



SAPIENZA
UNIVERSITÀ DI ROMA

METODI DI INTELLIGENZA ARTIFICIALE E MACHINE LEARNING IN FISICA

S. Giagu - AA 2019/2020
Lezione 9: 25.3.2020

ALGORITMI NON-METRICI: ALBERI DI DECISIONI

- Fino a questo momento tutti i classificatori considerati erano basati su features vectors costituiti da vettori di numeri reali (o interi) per i quali era possibile definire una misura della distanza tra vettori (in altre parole della similitudine tra essi)
- supponiamo ora che il problema di classificazione coinvolga “descrizioni” che siano discrete e soprattutto **senza nessuna naturale nozione di similitudine esplicita**
- in questo caso un approccio efficace è quello di allontanarsi dalla descrizione dei patterns in termini di vettori di feature reali, e di utilizzare **liste di attributi**
- gli alberi di decisioni (**decision trees**) sono modelli computazionali concettualmente molto semplici in grado di descrivere quei casi in cui è richiesto un certo grado di “spiegazione” o “descrizione” della decisione di classificazione
- funzionano molto anche con feature convenzionali (vettori di numeri reali) e sono in grado di approssimare superfici di decisione complesse e non lineari per qualsiasi tipo di input (i.e. funzionano come classificatori, risolvono problemi di regressione, e individuano automaticamente le feature più interessanti rispetto ad un dato problema)

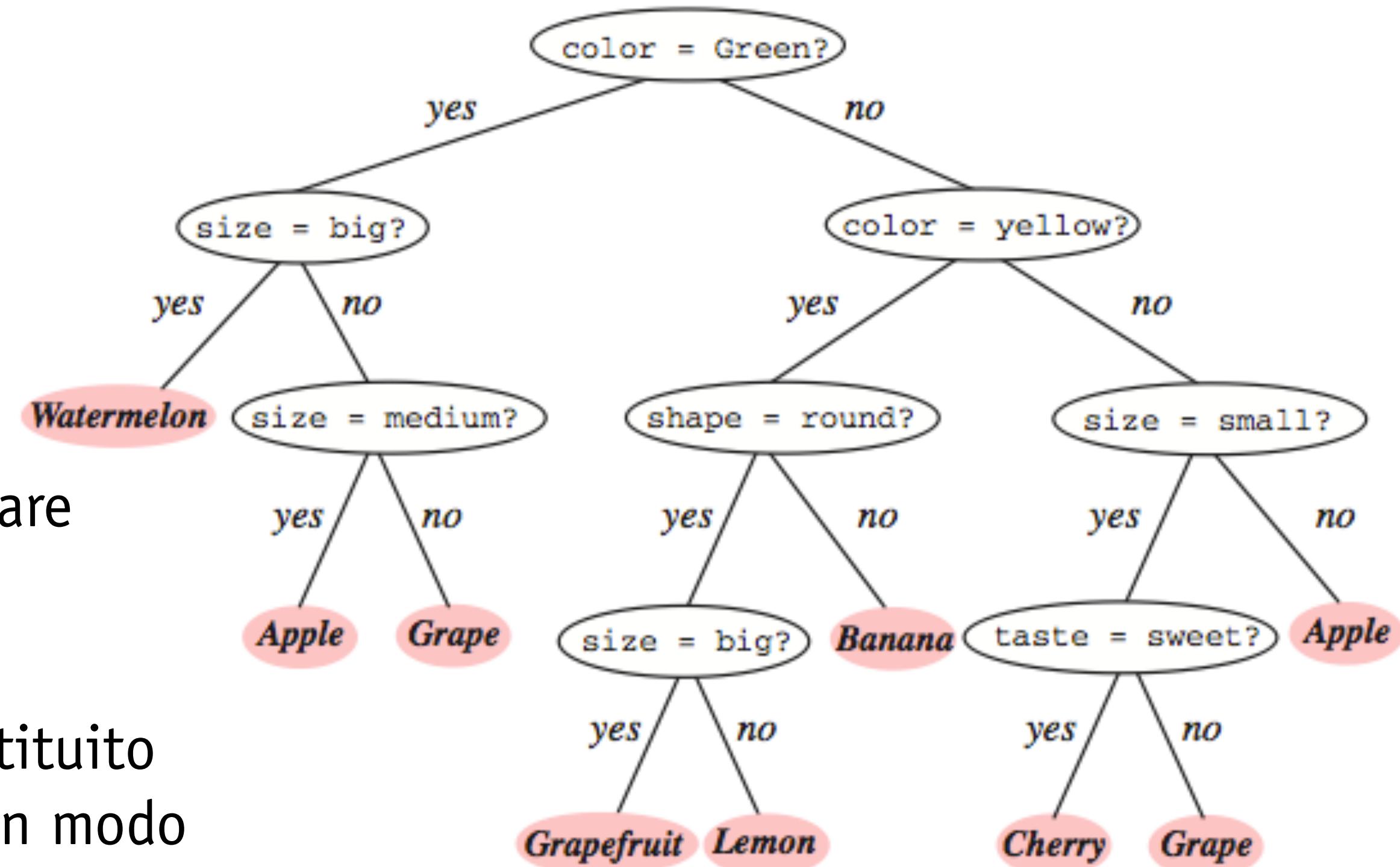


ALBERI DI DECISIONI BINARIE

Un albero di decisioni binarie (binary decision tree) è uno specifico tipo di grafo che non può contenere loops

- esempio: classificazione di un frutto

- oggetto da classificare rappresentato da una ntuple di 4 attributi: colore, forma, sapore, grandezza
- un possibile feature vector: {rosso, rotondo, dolce, piccolo}
- senza un criterio metrico di similitudine, è intuitivo classificare il frutto tramite una sequenza di domande binarie (alle quali dare una risposta si/no, vero/falso)
- la sequenza di domande viene rappresentata da un grafo costituito da una collezione di nodi e connessioni (edges) organizzate in modo gerarchico, in cui tutti i nodi (tranne il root node) hanno esattamente una connessione entrante e due connessioni uscenti



COSTRUZIONE DI UN BINARY DECISION TREE

**Principio guida: Occam's razor simplicity
i.e. massimizzare la purezza (purity) ad ogni suddivisione**

Gini-index: misura di dissimilarità tra distribuzioni

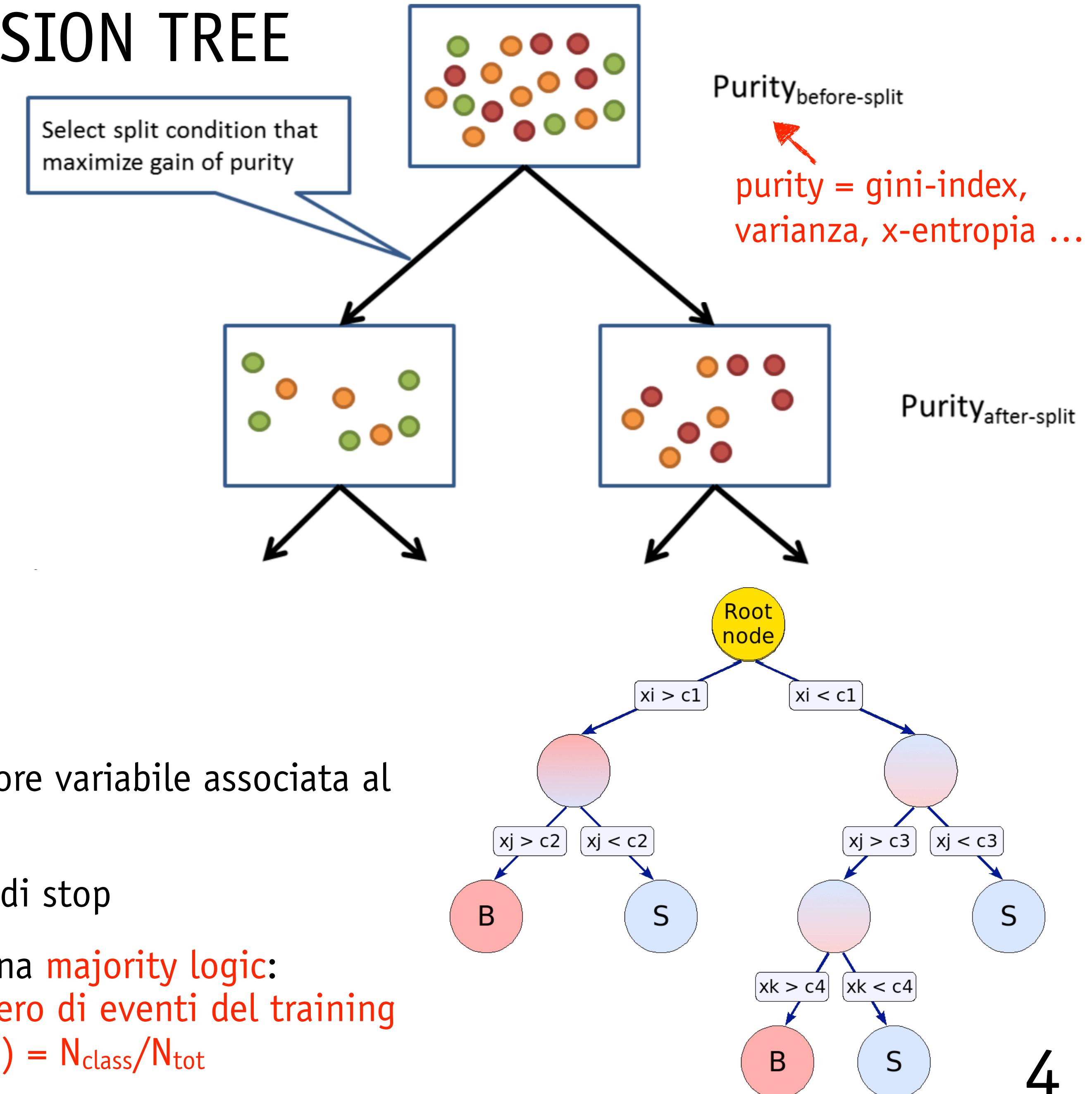
$$G = \sum p_i(1-p_i) \text{ con } p_i = N_{ci}/(N_{tot})$$

