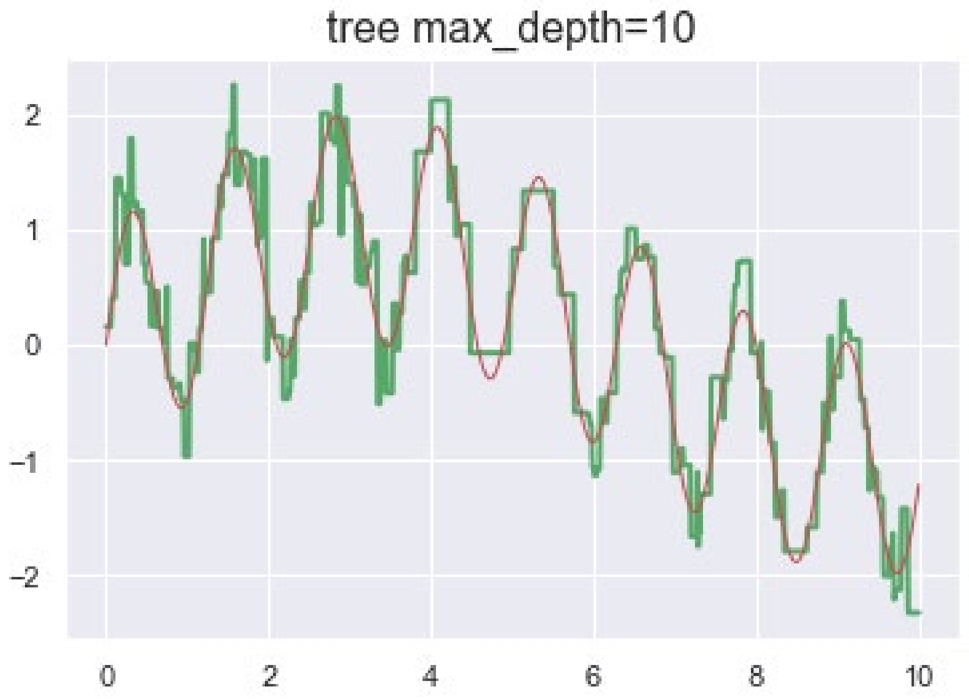
The Algorithm to build a Random Forest

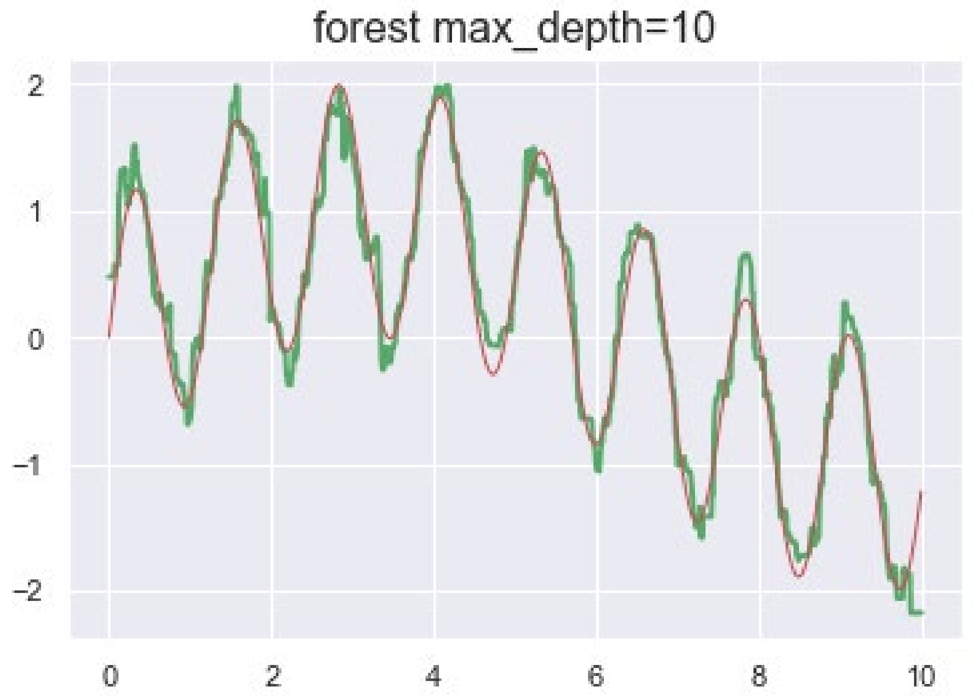
- Step-0: set the number of trees and hyper-parameters (max_depth, etc)
- Step-1: **randomly** select **a subset of** training data points (bootstrap samples)

