Lecture 3.3
Supervised learning:
Decision Trees & Ensemble Learning

Universitat de les Illes Balears

Departament de Ciències Matemàtiques i Informàtica

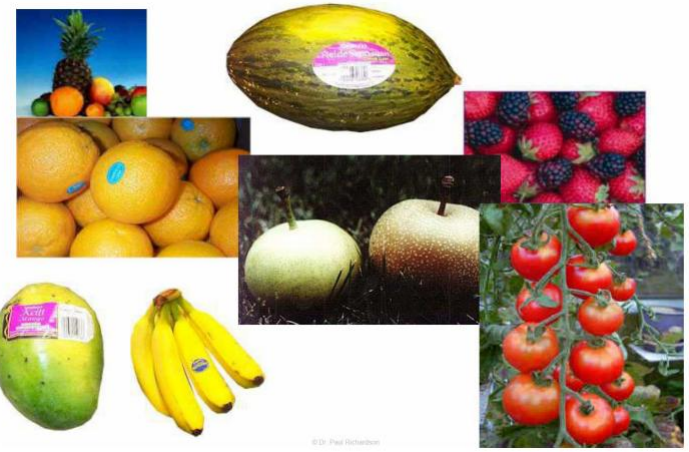
11752 Aprendizaje Automático
11752 Machine Learning
Máster Universitario
en Sistemas Inteligentes

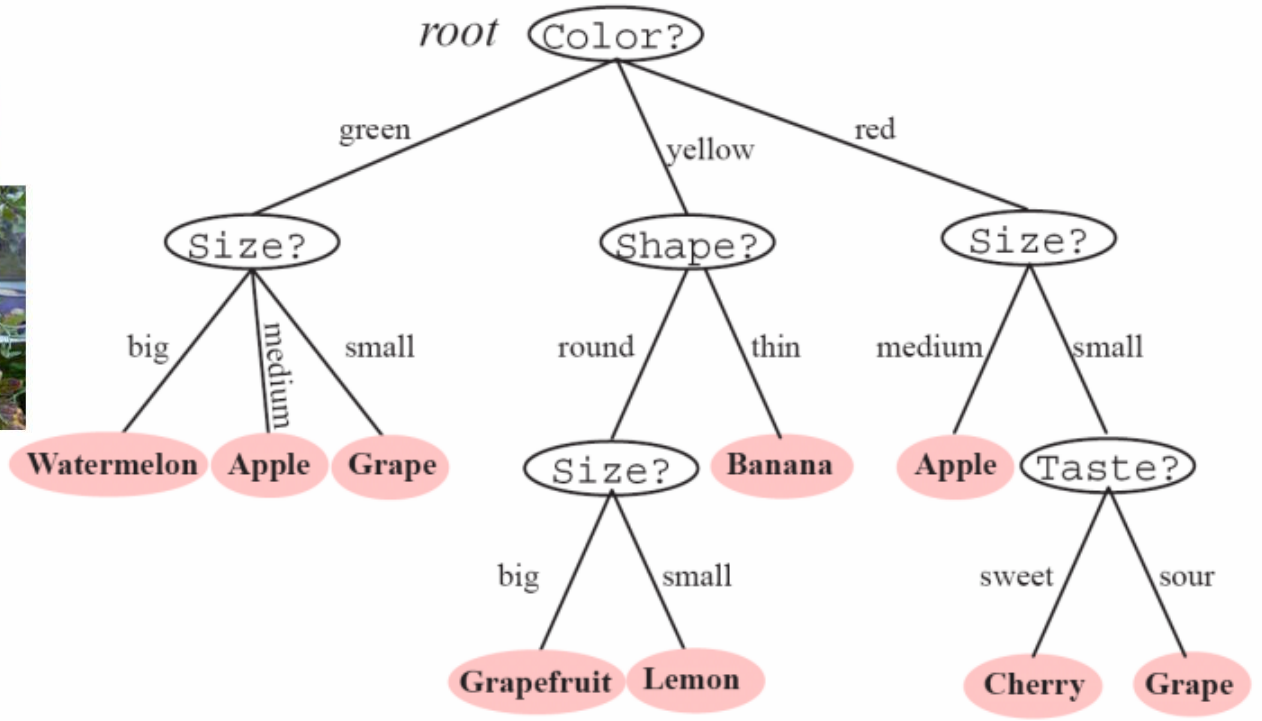
Alberto ORTIZ RODRÍGUEZ

Contents

Definition

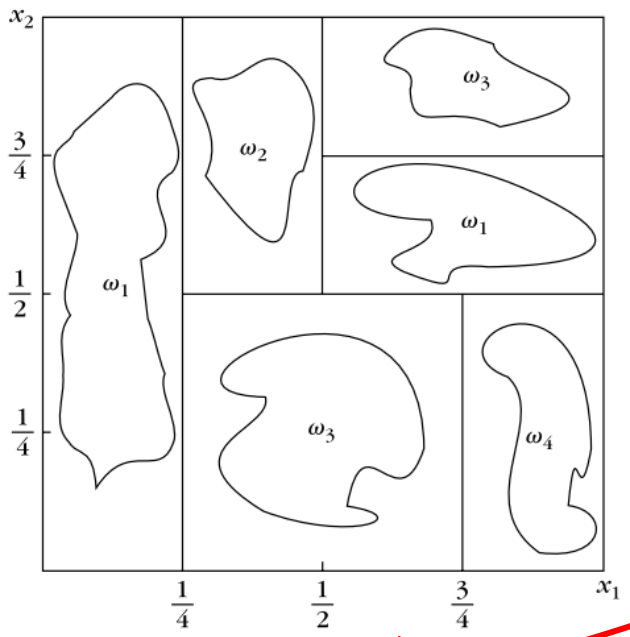
- The classification process consists in the **sequential application of a set of questions**, in which the next question depends on the result of the previous ones
  - The set of questions is organized over a **directed tree** as a multistage decision system.
    - Naturally handle all kind of data: metric and non-metric, any number of dimensions
    - The sequence of questions split the feature space into non-overlapping regions
    - Not all features need to be evaluated to make a decision

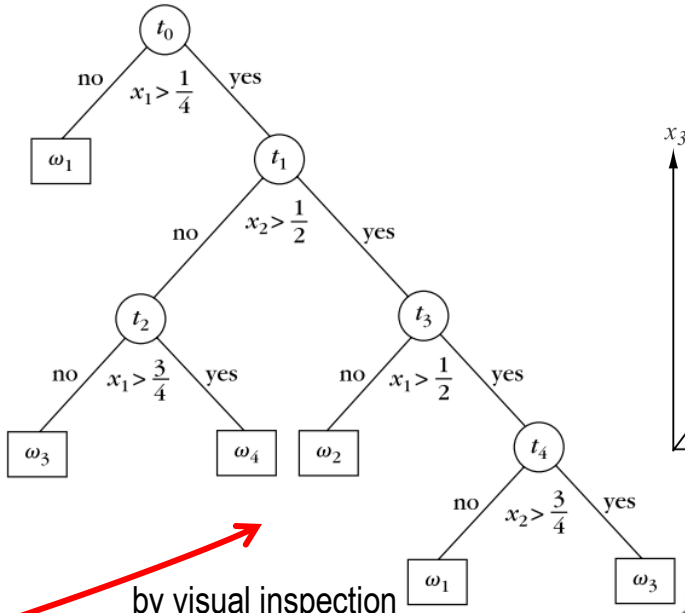


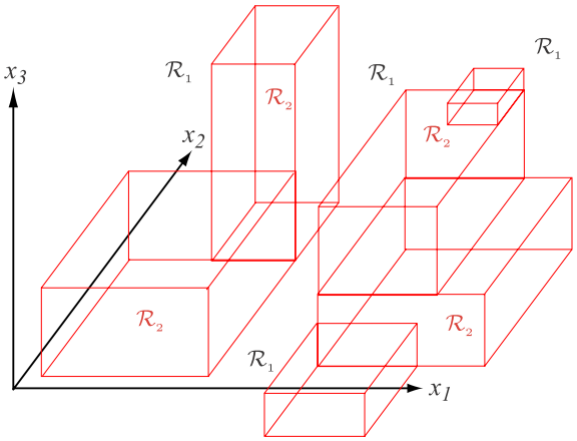


Definition

- Most popular among decision trees: Ordinary **Binary** Classification Tree (OBCT)

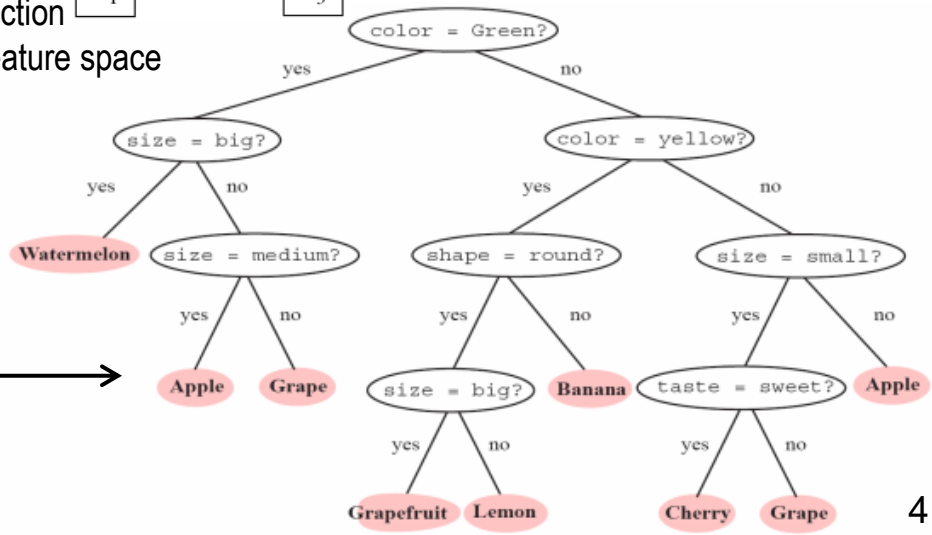






by visual inspection
of dataset in feature space

- In general, split the space into **hyper-rectangles**, with sides parallel to the axes
- Decision trees can always be converted into OBCT (more nodes)
- We will consider OBCT for simplicity