2 classi $c_1=S$ e $c_2=B$:

$S=B \rightarrow G = 0.5$ (minima similarità)

$S \gg 0 \ll B \rightarrow G = 0$ (massima dissimilarità)

- start: root node
- split del campione di training con un taglio sulla migliore variabile associata al nodo
- continua fino a quando si raggiunge un fissato criterio di stop
- ogni foglia terminale viene classificata in accordo ad una **majority logic**:
classe di assegnazione = quella con il maggior numero di eventi del training set nella foglia, oppure è possibile dare una $p(\text{class}) = N_{\text{class}}/N_{\text{tot}}$



ALGORITMO

Training

- › Input: training set $X^\ell = \{(\mathbf{x}_i, y_i)\}_{i=1}^\ell$

1. Greedily split X^ℓ into R_1 and R_2 :

$$R_1(j, t) = \{\mathbf{x} \in X^\ell | \mathbf{x}_j < t\}, \quad R_2(j, t) = \{\mathbf{x} \in X^\ell | \mathbf{x}_j > t\}$$

optimizing a given loss: $Q(X^\ell, j, t) \rightarrow \min_{(j, t)}$

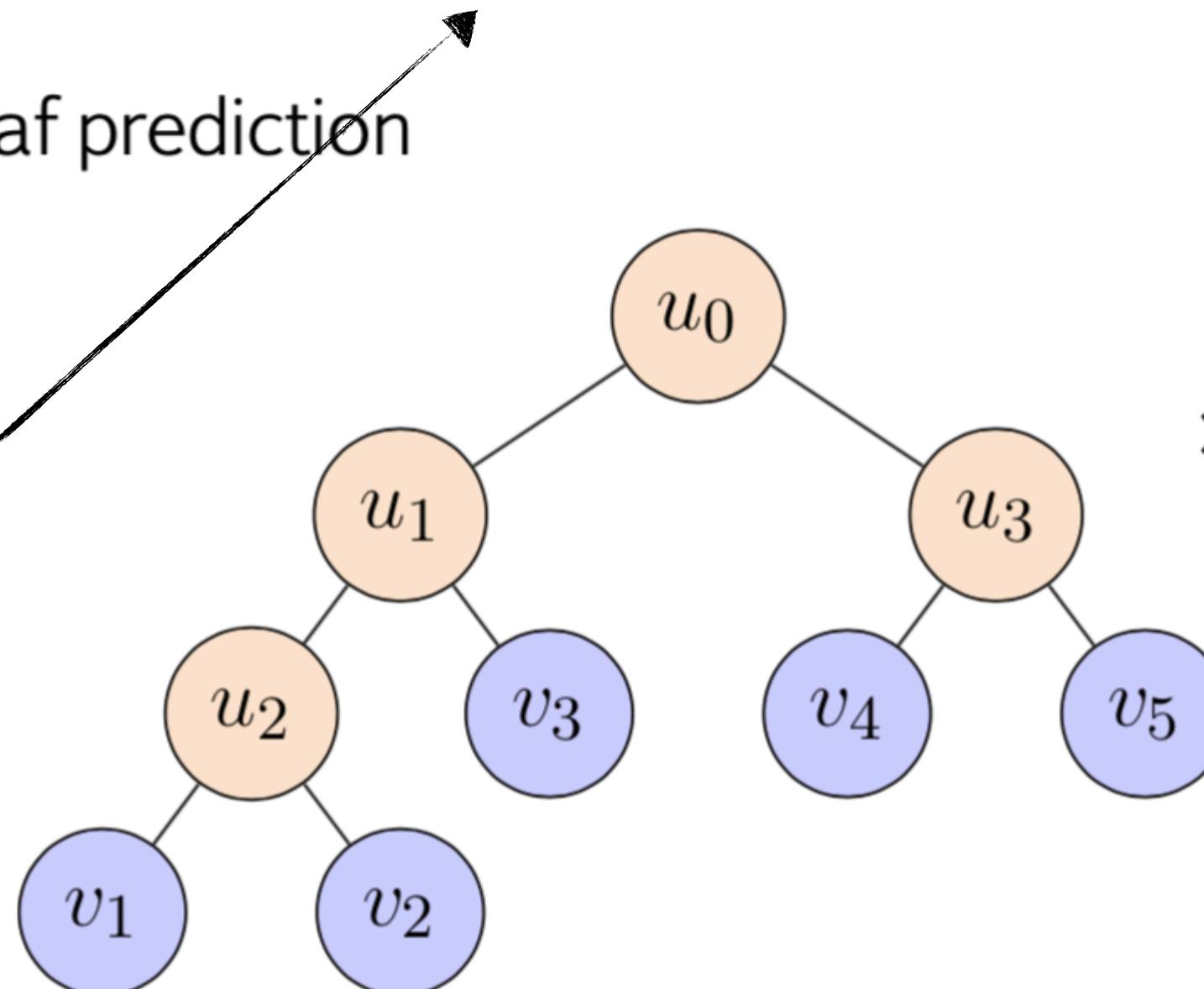
2. Create internal node u corresponding to the predicate $[\mathbf{x}_j < t]$
3. If a stopping criterion is satisfied for u ,
declare it a leaf, setting some $c_u \in \mathbb{Y}$ as leaf prediction
4. If not, repeat 1–2 for $R_1(j, t)$ and $R_2(j, t)$

- › Output: a decision tree V

NOTAZIONE: [...] = if (...) return 1; else 0

Application (prediction)

- › Decision tree is a binary tree V
- › Internal nodes $u \in V$: predicates
 $\beta_u : \mathbb{X} \rightarrow \{0, 1\}$
- › Leafs $v \in V$: predictions x
- › Algorithm $h(\mathbf{x})$ starts at $u = u_0$
 - › Compute $b = \beta_u(\mathbf{x})$
 - › If $b = 0$, $u \leftarrow \text{LeftChild}(u)$
 - › If $b = 1$, $u \leftarrow \text{RightChild}(u)$
 - › If u is a leaf, return b
- › In practice: $\beta_u(\mathbf{x}; j, t) = [\mathbf{x}_j < t]$



CRITERIO DI STOP

-Criterio di stop tipicamente deciso tramite ricerca esaustiva o cross-validation:

- numero minimo di eventi nelle foglie
- massima profondità dell'albero
- massimo numero di foglie
- stop se tutti gli eventi di una classe finiscono nella stessa foglia
- miglioramento in un parametro di qualità (esempio purezza) inferiore ad una certa %



NOTA: K-FOLD CROSS VALIDATION

- metodo statistico costituito da una procedura di re-sampling usato di routine nella stima delle prestazioni e nella ottimizzazione degli iperparametri di modelli di ML in presenza di campioni di dimensioni limitate
 - rispetto all'uso del validation set fissato (training+validation) minimizza l'effetto delle fluttuazioni statistiche presenti nel caso di campioni di dimensione troppo ridotta e permette di migliorare sensibilmente l'ottimizzazione di un dato modello
1. si suddivide il training set T in K partizioni di eguale dimensione: $T = T_1 \cup T_2 \cup \dots \cup T_K$
 2. si addestrano/costruiscono K modelli g_1, \dots, g_K , in cui il k -esimo modello è addestrato usando $K-1$ campioni (escludendo il campione T_k)
 3. si valutano le prestazioni di ogni modello g_i usando come validation set T_i
 4. l'indice di prestazione finale è dato dalla media delle prestazioni dei K -test:

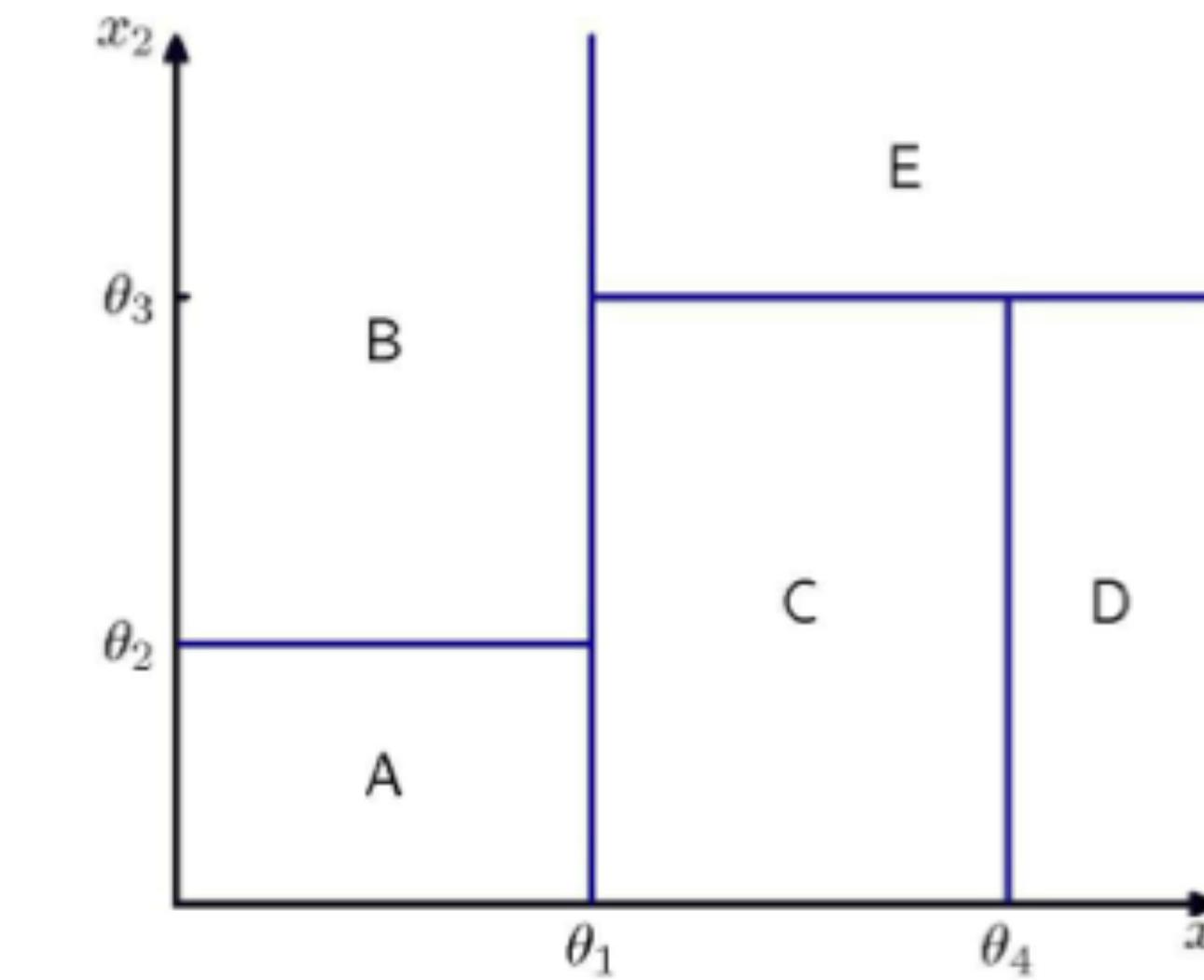
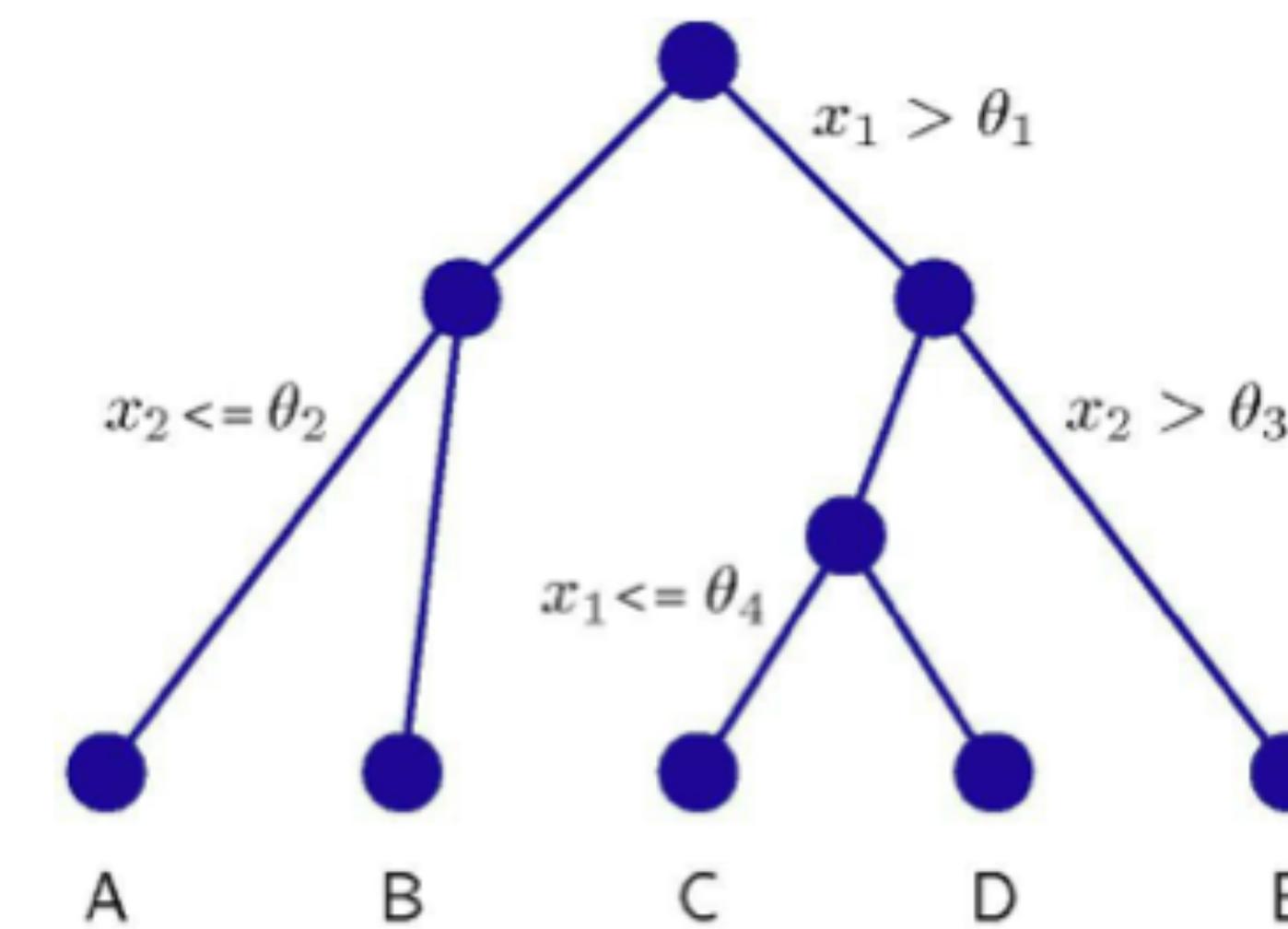
$$\text{CV-Score} = \frac{1}{K} \sum_{i=1}^K \text{Score}(g_i, T_i)$$