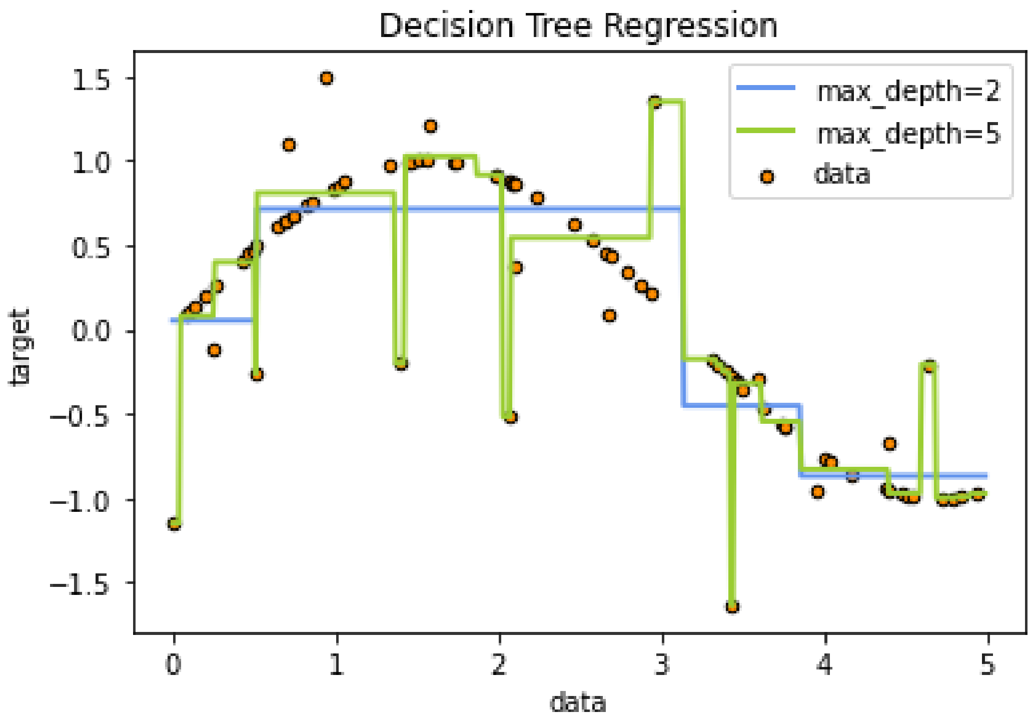
  build a tree using pre-defined hyper-parameters (max_depth, etc)
- Repeat Step-1 until we get all of the trees (e.g., 100).
- $T_i(x)$ denotes the output of a tree (predicted class label or target value of x)
- Regression: the predicted target value is $\hat{y} = \frac{1}{100}\sum_{i=1}^{100} T_i(x)$
- Classification: the predicted class label is $\hat{y} = majority\ vote\{T_1(x), \ldots T_{100}(x)\}$
- The output $\hat{y}$ of the random forest may be a vector (multi-class, multi-output)