Set of Questions

- For the **OBCT**s, questions are of the form ***is feature*** $\boldsymbol{x_k \leq \alpha_k}$ *?*
  - Once α_k has been selected, a split of the training set X is defined
  - How to select x_k & α_k?
    - A priori, it seems there are infinite possibilities regarding the α_k
    - Actually, only the $n_k \leq N$ different values taken by feature k within X have to be considered

      α_k would be taken halfway between consecutive distinct values of x_k

      e.g. if $X = \{(0.4, 0.2), (0.56, 0.1), (1.0, -0.5), (3.2, 2.0), (-1.5, -1.5)\}$ the **candidate** questions would be as follows:

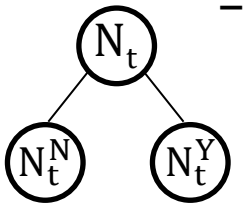
x_1	-1.5	0.4	0.56	1.0	3.2
x_2	-1.5	-0.5	0.10	0.2	2.0

x_1	$\leq \frac{-1.5+0.4}{2} = -0.55$	$\leq \frac{0.4+0.56}{2} = 0.48$	$\leq \frac{0.56+1}{2} = 0.78$	$\leq \frac{1+3.2}{2} = 2.1$
x_2	$\leq \frac{-1.5-0.5}{2} = -1$	$\leq \frac{-0.5+0.1}{2} = -0.25$	$\leq \frac{0.1+0.2}{2} = 0.15$	$\leq \frac{0.2+2.0}{2} = 1.1$

- Now, we have to choose which question from the candidates $\Rightarrow$ which x_k & α_k ?
  - **Choose the best split** according to a **splitting criterion**

Splitting Criterion

- Quantify **node impurity** and split so that the overall impurity $I(N_t^Y)$ and $I(N_t^N)$ of the descendant nodes N_t^Y and N_t^N is optimally decreased with respect to the ancestor node's impurity $I(N_t)$



  - Impurity for node N_t comprising samples $X_t \subseteq X$: **entropy** H (ID3), **Gini index** G (CART)

$$I(N_t) = H(X_t) = -\sum_{i=1}^{M} P(\omega_i|N_t)\, log_2\, P(\omega_i|N_t)$$

$$I(N_t) = G(X_t) = \sum_{i=1}^{M} P(\omega_i|N_t)[1 - P(\omega_i|N_t)] = 1 - \sum_{i=1}^{M} P(\omega_i|N_t)^2$$

    where $P(\omega_i \mid N_t) = P(x \in \omega_i | \, x \in X_t)$

    - In practice, $P(\omega_i \mid N_t) = \frac{n_t^i}{n_t}$, where n_t^i is the number of elements of X_t that belongs to ω_i

  - Impurity is maximum if $P(\omega_i|N_t) = 1/M \; \forall i$, and minimum if $\exists i \mid P(\omega_i|N_t) = 1$ for a certain class ω_i $(H\colon\ 0 \cdot \log_2 0 \to 0)$
  - After a split, the decrease in **node impurity** is defined as:

$$\Delta I(N_t) = I(N_t) - \frac{n_t^Y}{n_t} I(N_t^Y) - \frac{n_t^N}{n_t} I(N_t^N)$$

  - The goal is thus to adopt, **from the set of candidate questions**, the one that performs the split leading to the **highest decrease of impurity**:

$$(x_k^*, \alpha_k^*) = \text{argmax}\, \Delta I(N_{x_k,\alpha_k})$$

Examples

- **Example 1**. In a tree classification task, the set X_t, associated with node N_t, contains $n_t = 10$ samples. Four of these belong to class ω_1, four to class ω_2, and two to class ω_3. The node splitting results into two new subsets X_t^Y, with three samples from ω_1, and one from ω_2, and X_t^N, with one sample from ω_1, three from ω_2, and two from ω_3. Determine the **decrease in impurity** after splitting using the entropy.

$$\Delta I(N_t) = I(N_t) - \frac{n_t^Y}{n_t} I(N_t^Y) - \frac{n_t^N}{n_t} I(N_t^N)$$

$$I(N_t) = -\frac{4}{10}\log_2\frac{4}{10} - \frac{4}{10}\log_2\frac{4}{10} - \frac{2}{10}\log_2\frac{2}{10} = 1.521$$

$$I(N_t^Y) = -\frac{3}{4}\log_2\frac{3}{4} - \frac{1}{4}\log_2\frac{1}{4} = 0.815$$

$$I(N_t^N) = -\frac{1}{6}\log_2\frac{1}{6} - \frac{3}{6}\log_2\frac{3}{6} - \frac{2}{6}\log_2\frac{2}{6} = 1.472$$

$$\Delta I(N_t) = 1.521 - \frac{4}{10}(0.815) - \frac{6}{10}(1.472) = 0.315$$

Classification Rules

- **Stop-splitting rule**:
  When one decides to stop splitting a node N_t and declares it as a leaf of the tree?
  1. Stop if node N_t is pure, i.e. all the samples belong to a single class
  2. Given a threshold τ_I on the impurity decrease for a node, stop splitting if $\Delta I(N_t) < \tau_I$
  3. Stop if $n_t^Y < \tau_n$ or $n_t^N < \tau_n$
  4. Stop if, after the splitting, the depth of the tree is above a threshold τ_D

  Alternative approach: make the tree grow to a larger size and then use a **pruning criterion**
  which leaf can be suppressed with the least impact on the classification error?

- **Class assignment rule**:
  Once a node is declared to be a leaf, which class label it must be given?
  – Typically: $j = \arg\max_i \{P(\omega_i|N_t)\} = \arg\max_i \left\{\frac{n_t^i}{n_t}\right\}$

Algorithm (for building OBCTs)

NL $\leftarrow X$ # *node list*

while non-empty(NL) **do**

$N_t \leftarrow$ next-element(NL) # *and remove it from the list*

if stop-splitting(N_t) **then**

Declare N_t as a leaf node and designate it with class label ω_j:

$$j = \arg\max_i \{P(\omega_i|N_t)\} = \arg\max_i \left\{\frac{n_t^i}{n_t}\right\}$$

else

for every feature $x_k, k = 1, 2, \ldots, L$

for every different value $\alpha_{ki}, i = 1, 2, \ldots,$ taken by x_k in X_t

Generate N_t^Y and N_t^N according to the question

is $x_k \leq \alpha_{ki}$?

Calculate $\Delta I(N_t) = I(N_t) - \frac{n_t^Y}{n_t} I(N_t^Y) - \frac{n_t^N}{n_t} I(N_t^N)$

end

For feature x_k, choose α_{ki^*} leading to the maximum $\Delta I(N_t)$

end

Choose x_{k^*} leading to the maximum $\Delta I(N_t)$ for $\alpha_{k^*i^*}$

Insert the two associated subnodes N_t^Y and N_t^N in NL

end

end

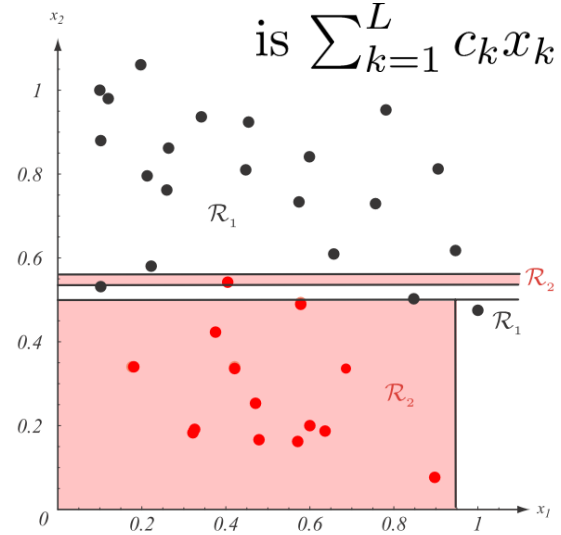
Final remarks

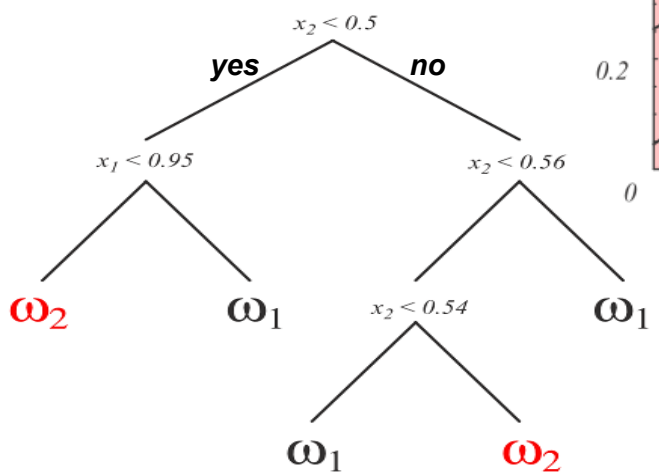
- Algorithms for building decision trees:
  - **ID3** (Iterative Dichotomiser 3) was developed in 1986 by Ross Quinlan. The algorithm creates a multiway tree, finding for each node (i.e. in a greedy manner) the categorical feature that will yield the largest information gain for categorical targets. Trees are grown to their maximum size and then a pruning step is usually applied to improve the ability of the tree to generalize to unseen data.
    - **C4.5** is the successor of ID3. The restriction that features must be categorical was removed by dynamically defining a discrete attribute (based on numerical variables) that partitions the continuous attribute value into a discrete set of intervals. C4.5 converts the trained trees (i.e. the output of the ID3 algorithm) into sets of **if-then rules**. The accuracy of each rule is then evaluated to determine the order in which they should be applied. Pruning is done by removing a rule's precondition if the accuracy of the rule improves without it.
    - **C5.0** is Quinlan's latest version release under a proprietary license. It uses less memory and builds smaller rule-sets than C4.5 while being more accurate.
  - **CART** (Classification and Regression Trees) is very similar to C4.5, but it differs in that it supports numerical target variables (regression) and does not compute rule sets. CART constructs binary trees using the feature and threshold that yield the largest information gain at each node.
- Scikit-learn implements an **optimized version of the CART algorithm**, although the scikit-learn implementation does not support categorical variables for now.