DECISION TREES E SUPERFICI DI SEPARAZIONE

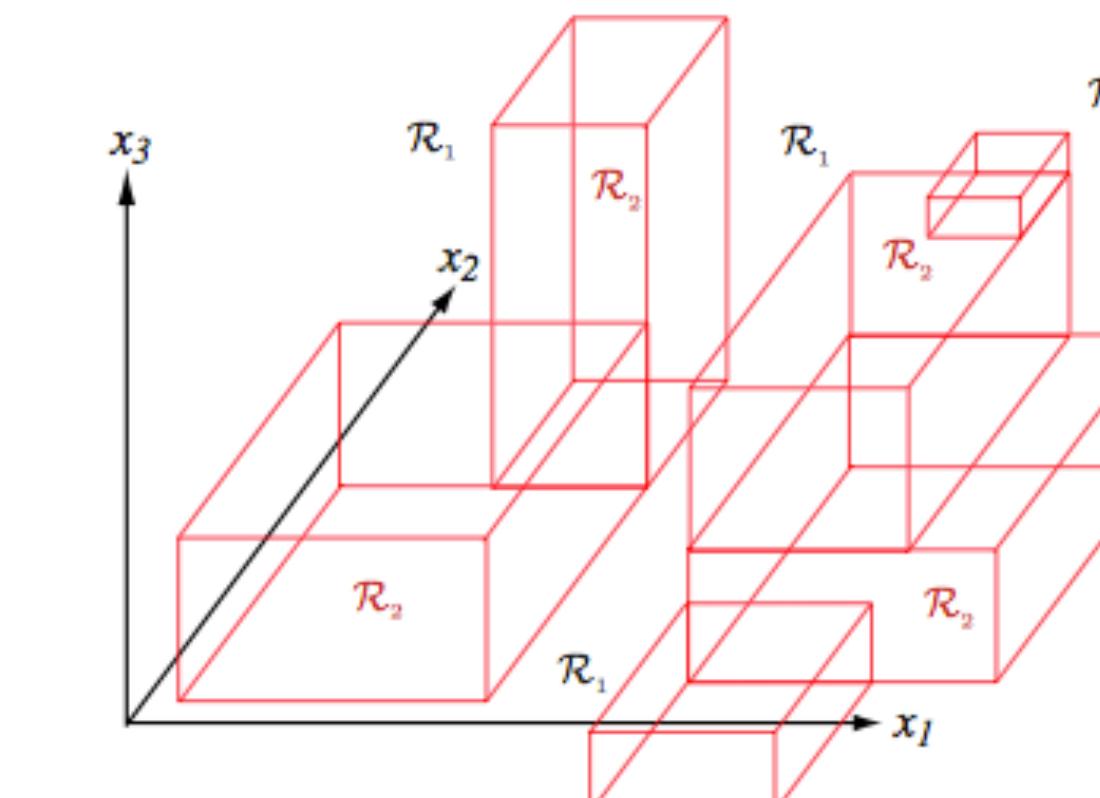
geometricamente un decision tree è semplicemente un histogramma d-dimensionale i cui bin sono costruiti ricorsivamente

ad ogni bin è associato il valore della funzione $g(x)$ che l'algoritmo approssima con funzioni costanti a tratti



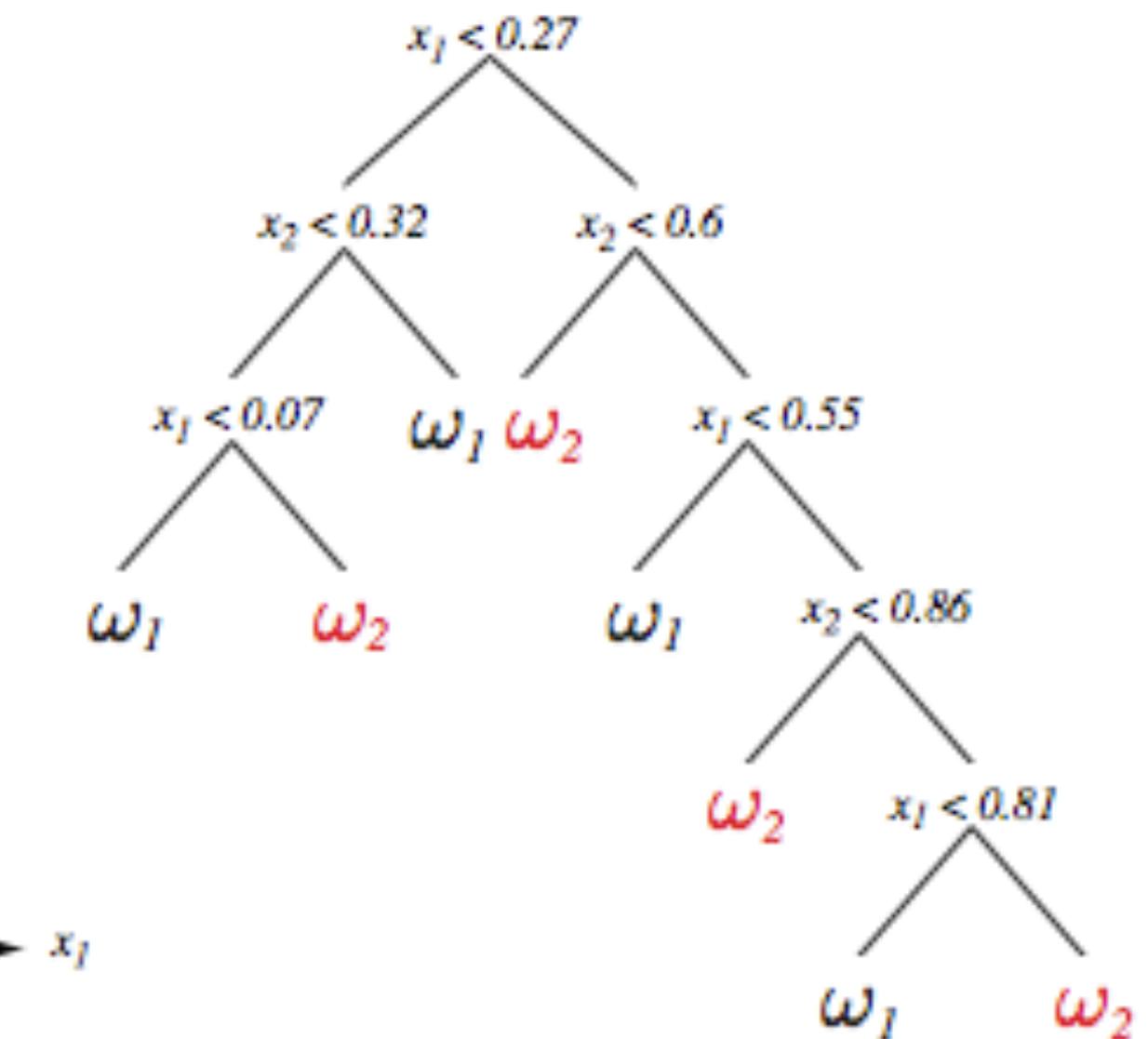
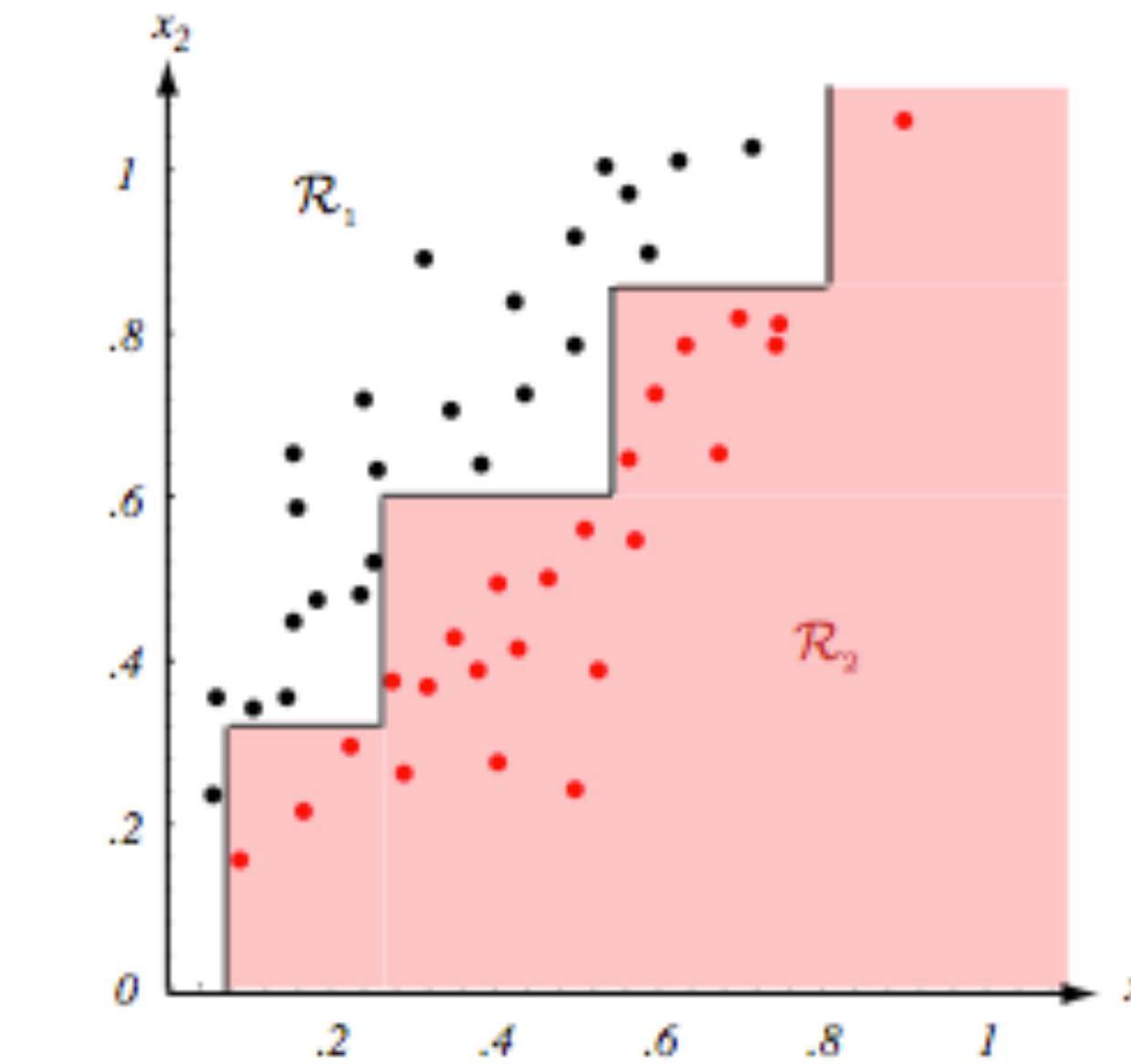
in >2 dim: iper-parallelepipedi