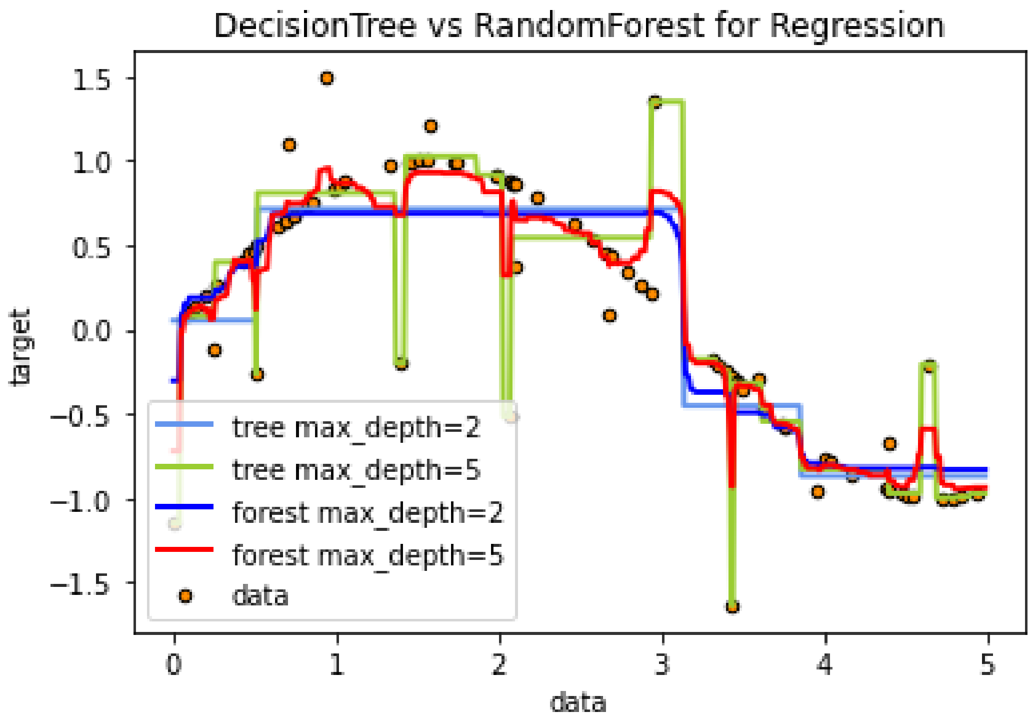
see the demo:
Decision_Trees_and_Random_Forests.ipynb


tree max_depth=3
forest max_depth=3
tree max_depth=10
forest max_depth=10


tree max_depth=10
2
1
0
−1
−2
0
2
4
6
8
10


forest max_depth=10
2
1
0
−1
−2
0
2
4
6
8
10





to handle large outliers, bagging is not enough, we need robust regression loss instead of MSE loss, or RANSAC.

Feature Importance in Random Forests

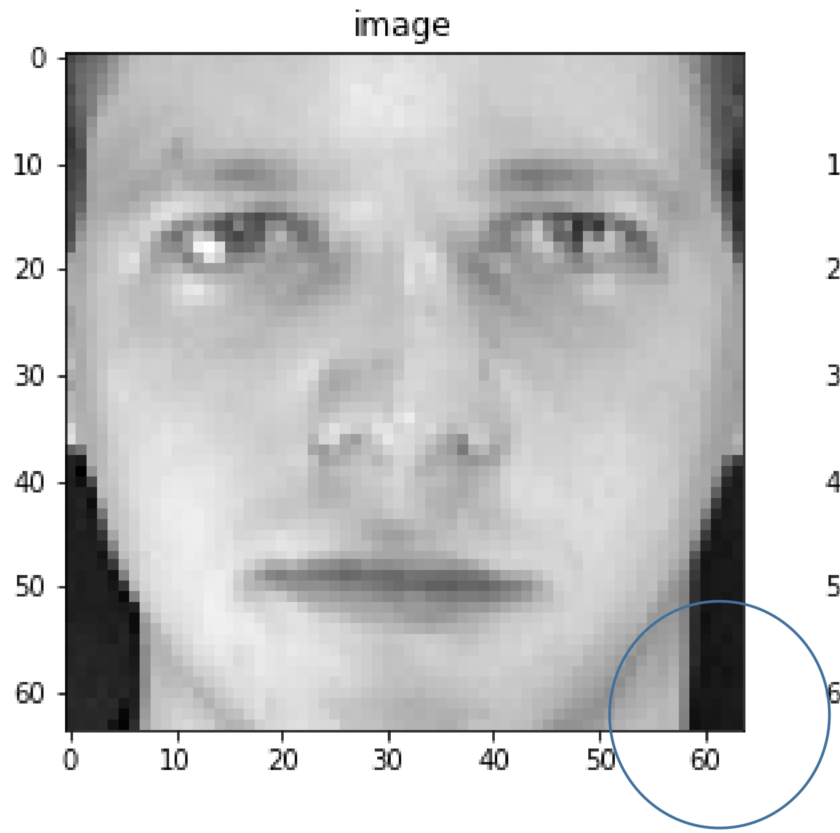
- Which feature is the most important for classification/regression?
- Measure feature importance:
  - vary a feature of the data points, and see how the outputs change
  - the average value of the function E used for node/region splitting