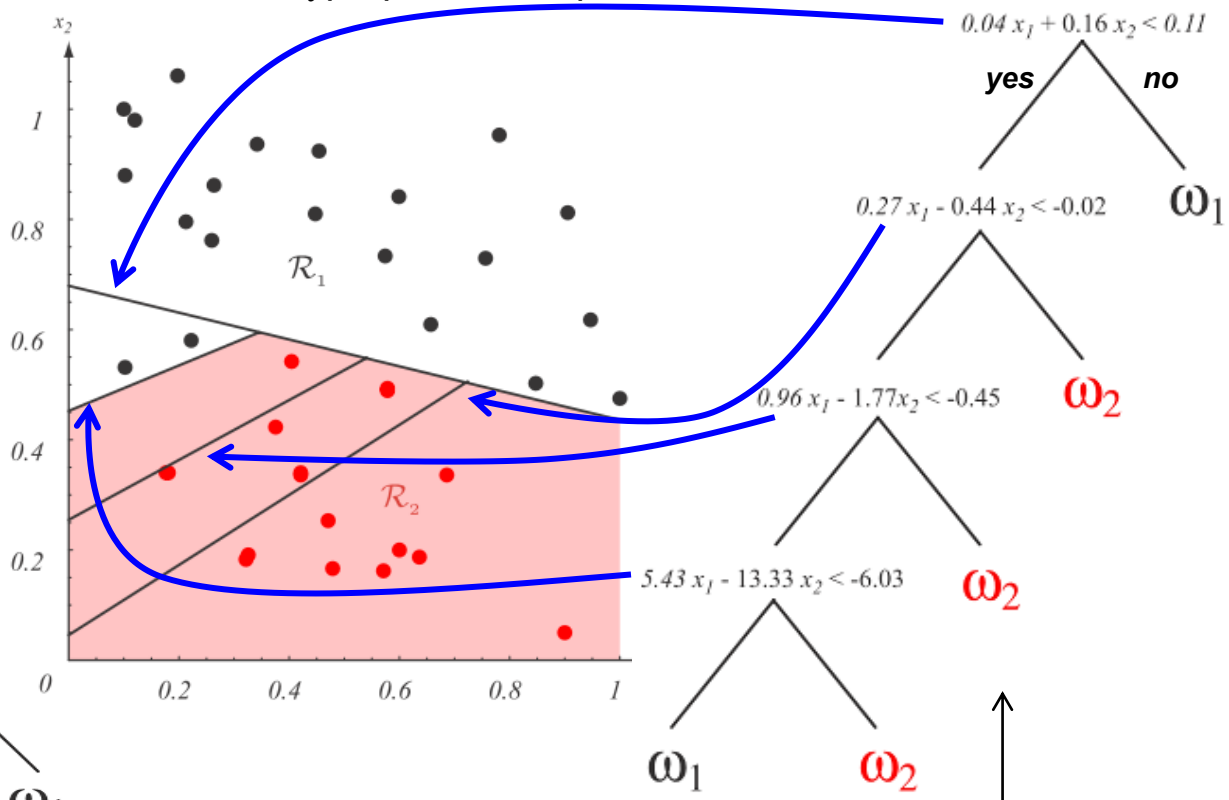
Final remarks

- More general partitions of the feature space are possible via questions of the type:

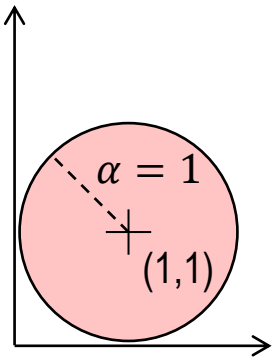
  is $\sum_{k=1}^{L} c_k x_k < \alpha_k$? i.e. via hyperplanes not parallel to the axes







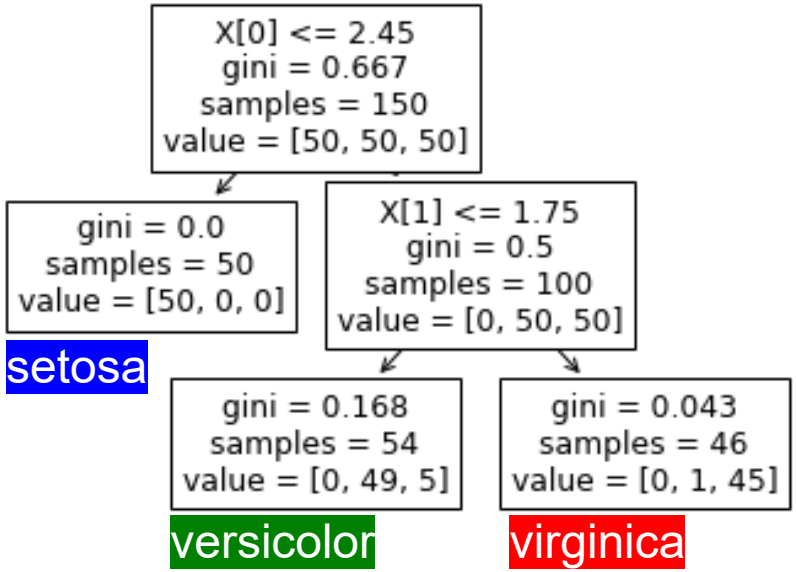
- In general, questions such as $\Phi(x) < \alpha$, e.g. $\Phi(x) = x_1^2 + x_2^2 - 2x_1 - 2x_2 + 2$
  - This can lead to a better partition of the feature space but training gets more difficult !!

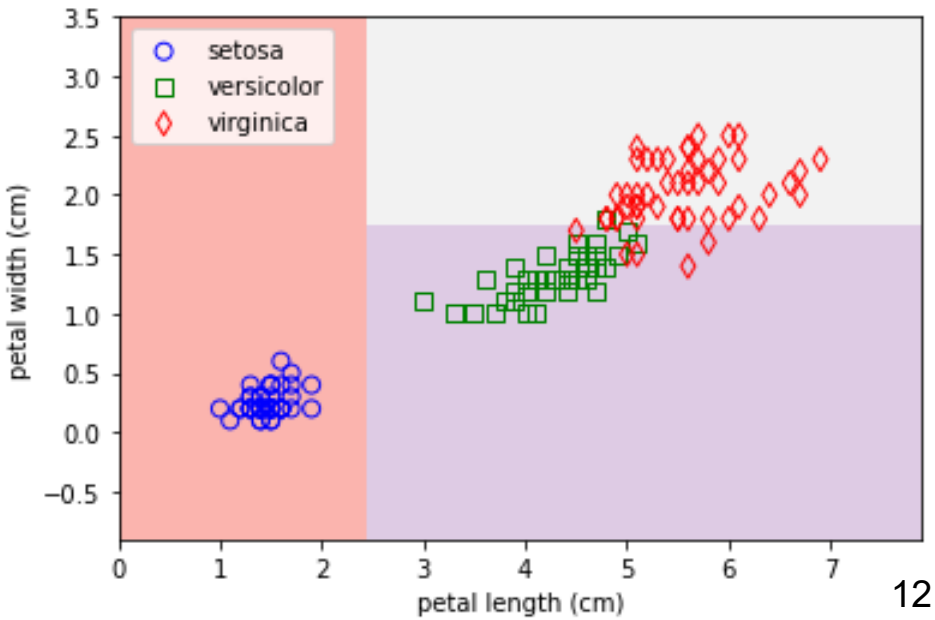


Examples

- **Example 2**

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target
tree = DecisionTreeClassifier(
    criterion="gini",
    max_depth=2,
    min_samples_leaf=5,
    min_impurity_decrease=0.1,
    random_state=100)
tree.fit(X, y)