con un albero sufficientemente profondo è possibile approssimare arbitrariamente bene qualsiasi iper-superficie

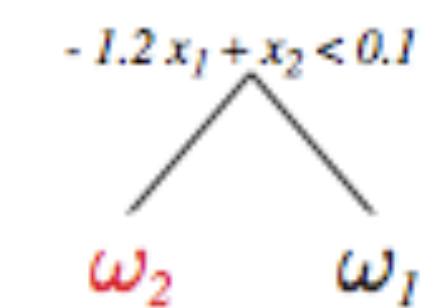
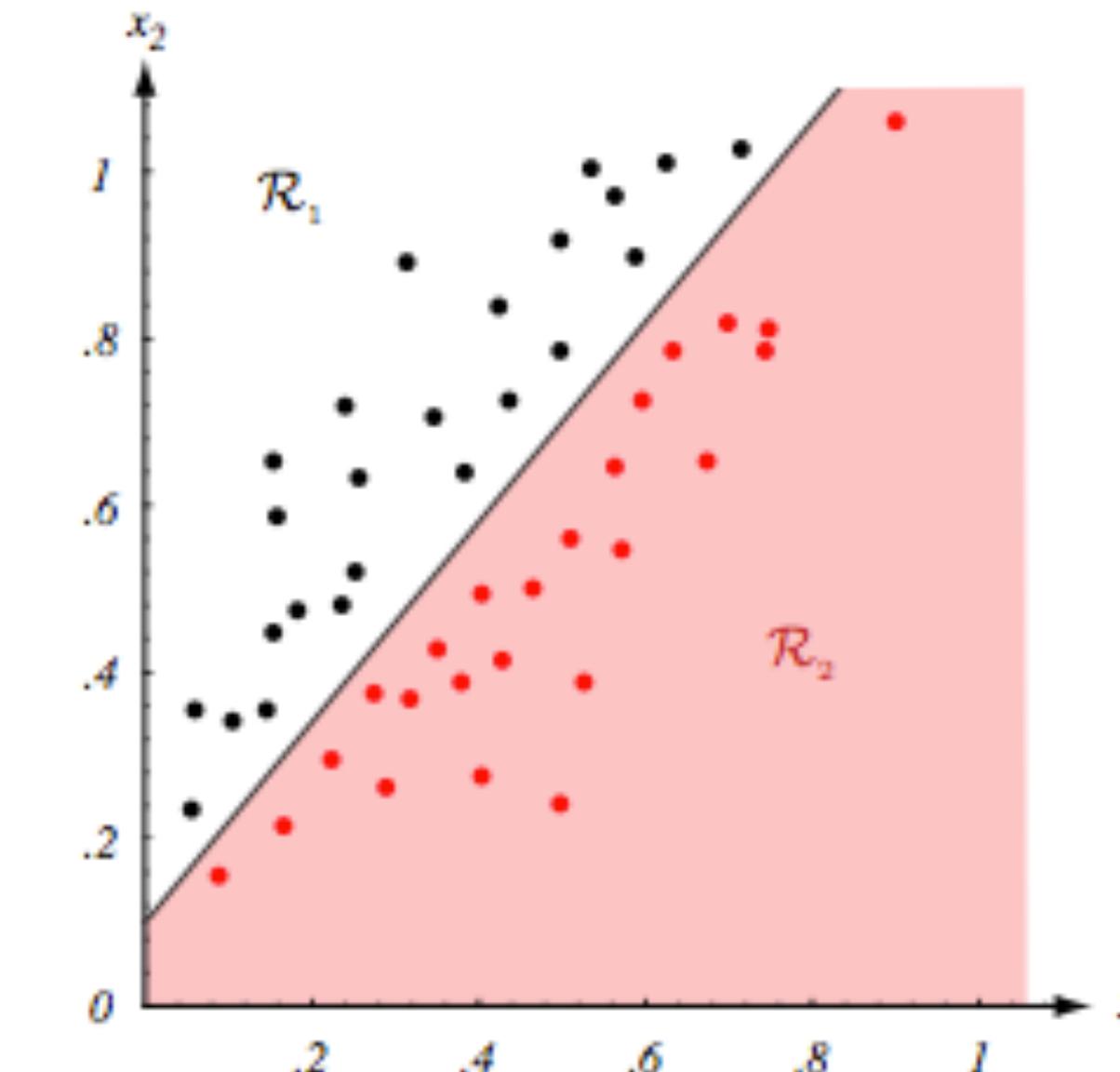


SCELTA DELLE FEATURE IN UN ABERO DI DECISIONI

se la regole di decisione nei nodi non rappresentano la forma dei dati di training è facile ottenere superfici complesse e non ottimali ...



è utile permettere nell'implementazione delle regole di decisione forme funzionali delle feature in input (per esempio combinazioni lineari) per ottenere grafi semplici e robusti (che generalizzano meglio) ...