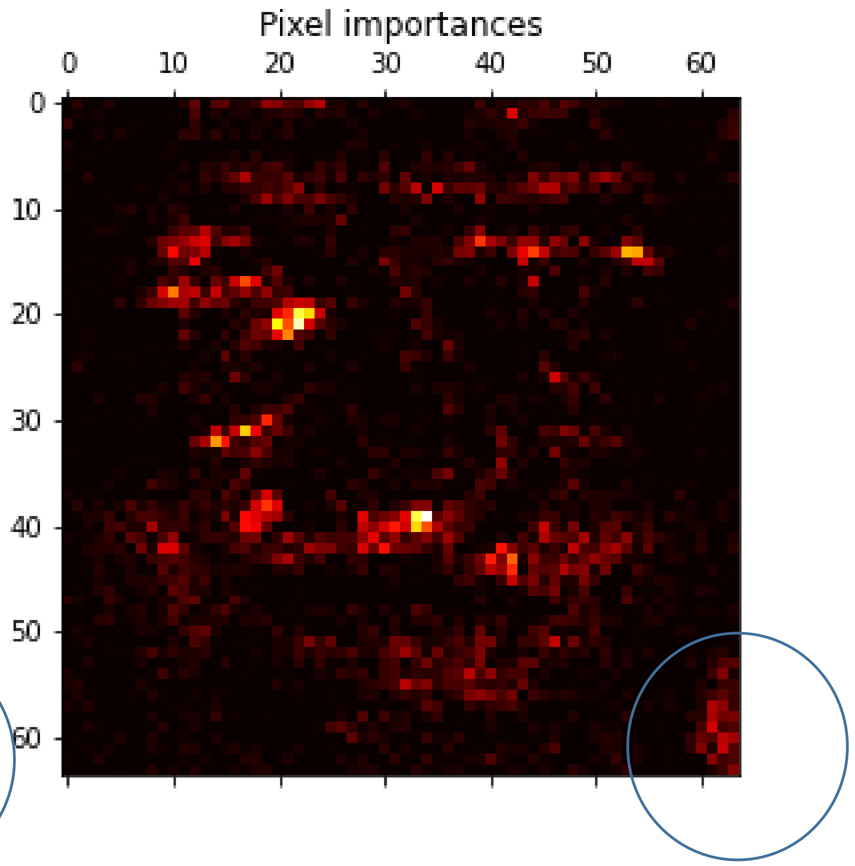
    E is used to find the best split for each feature

    take the average of E associated with a feature across the trees.

- Random forests are often used for feature selection see demo Feature_Importance_RF.ipynb

RandomForestClassifier identified important pixels/features for facial recognition





Feature importance shows the importance of a feature across the entire dataset, not just for one data sample

Feature importance: subjective or objective ?

- subjective: feature importance is only associated with a specific model

Feature-1 is important for classifier-A, but not for classifier-B

- objective: feature importance is the same for all the models

Feature-1 is important for classifier-A, classifier-B, etc

- Is this a good approach?

Use Random Forest to find the top 10 features and then fit a linear model on the data with 10 features, assuming that the 10 features are also important for the linear model?

Boosting

- Boosting
  - The weak models are trained one after another to minimize a loss
    train model-2 after model-1 is trained
    train model-3 after model-2 is trained
    ….
  - Combine the outputs (e.g., linearly)

- Adaboost
- XGBoost
- LightGBM
- CatBoost

XGBoost: Boosting Trees

- A training dataset $\{(x_n, y_n), n = 1, \ldots, N\}$
- We combine K trees into a forest (not random forest)

The predicted target value for x_n from the forest is

$$\hat{y}_n^{(K)} = \sum_{k=1}^{K} f_k(x_n)$$

where $f_k(x_n)$ is the output from the tree-k

The output is NOT the average !