plot_tree(tree)
plt.show()
```



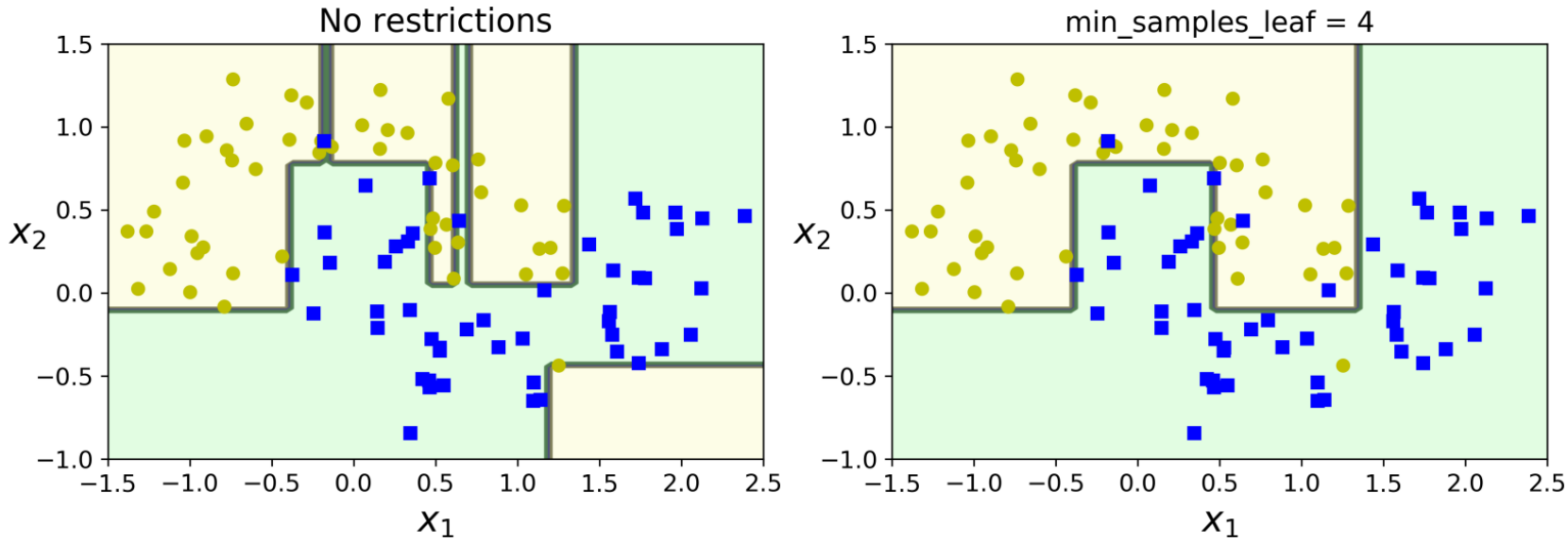


regularization hyperparameters

- avoid overfitting
- only some hyperparameters are shown, there are others

Examples

- **Example 3.** Effect of regularization parameters



  - **Left**: overfitting, **Right**: will probably generalize better
  - **Left**: The splitting process continues until all nodes are pure if not stopped before
  - **Right**: Increasing *min_** hyperparameters and/or reducing *max_** hyperparameters regularize the model, and avoids overfitting

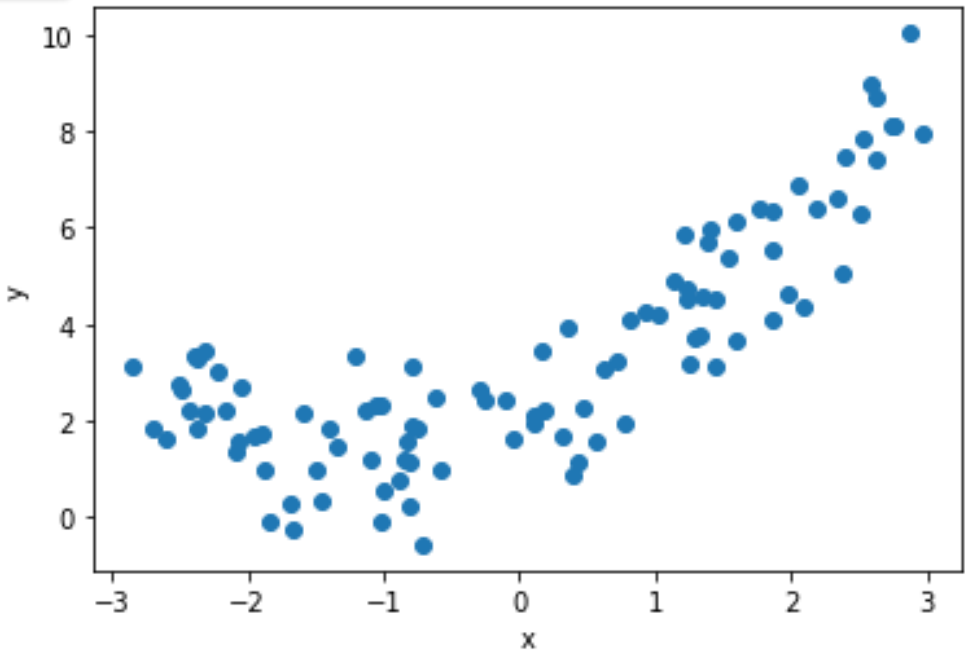
Contents

- Decision trees
- Regression trees
- Ensemble learning
- Random forests

Regression trees

- Decision trees are also capable of performing regression tasks.
  By way of example, we can build a **regression tree** using scikit-learn:

```
from sklearn.tree import DecisionTreeRegressor

# 1D quadratic synthetic dataset
m = 100
X = 6 * np.random.rand(m,1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m,1)

tree = DecisionTreeRegressor(max_depth=2)
tree.fit(X, y)
```

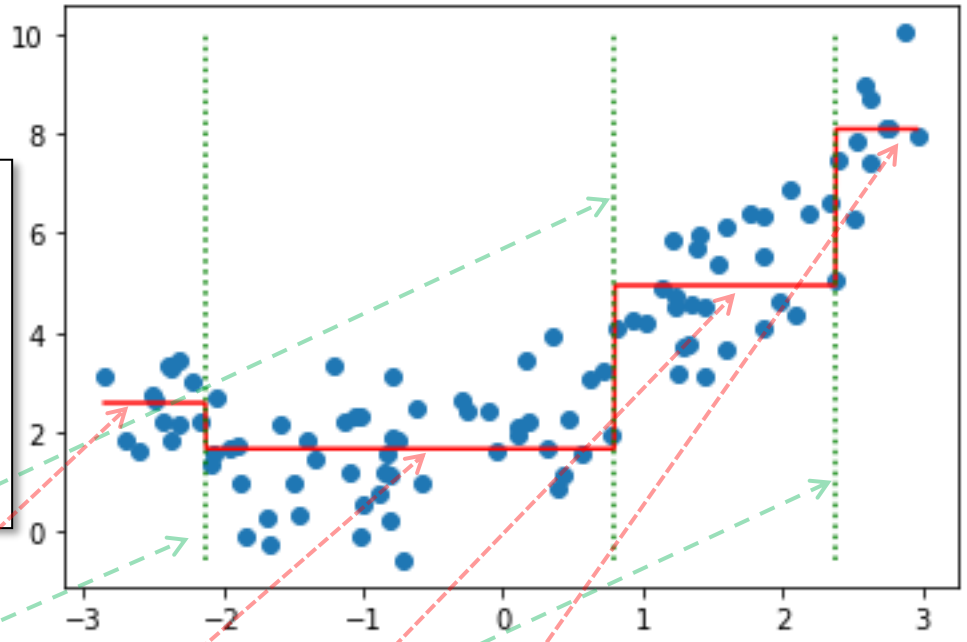


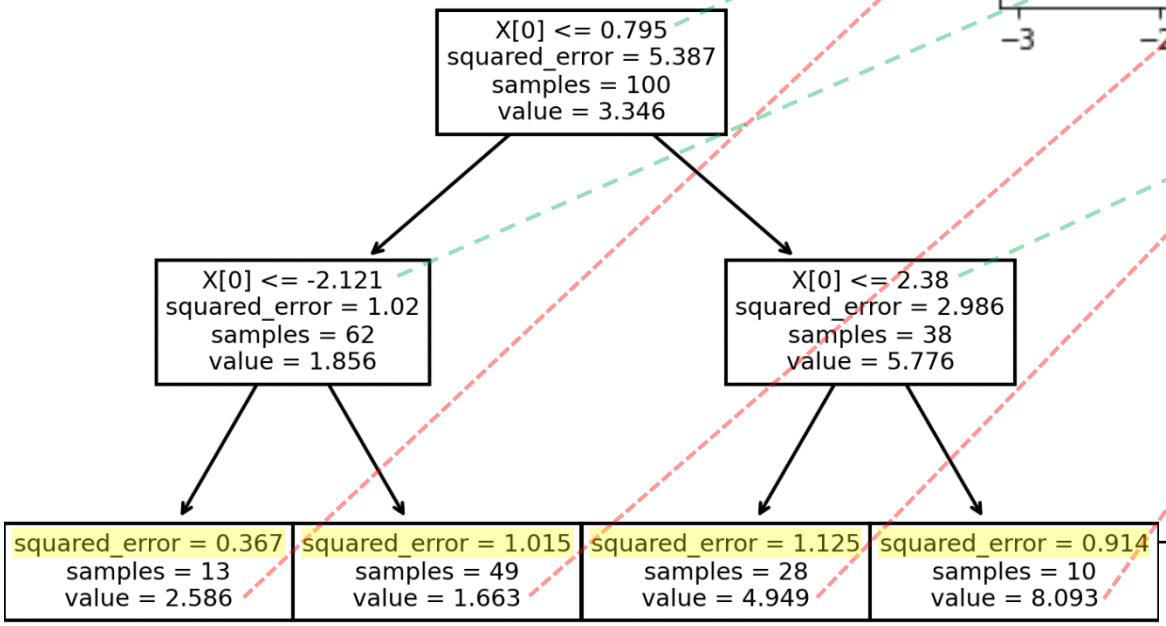
- The tree will look very **similar to the classification tree** we have built earlier.
- The main difference is that instead of predicting a class, it will **predict a value**.

Regression trees

- **Example 1**
  Noisy quadratic dataset