BINARY DECISION TREES: VANTAGGI E PROBLEMI

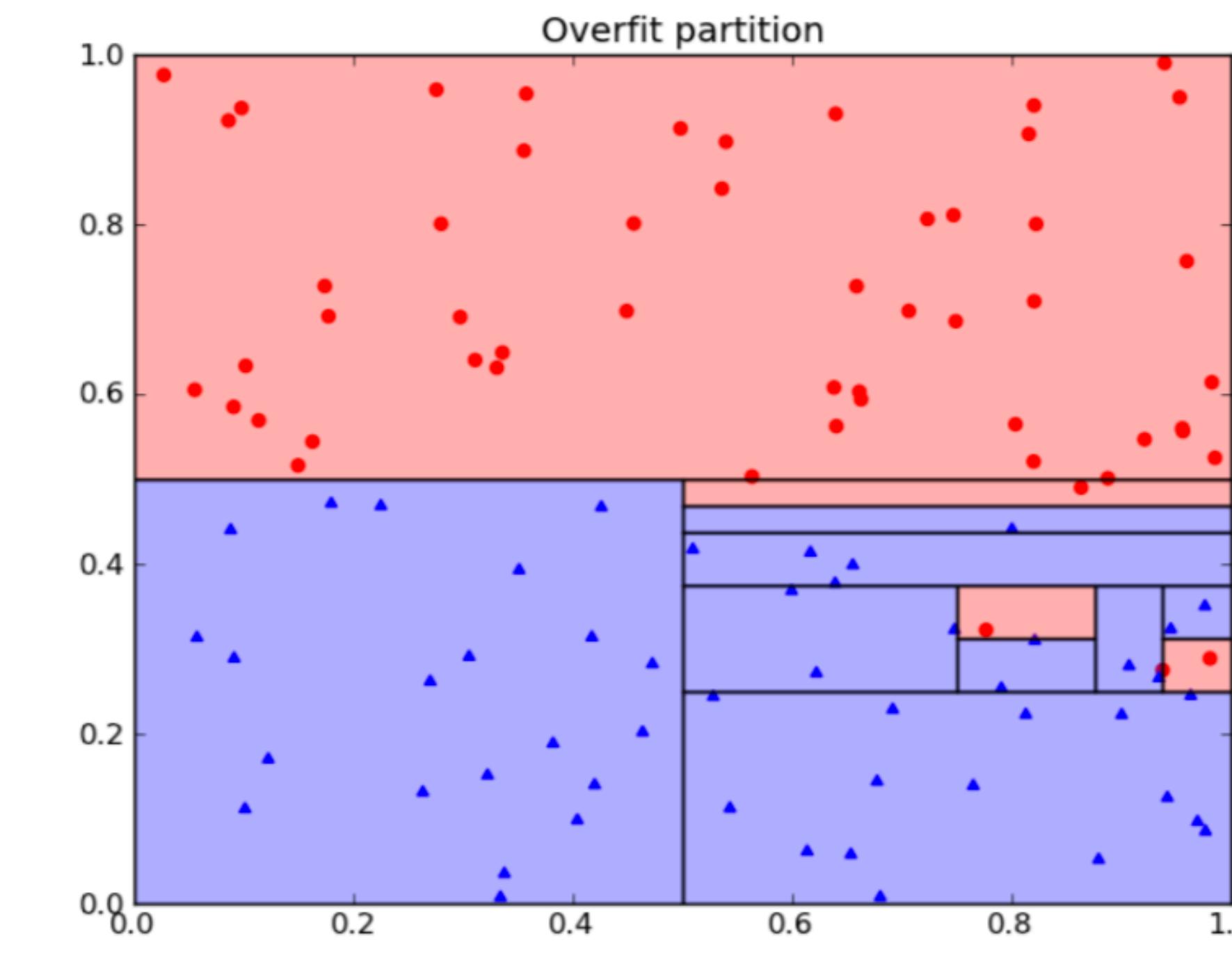
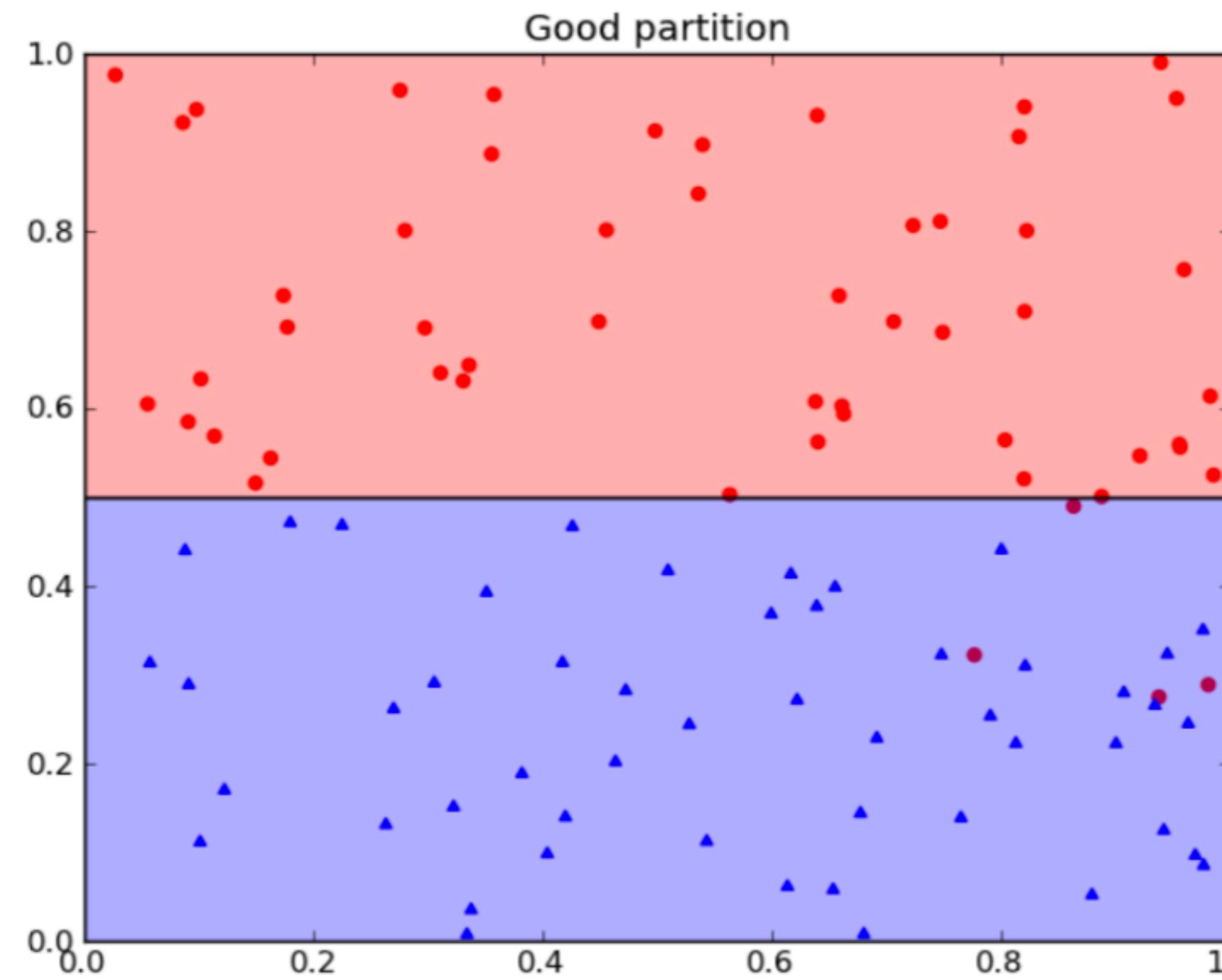
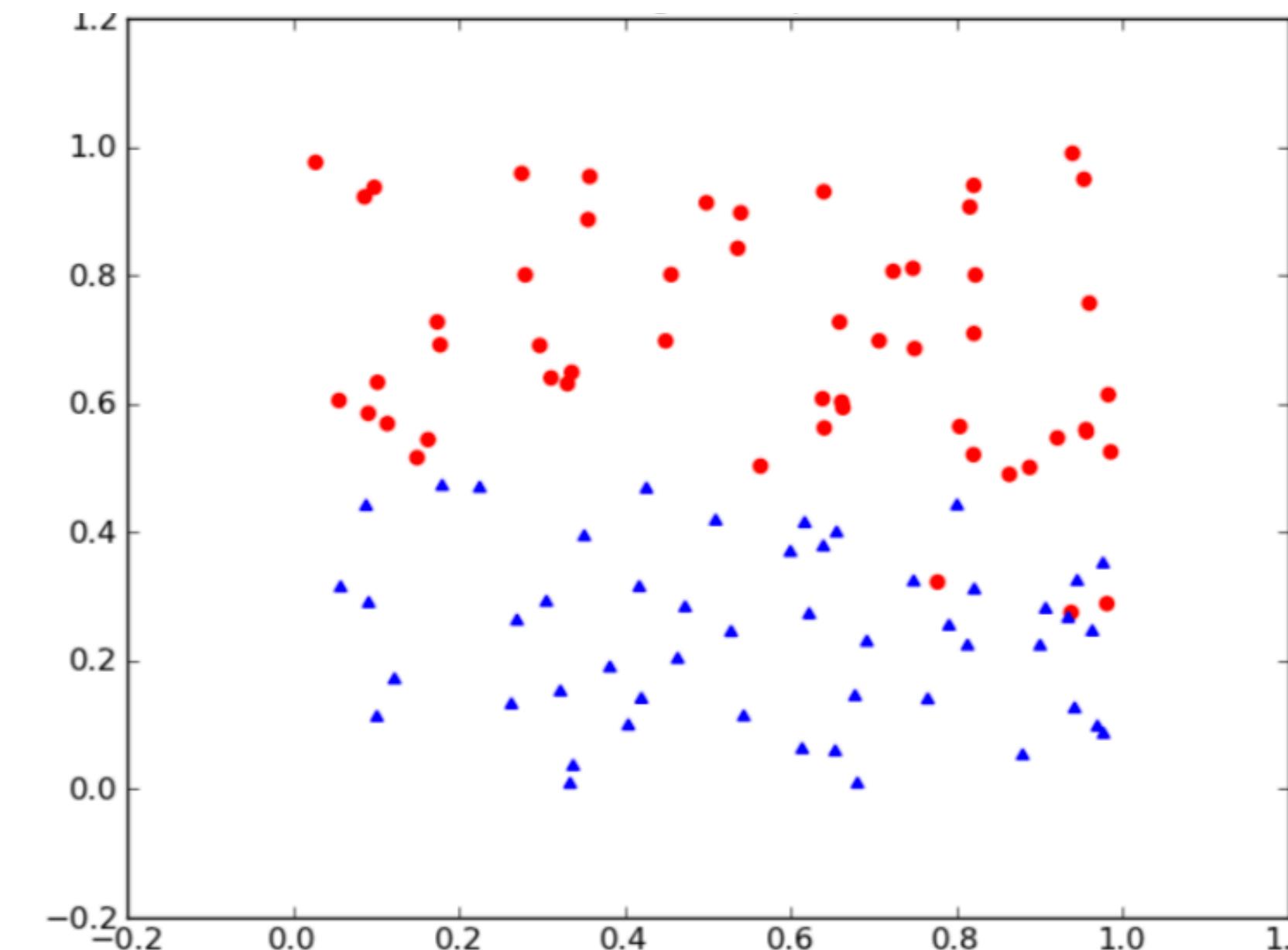
- ☺ trasparenti (i.e. facili da interpretare e capire)
- ☺ rappresentazione grafica
- ☺ funzionano in modo intuitivo come partizionamento ricorsivo del campione in regioni via via di più alta purezza
- ☺ poco sensibile a variabili deboli