https://arxiv.org/pdf/1603.02754.pdf

XGBoost

- A training dataset $\{(x_n, y_n), n = 1, \ldots, N\}$
- We have K trees and combine them into a forest (not random forest)

  The predicted target value from the forest is

  $\hat{y}_n^{(K)} = \sum_{k=1}^{K} f_k(x_n)$ where $f_k(x_n)$ is the output from the tree-k
- The loss function of the forest:

$$L_K = \sum_{n=1}^{N} l\left(y_n, \hat{y}_n^{(K)}\right) + \lambda \sum_{k=1}^{K} \Omega(f_k)$$

e.g., $l(y_n, \hat{y}_n^{(K)}) = ||y_n - \hat{y}_n^{(K)}||_2^2$ (L2 norm squared) for regression

$\Omega(f_k)$ could be the number of leaf nodes in the tree-k

λ is a "user-defined" parameter (hyper-parameter), let's set $\lambda = 1$

XGBoost : additive training

- The loss function:$L_K = \sum_{n=1}^{N} l\left(y_n, \hat{y}_n^{(K)}\right) + \sum_{k=1}^{K} \Omega(f_k)$
- Tree-1: build the first tree f_1, and the prediction is $\hat{y}_n^{(1)} = f_1(x_n)$

  using the loss $L_1 = \sum_{n=1}^{N} l\left(y_n, \hat{y}_n^{(1)}\right) + \Omega(f_1)$
- Tree-2: build the second tree f_2, and the prediction is

$$\hat{y}_n^{(2)} = f_1(x_n) + f_2(x_n) = \hat{y}_n^{(1)} + f_2(x_n)$$

  using the loss $L_2 = \sum_{n=1}^{N} l\left(y_n, \hat{y}_n^{(2)}\right) + \Omega(f_2)$

$$= \sum_{n=1}^{N} l\left(y_n, \hat{y}_n^{(1)} + f_2(x_n)\right) + \Omega(f_2)$$

in the loss L_2, $\hat{y}_1^{(n)}$ is a constant, we only optimize/build the second tree f_2

Input data: $\{(x_n, y_n), n = 1, \ldots, N\}$
Output K trees $f_1(x_n), \ldots, f_K(x_n)$
The prediction is $\hat{y}_n^{(K)} = \sum_{k=1}^{K} f_k(x_n)$

let's consider one data point (x, y)
Build the first tree $f_1(x)$, such that:

$$y \approx f_1(x)$$

the regression error is

$$|y - f_1(x)|$$

Build the second tree $f_2(x)$, such that:

$$y - f_1(x) \approx f_2(x)$$

the regression error is

$$|y - f_1(x) - f_2(x)| < |y - f_1(x)|$$

Build the third tree $f_3(x)$, such that:

$$y - (f_1(x) + f_2(x)) \approx f_3(x)$$

The regression error is

$$|y - f_1(x) - f_2(x) - f_3(x)|$$
$$< |y - f_1(x) - f_2(x)|$$

Combine the three trees for regression

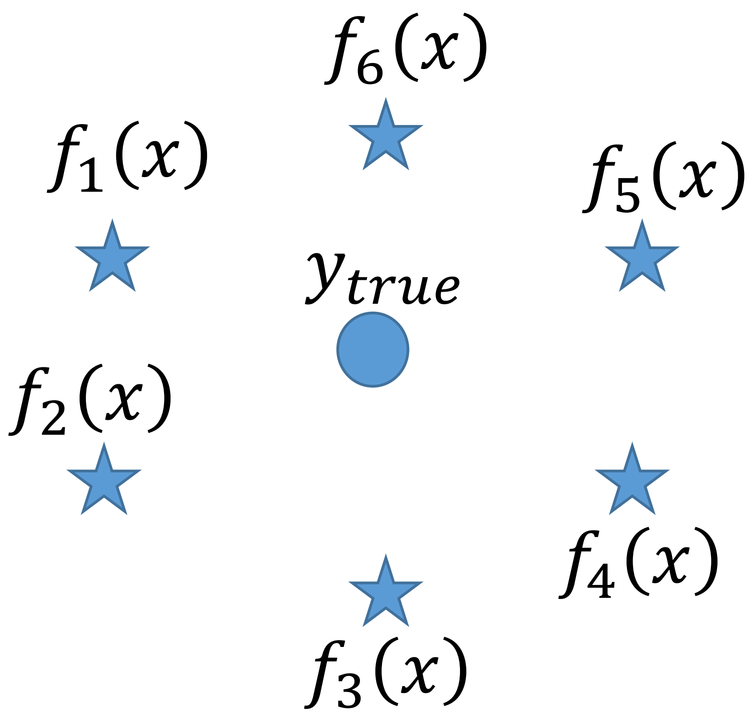
$$y \approx \hat{y} = f_1(x) + f_2(x) + f_3(x)$$