```
from sklearn.tree import DecisionTreeRegressor

# 1D quadratic synthetic dataset
m = 100
X = 6 * np.random.rand(m,1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m,1)

tree = DecisionTreeRegressor(max_depth=2)
tree.fit(X, y)
```





To use a regression tree, one has to traverse the tree starting at the root until reaching a leaf node,
e.g. for $x = 0.6$ the prediction would be 1.663

MSE global: $\frac{1}{4}\sum_j e_j^2 = 0.8553$

Regression trees

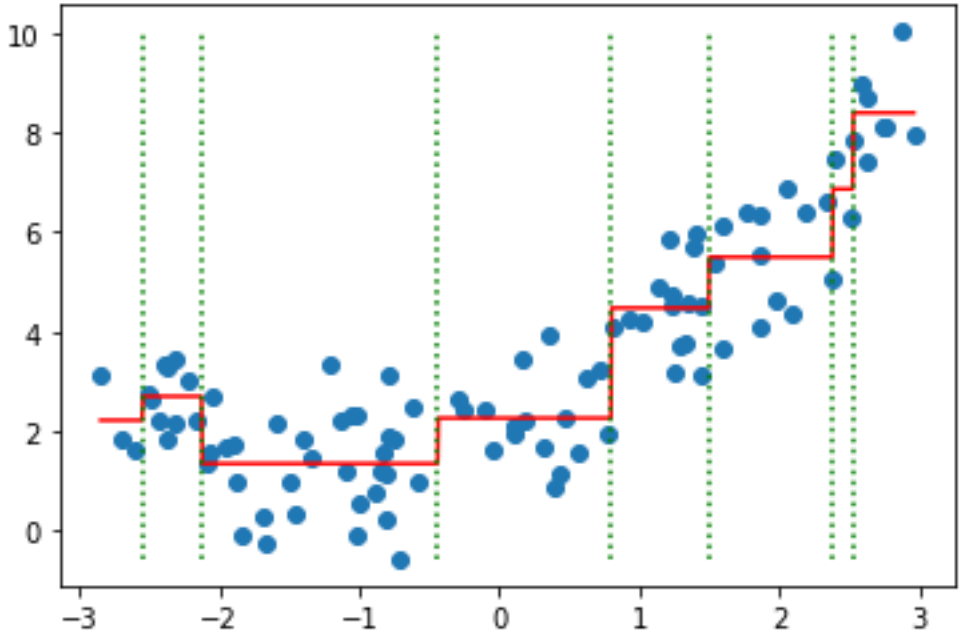
- In regression trees, predictions are obtained as the **average target value** for the samples associated to the corresponding leaf node
  - The quality of the predictions is calculated as the Mean Squared Error (MSE) of the target values for the involved samples, i.e. for a node N_t:

$$\mathrm{MSE}(N_t) = \frac{1}{n_t} \sum_{i \in N_t} (y_i - \bar{y}_t)^2, \quad \bar{y}_t = \frac{1}{n_t} \sum_{i \in N_t} y_i$$

- For the previous example, if we increase the tree depth, the MSE values get lower:

MSE at leaves (max. depth = 3)			
1	2	3	4
0.4450	0.2905	0.9338	0.6226
5	6	7	8
0.7204	1.0334	0.3388	0.5881

  - Average MSE for max_depth = 2 → 0.8553
  - Average MSE for max_depth = 3 → 0.6216



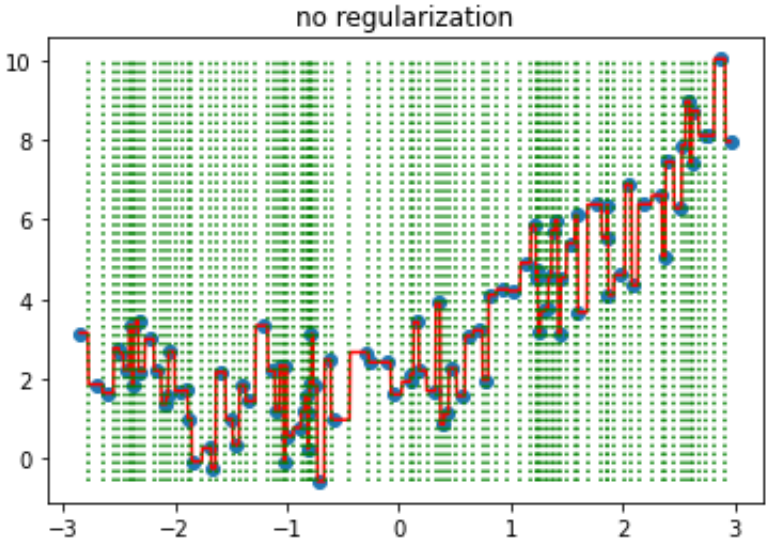
Regression trees

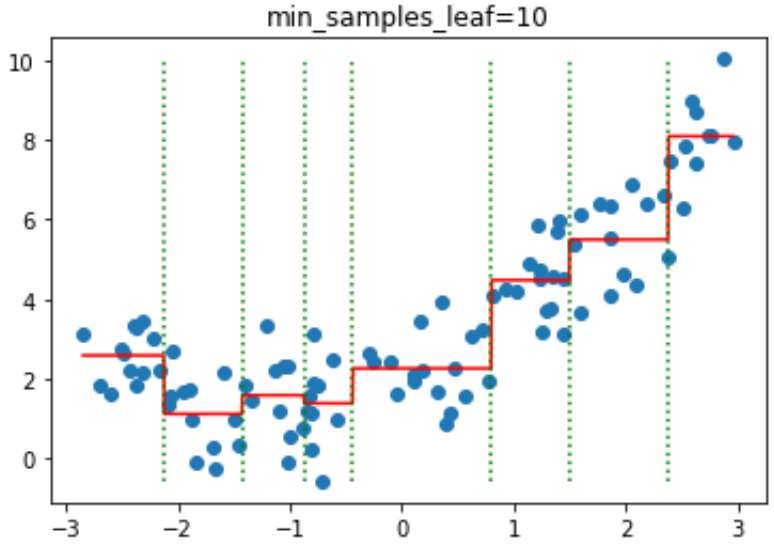
- The tree building algorithm works mostly the same way as for classification trees, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that **minimizes the MSE**:

$$\Delta\text{MSE}(N_t) = \text{MSE}(N_t) - \frac{n_t^Y}{n_t}\text{MSE}(N_t^Y) - \frac{n_t^N}{n_t}\text{MSE}(N_t^N)$$

  - The node with the highest reduction in MSE is chosen for splitting

- Just like for classification tasks, regression trees are also **prone to overfitting** if no regularization is performed:





Regression trees

- **Example 2**
  California housing dataset:
  - Median house value for California districts expressed in $100,000
    (1990 U.S. census, using one row per census block group [= smallest geographical unit that is published])
  - 20640 samples, 8 features, target $y \in 0.15 - 5$

```
from sklearn.datasets import fetch_california_housing
from sklearn.tree import DecisionTreeRegressor

X, y = fetch_california_housing(return_X_y=True)

tree = DecisionTreeRegressor(max_leaf_nodes=10)
tree.fit(X, y)
```

MSE at leaves				
1	2	3	4	5
0.670	0.394	0.702	1.168	0.536
6	7	8	9	10
0.438	0.509	1.006	0.526	0.778

Contents

- Decision trees
- Regression trees
- Ensemble learning
- Random forests

Ensemble learning

- Suppose you ask a complex question to thousands of random people and then **aggregate** their answers
  - In many cases you will find that this aggregated answer is better than a single expert's answer
  - This is called the ***wisdom of the crowd***
- Similarly, if you **aggregate the predictions of a group of predictors** (classifiers or regressors), you will often get better predictions than with the best individual predictor
  - A group of predictors is called an **ensemble**
  - Hence, this technique is called **Ensemble Learning**
  - An Ensemble Learning algorithm is called an **Ensemble method**
- In general, ensemble classifiers **scale very well** as training and predictions can be performed in parallel, i.e. via different CPU cores/servers
- You will often use ensemble methods **near the end of a project**, once you have already built a few good predictors, to combine them into an even better predictor

Ensemble learning

- It is possible to find a number of ensemble methods:
  - **Voting** (**hard** and **soft** voting)
  - **Bagging** (and **pasting**)
  - **Stacking** (and **blending**)
  - Boosting (Adaptive boosting/Adaboost and variants, Gradient boosting/Gradient boosted regression trees [GBRT], Extreme gradient boosting [XGB])
  - Others …

- In this section, we will discuss the **voting**, **bagging** and **stacking** ensemble methods
  - The boosting methods are built by **sequentially adding predictors** to an ensemble, each one correcting its predecessor

    … while **voting**, **bagging** and **stacking** consider all the available predictors at once

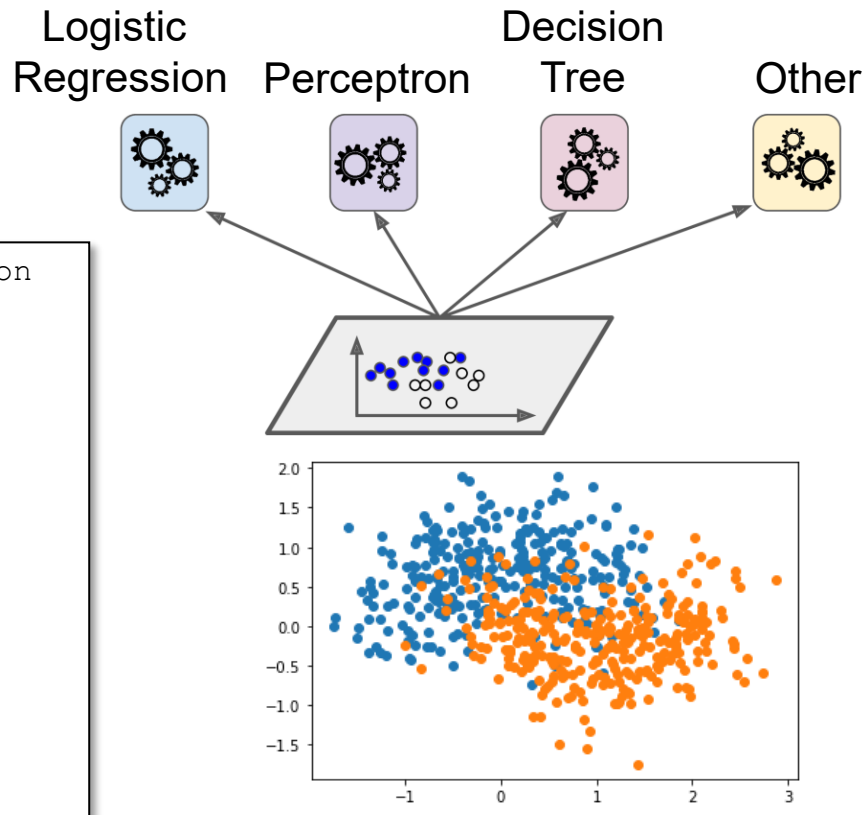
Voting classifier

- Suppose you have trained a few classifiers, each one achieving a certain accuracy

  e.g. a Logistic Regression classifier, a Perceptron, a Decision Tree, and maybe a few more