- :(in generale un singolo albero di decisioni ha prestazioni limitate
- :(tende a soffrire di overtraining se non si controlla la crescita dell'albero (pruning techniques)
- :(instabile: piccole variazioni nel campione di training possono avere grandi effetti nella topologia del grafo

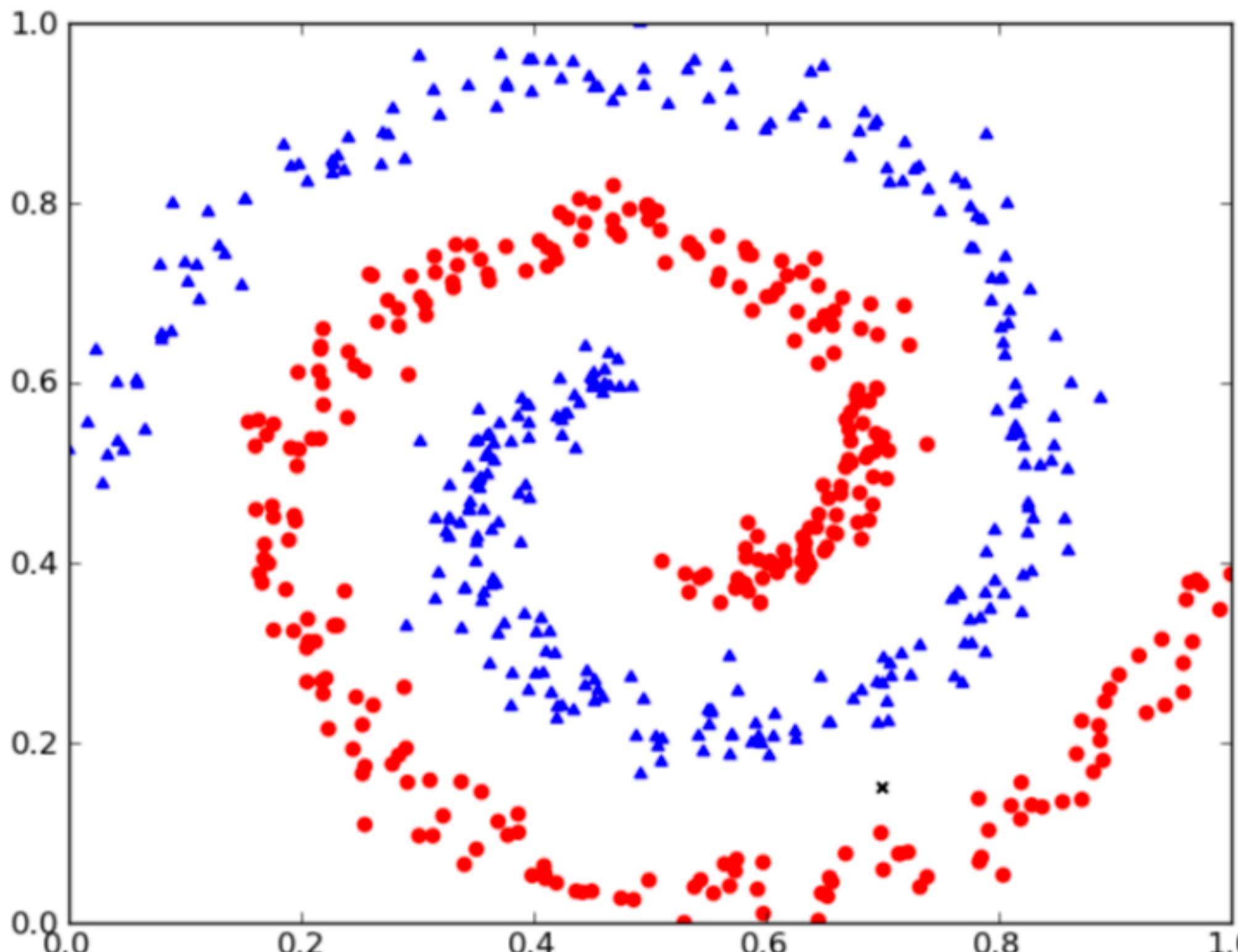


OVERFIT ...

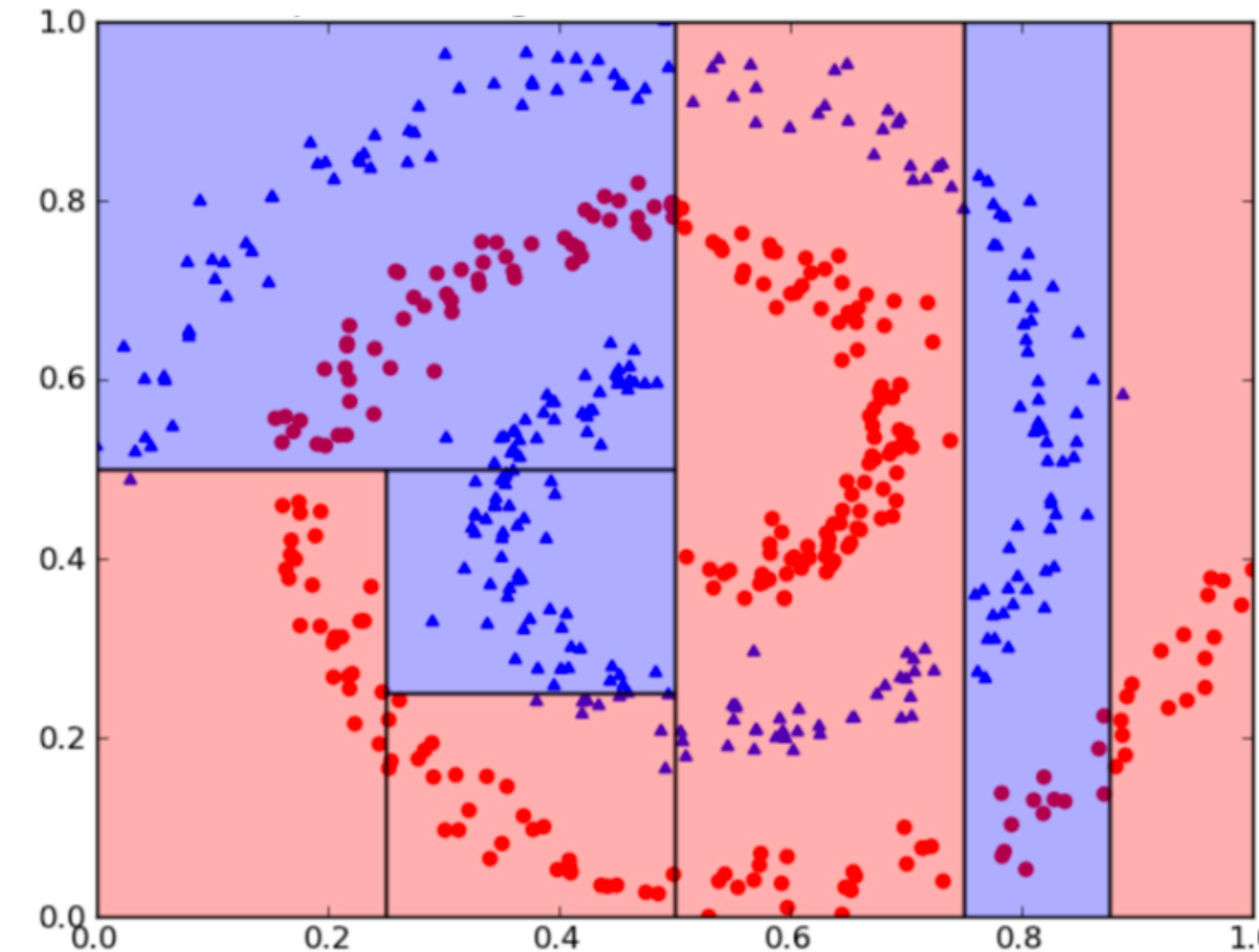
facile andare in overfit
se l'algoritmo non si
ferma presto ...