Add trees one-by-one to reduce the regression error gradually

Boosting Trees

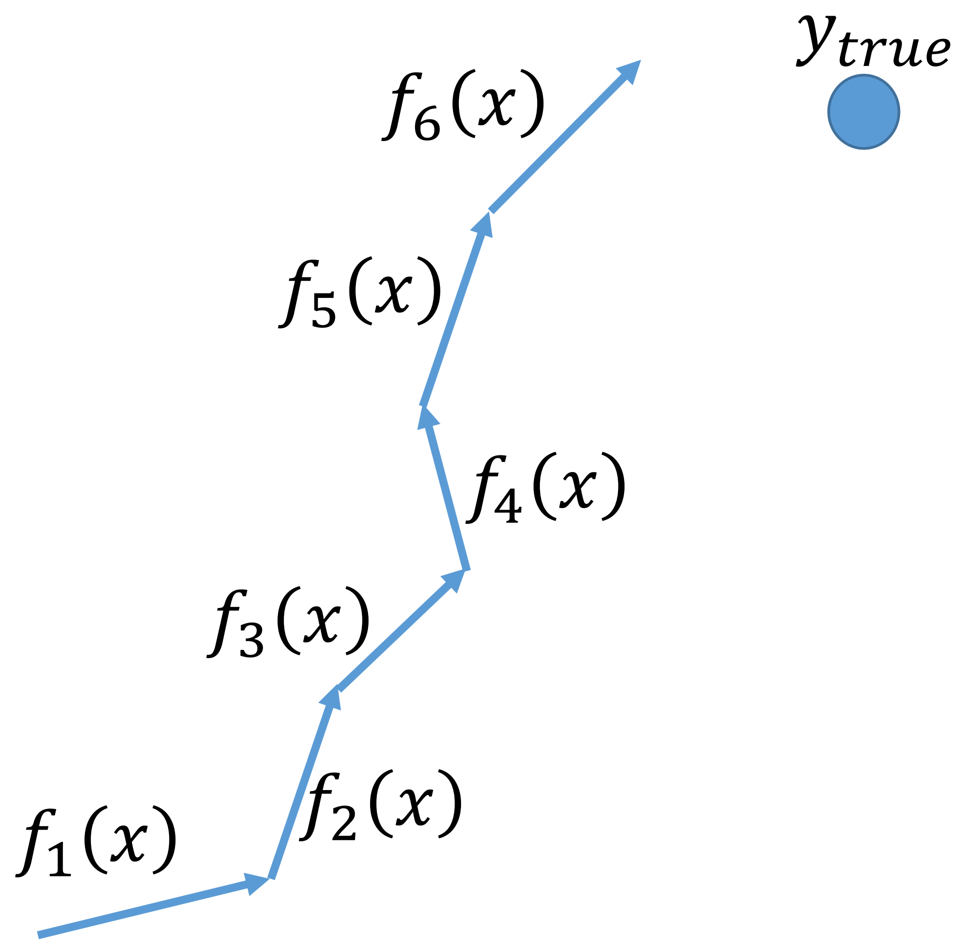
- XGBoost : https://xgboost.readthedocs.io
- LightGBM: https://lightgbm.readthedocs.io
- CatBoost : https://github.com/catboost/catboost
- The python packages can be downloaded from anaconda
- Boosting trees are very useful to handle tabular data (tables)

Bagging (RF) vs Boosting (XGBoost)



$$y_{pred} = \frac{1}{6}\sum_{i=1}^{6} f_i(x)$$

the goal is to reduce variance



$$y_{pred} = \sum_{i=1}^{6} f_i(x)$$

the goal is to reduce bias