```
from sklearn.model_selection
   import train_test_split
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=600, noise=0.4)
X_train, X_test, y_train, y_test =
   train_test_split(X, y, test_size=0.3)
```

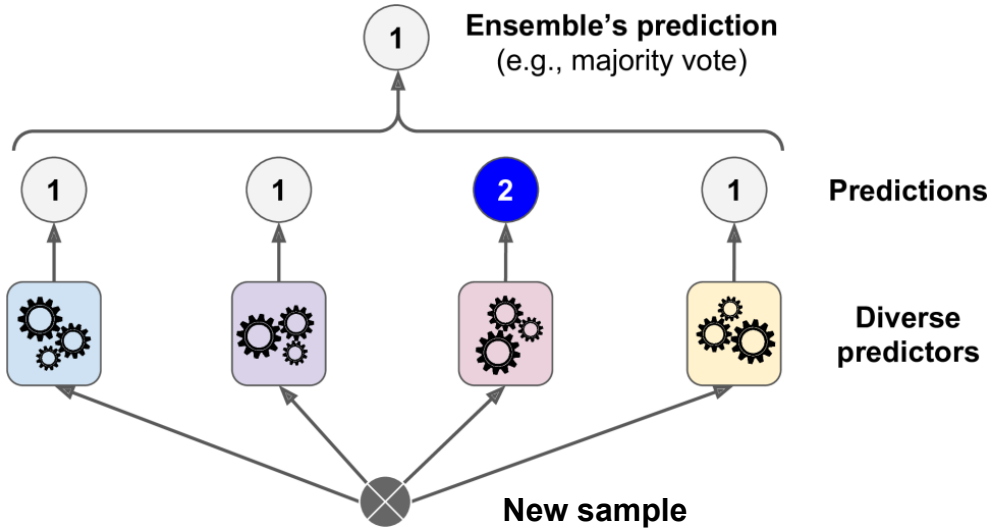
```
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import Perceptron
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

log_clf = LogisticRegression()
per_clf = Perceptron(tol=1e-3)
tre_clf = DecisionTreeClassifier(
    criterion="gini",
    max_depth=2,
    min_samples_leaf=5,
    min_impurity_decrease=0.1)

for clf in (per_clf, log_clf, tre_clf):
  clf.fit(X_train, y_train)
  y_pred = clf.predict(X_test)
  print(accuracy_score(y_test, y_pred))
```




```
0.7444 # Perceptron
0.8222 # Logistic Regression
0.7778 # Decision Tree
```

Voting classifier

- A very simple way to create a better classifier is to aggregate the predictions of each classifier and predict the class that **gets the most votes**.
  - This majority-vote classifier is called a **hard voting classifier**.




```
from sklearn.ensemble import VotingClassifier
vot_clf = VotingClassifier(
  estimators=[('per', per_clf), ('lr', log_clf),
        ('dt', tre_clf)], voting='hard')
vot_clf.fit(X_train, y_train)
y_pred = vot_clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

```
0.8278 # hard voting classif.
```

Voting classifier

- If the classifiers are able to estimate class probabilities, then you can take them into account and get a **soft voting classifier**.
  - In scikit-learn, the individual classifiers has to have a `predict_proba()` method
  - In such a case, `voting = "soft"`.

Two-class classification problem

$$f_1(x) = p(\omega_1|x) = 0.8$$
$$f_2(x) = p(\omega_1|x) = 0.51$$
$$f_3(x) = p(\omega_1|x) = 0.1$$
$$f(x) = \frac{0.8 + 0.51 + 0.1}{3} = \frac{0.141}{3} = 0.047 < 0.5$$
$$\Rightarrow x \rightarrow \omega_2$$

Multi-class classification problem

$$f_1(x) = \begin{bmatrix} 0.05 \\ 0.15 \\ 0.71 \\ 0.09 \end{bmatrix} \quad f_2(x) = \begin{bmatrix} 0.20 \\ 0.12 \\ 0.43 \\ 0.25 \end{bmatrix} \quad f_3(x) = \begin{bmatrix} 0.17 \\ 0.03 \\ 0.51 \\ 0.29 \end{bmatrix}$$

$$f(x) = \frac{1}{3}\begin{bmatrix} 0.05 + 0.20 + 0.17 \\ 0.15 + 0.12 + 0.03 \\ 0.71 + 0.43 + 0.51 \\ 0.09 + 0.25 + 0.29 \end{bmatrix} = \begin{bmatrix} 0.14 \\ 0.10 \\ 0.55 \\ 0.21 \end{bmatrix}$$

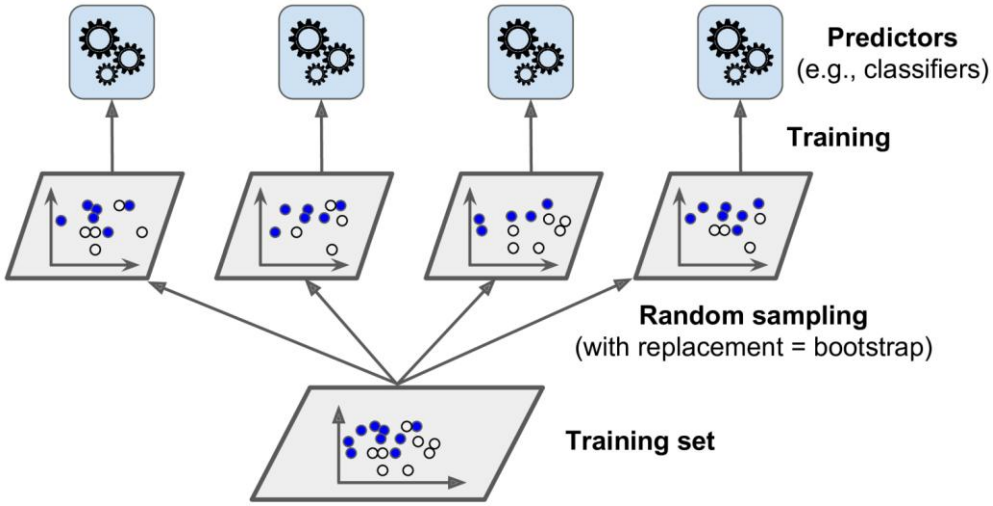
$$\Rightarrow x \rightarrow \omega_3$$

```
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import VotingClassifier
gnb_clf = GaussianNB()
vot_clf = VotingClassifier(
  estimators=[('gnb', gnb_clf), ('lr', log_clf),
        ('dt', tre_clf)], voting='soft')
vot_clf.fit(X_train, y_train)
y_pred = vot_clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

→ `0.8167 # soft voting classif.`

Bagging

- Apart from using different models and voting on the results, one can use the **same model but trained on different random subsets** of the training set:
  - **Bagging** (*bootstrap aggregating*): sampling with replacement (*bootstrapping* in statistics)
  - **Pasting**: sampling without replacement

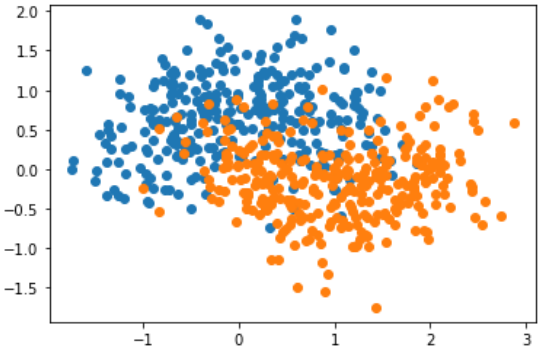


- Once all predictors are trained, the ensemble can make a prediction for a new sample by simply aggregating the predictions for all:
  - **statistical mode** for classification
  - **average** for regression

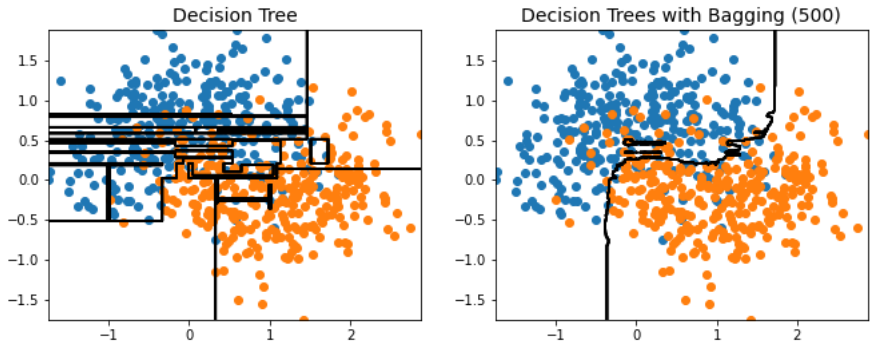
Bagging

- The following code trains an **ensemble of decision trees** (= ntree), each trained on 100 training samples randomly chosen from the training set with replacement
  - The bagging classifier automatically performs **soft voting** if the base classifier, e.g. decision tree, can estimate class probabilities

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=600, noise=0.4)
X_train, X_test, y_train, y_test =
   train_test_split(X, y, test_size=0.3)
```



```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier

tree_clf = DecisionTreeClassifier()
tree_clf.fit(X_train, y_train)
y_pred_t = tree_clf.predict(X_test)
print(accuracy_score(y_test, y_pred_t))

bag_clf = BaggingClassifier(
  DecisionTreeClassifier(), n_estimators=ntree,
  max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred_e = bag_clf.predict(X_test)
print(accuracy_score(y_test, y_pred_e))
```




```
0.8056 # tree classifier
0.8722 # bagging classifier (10)
0.8556 # bagging classifier (500)
```

Bagging

- With bagging, some instances may be sampled several times for any given predictor (because it is sampling with replacement), while others may not be sampled at all.
- The training samples that are not chosen are called **out-of-bag** (oob) samples.
- Since a predictor never sees the oob samples during training, it can be evaluated on these samples, without the need for a separate validation set, as this code shows:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier

bag_clf = BaggingClassifier(
  DecisionTreeClassifier(), n_estimators=500,
  max_samples=100, bootstrap=True, n_jobs=-1,
  oob_score=True)
bag_clf.fit(X_train, y_train)
print(bag_clf.oob_score_)
y_pred_e = bag_clf.predict(X_test)
print(accuracy_score(y_test, y_pred_e))
```

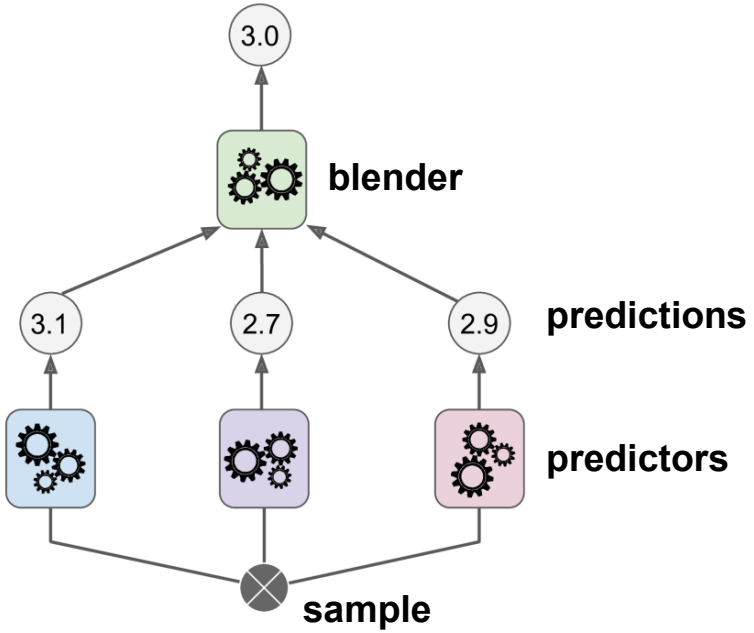
```
0.8690 # from OOB
0.8556 # from test set
```

- Bagging can be easily applied to **regression** problems (provided the base model is also a regression model):

```
rf_reg = BaggingRegressor(DecisionTreeRegressor(),
           n_estimators=500, max_samples=100, n_jobs=-1)
```

Stacking

- **Stacking** is a short form for **stacked generalization**
- Instead of using trivial functions (such as hard voting) to aggregate the predictions, this ensemble method **trains a model** to perform the aggregation
- By way of example, let us consider a regression task:
  - Each of the ensemble models, named as **first-layer predictors**, predict a different value: 3.1, 2.7, 2.9
  - The final predictor, named as **blender**, **meta learner** or **second-layer predictor**, takes these predictions as inputs and makes the final prediction 3.0



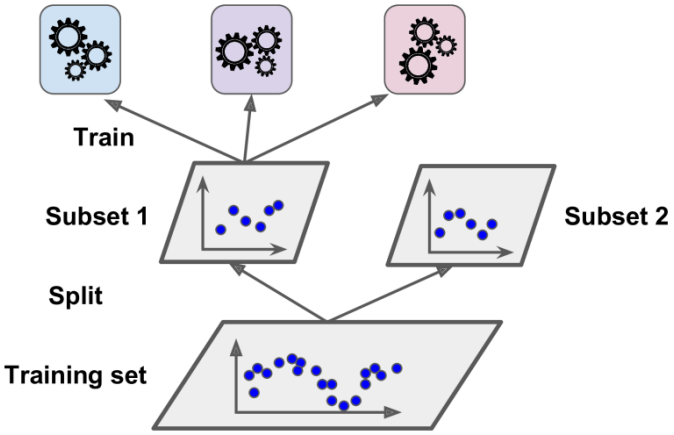
- Notice that the layer 2 can implement any form of aggregation:
  - Therefore, weighted average and even voting can be considered as **particular cases of stacking**, where the layer 2 implements (a) the average of the predictors' outputs or (b) a majority voting mechanism

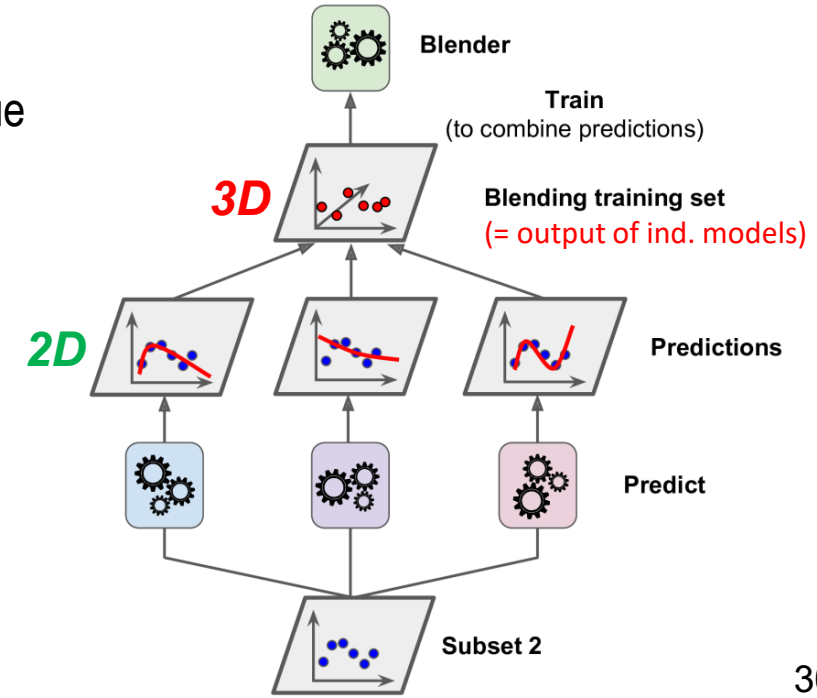
Stacking

- Training can be accomplished in several ways:
  - by means of the **hold-out set** concept, leading to **blending** instead of stacking

    or
  - by means of **k-fold cross validation**, giving rise to the **stacking** method itself
- Let us illustrate the **blending** method first: the hold-out set concept splits the training set into two subsets, where
  - the **subset 1** is used for training the first-layer predictors
  - the **subset 2** is used for training the blender

  $\Rightarrow$ the blender learns to predict the target value given the first-layer predictions





Stacking

- A python implementation of **stacking** follows: (*moons* dataset)

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import StackingClassifier
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                   test_size=0.3)
# build layer 1
layer1 = []
layer1.append(('bc', GaussianNB()))
layer1.append(('lr', LogisticRegression()))
layer1.append(('dt', DecisionTreeClassifier()))
# build stack
layer2 = LogisticRegression(penalty=None)
ensemble_clf = StackingClassifier(estimators=layer1,
                        final_estimator=layer2, cv=5)
# 1. train & evaluate individual classifiers
for name, model in layer1:
 acc = model.fit(X_train, y_train).score(X_test,y_test)
 print('%s %.3f' % (name, acc))
# 2. train & evaluate ensemble classifier, retrain ind. clas.
acc = ensemble_clf.fit(X,y).score(X_test,y_test)
print('%s %.3f (%.3f)' % ('stacking',acc))
```

Stacking:

- The 1st layer estimators are trained using the training set
- The 2nd layer estimator is trained through cross-validation

```
bc 0.828
lr 0.822
dt 0.806
stacking 0.833
```

Stacking

- A python implementation of **stacking** follows: (*moons* dataset)

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import StackingClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
```

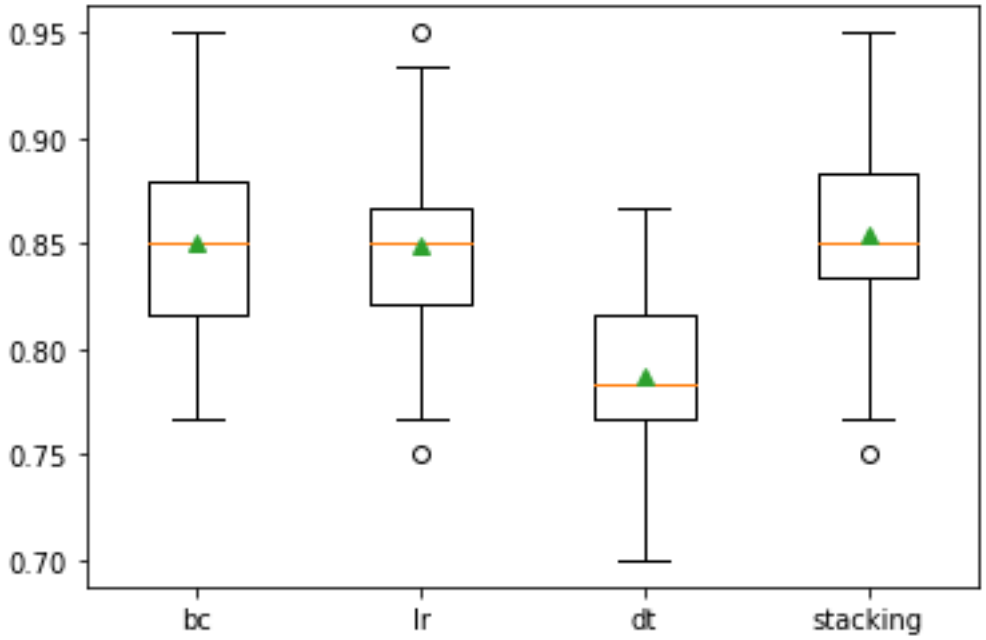
```
# build layer 1
layer1 = []
layer1.append(('bc', GaussianNB()))
layer1.append(('lr', LogisticRegression()))
layer1.append(('dt', DecisionTreeClassifier()))
# build stack
layer2 = LogisticRegression(penalty=None)
ensemble_clf = StackingClassifier(estimators=layer1,
                                  final_estimator=layer2)
# 1. train & evaluate individual classifiers
for name, model in layer1:
 scores = evaluate_model(model, X, y)
 print('%s %.3f (%.3f)' %
    (name, np.mean(scores), np.std(scores)))
# 2. train & evaluate ensemble classifier
scores = evaluate_model(ensemble_clf, X, y)
print('%s %.3f (%.3f)' %
   ('stacking', np.mean(scores), np.std(scores)))
```

```
def evaluate_model(model, X, y):
 cv = RepeatedStratifiedKFold(
      n_splits=10,n_repeats=3)
 scores = cross_val_score(model, X, y,
      scoring='accuracy', cv=cv,
      n_jobs=-1, error_score='raise')
 return scores
```

30 evaluations !!

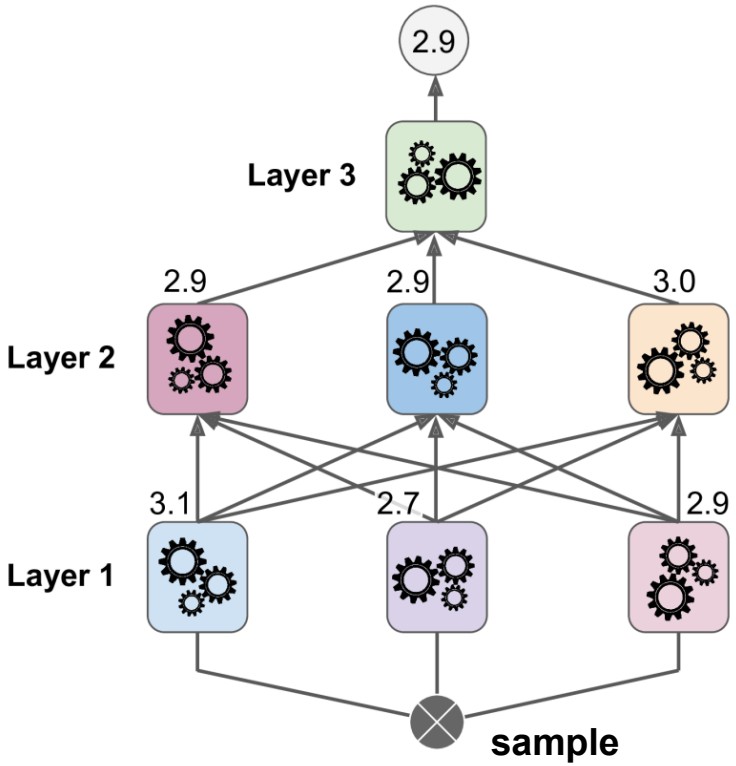
```
bc 0.850 (0.045)
lr 0.849 (0.045)
dt 0.796 (0.045)
stacking 0.852 (0.047)
```

- A python implementation of **stacking** follows (cont.)



Stacking

- It is actually possible to train several blenders, e.g. using different classifiers, to get a **whole layer of blenders**
- Now, the trick is to split the training set into **three subsets**:
  - the 1st subset is used to train the first layer,
  - the 2nd subset is used to create the training set for the second layer (using predictions made by the predictors of the first layer), and
  - the 3rd subset is used to create the training set for the third layer (using predictions made by the predictors of the second layer)



- Once all this is done, we can make a prediction for a new sample by **going through each layer sequentially**
- In **scikit-learn**, multiple stacking layers can be achieved by assigning *final_estimator* to a *StackingClassifier*

Contents

- Decision trees
- Regression trees
- Ensemble learning
- Random forests

Random Forests

- A **random forest** (RF) is an ensemble of decision trees, generally trained via the bagging method (sometimes pasting):

```
bag_clf = BaggingClassifier(
  DecisionTreeClassifier(), n_estimators=500, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
```

  - It has proved so **successful** that this ensemble method has its own name and an optimized implementation:

```
from sklearn.ensemble import RandomForestClassifier
rf_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
rf_clf.fit(X_train, y_train)
```

  - With a few exceptions, an RF has all the **hyperparameters of a DT**, e.g. to control how trees are grown, plus all the **hyperparameters of a bagging classifier** to control the ensemble itself

```
rf_clf = RandomForestClassifier(min_samples_leaf=10, max_depth=4,
                n_estimators=500, max_samples=100, n_jobs=-1)
```

Random Forests

- Actually, an RF introduces **extra randomness** when growing trees:
  - Instead of searching for the best feature when splitting a node, it searches for the **best feature among a random subset of features**
  - This results in a greater tree diversity, and generally yielding an overall better model

```
rf_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
rf_clf.fit(X_train, y_train)
```

```
bag_clf = BaggingClassifier(
  DecisionTreeClassifier(splitter="random"),
  n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

- *splitter* = "random" → choose the best split for a random subset of features
- *max_samples* = 1.0 → use all samples for training

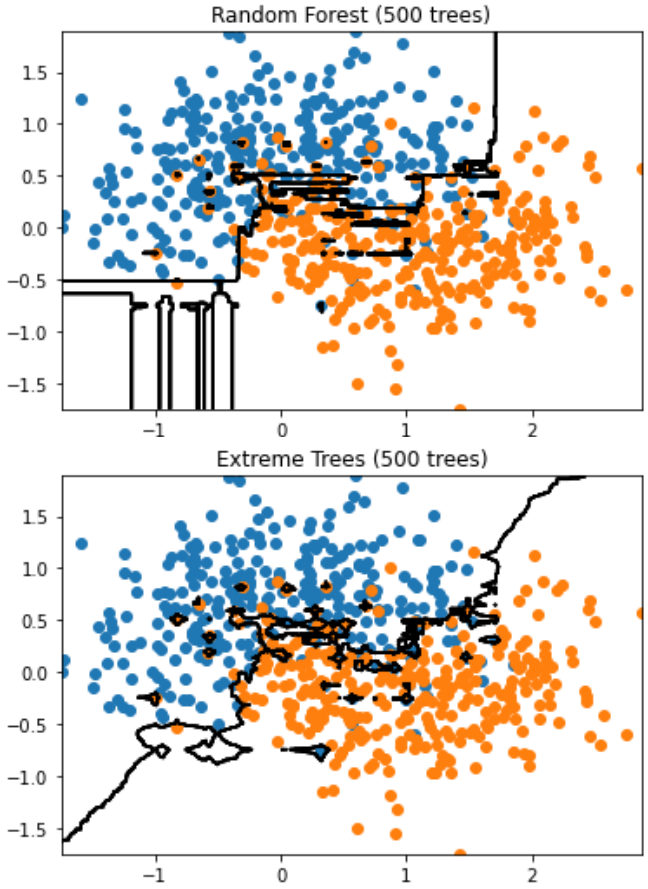
- As expected, random forests can be used for **both classification** and **regression** tasks

```
from sklearn.ensemble import RandomForestRegressor
rf_reg = RandomForestRegressor(n_estimators=500, max_depth=2,
                               max_samples=100, n_jobs=-1)
rf_reg.fit(X_train, y_train)
```

- An **Extremely Randomized Trees** ensemble (or *Extra-Trees*, ET, for short) incorporates more randomness than RF:
  - Instead of optimizing the threshold at each splitting, these thresholds are chosen randomly
  - This trades more bias for a lower variance
  - It also makes ET much faster to train than regular RF

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier

rf_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
rf_clf.fit(X_train, y_train)
yp_rf = rf_clf.predict(X_test)
print(accuracy_score(y_test, yp_rf))

et_clf = ExtraTreesClassifier(n_estimators=500, n_jobs=-1)
et_clf.fit(X_train, y_train)
yp_et = et_clf.predict(X_test)
print(accuracy_score(y_test, yp_et))
```

```
0.8500 (time = 0.5806 ± 0.0282 seconds, ± 3σ)
0.8278 (time = 0.4552 ± 0.0257 seconds, ± 3σ)
```



Random Forests

- Yet another great quality of Random Forests is that they make it easy to measure the **relative importance of each feature**:
  - One can measure the importance of each feature by looking at **how much the tree nodes that use that feature reduce impurity** on average (across all trees in the forest)
  - More precisely, one can give a weighted average, where each node's weight is equal to the **fraction of training samples** associated with it

```
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier

iris = load_iris()
rf_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
rf_clf.fit(iris["data"], iris["target"])
for name, score in zip(iris["feature_names"],rf_clf.feature_importances_):
    print(name, score)
```

```
sepal length (cm) 0.1062
sepal width (cm)  0.0269
petal length (cm) 0.4434
petal width (cm)  0.4235
```

Lecture 3.3
Supervised learning:
Decision Trees & Ensemble Learning

Universitat de les Illes Balears

Departament de Ciències Matemàtiques i Informàtica

11752 Aprendizaje Automático
11752 Machine Learning
Máster Universitario
en Sistemas Inteligentes

Alberto ORTIZ RODRÍGUEZ