... VS PRESTAZIONI



potere di classificazione limitato se ci si ferma troppo presto ...



PRUNING (POTATURA)

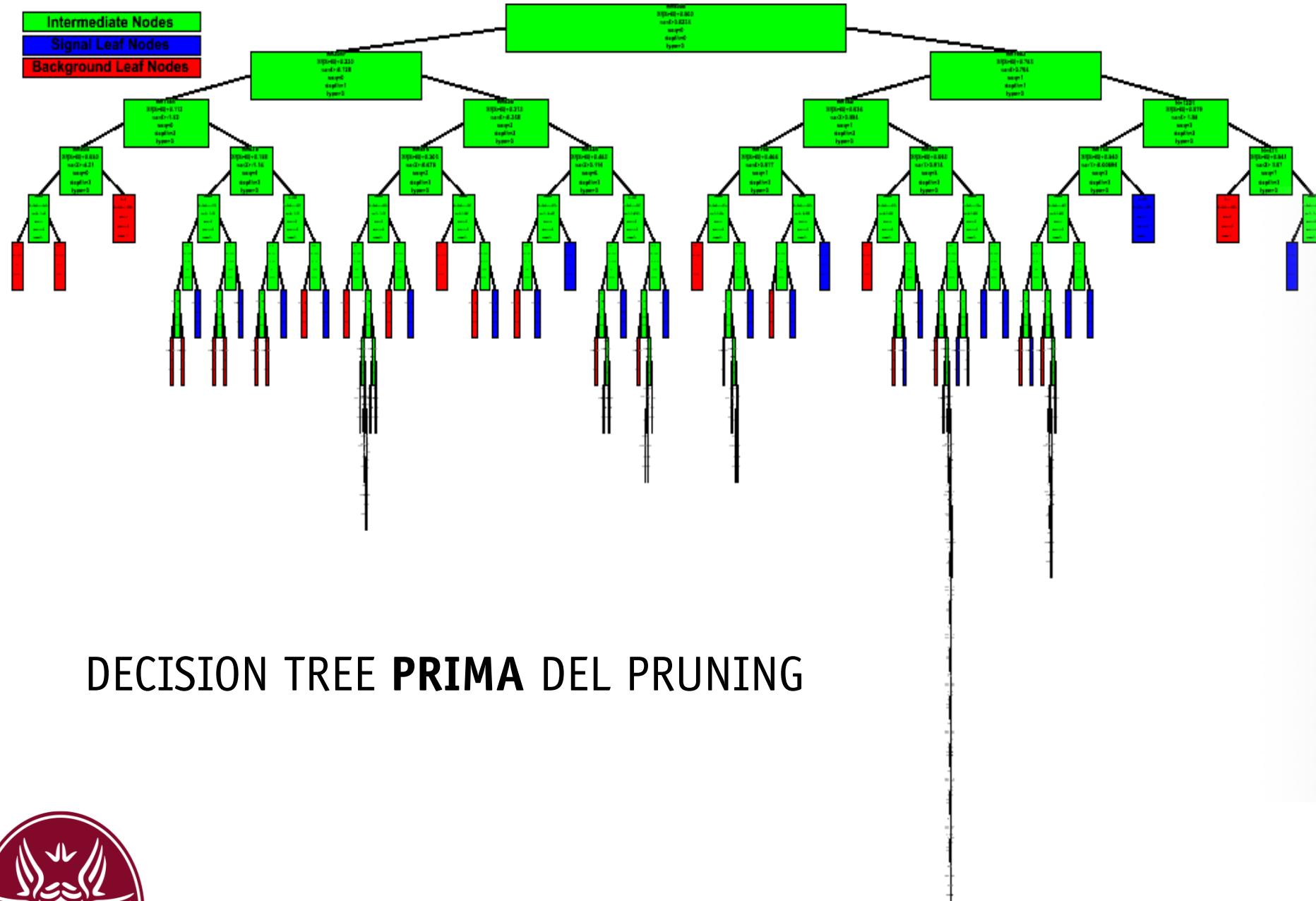
- ▶ per minimizzare la sensibilità dei decision tree alle fluttuazioni statistiche nei campioni di training vengono rimossi (potati) alcuni nodi dopo che l'albero è stato fatto crescere fino alla sua dimensione massima → minimizza la probabilità di overfit
- ▶ 2 algoritmi di potatura più usati:
 - ▶ **expected error:** vengono potati tutti i nodi per cui la stima dell'errore statistico nel nodo padre risulta inferiore all'errore statistico combinato dei nodi figli
 - ▶ **cost complexity:** vengono potati i nodi con il più piccolo valore della cost-complexity g :

$$g(t) = \frac{R(t) - R(T_t)}{f(T_t) - 1}$$

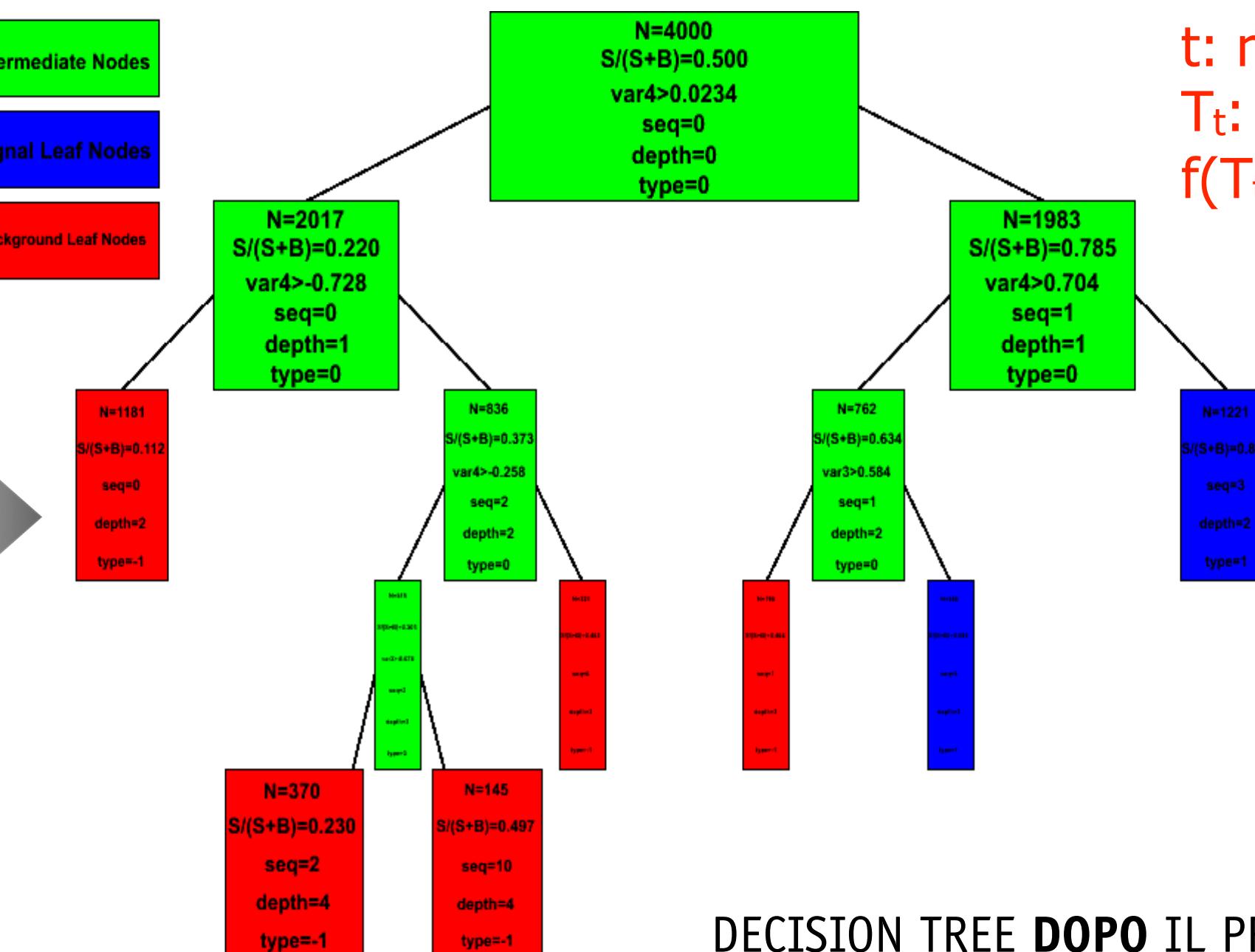
$R \equiv$ mis-classification rate
 t : nodo padre

T_t : tree che ha t come root

$f(T_t)-1$: numero di foglie da potare



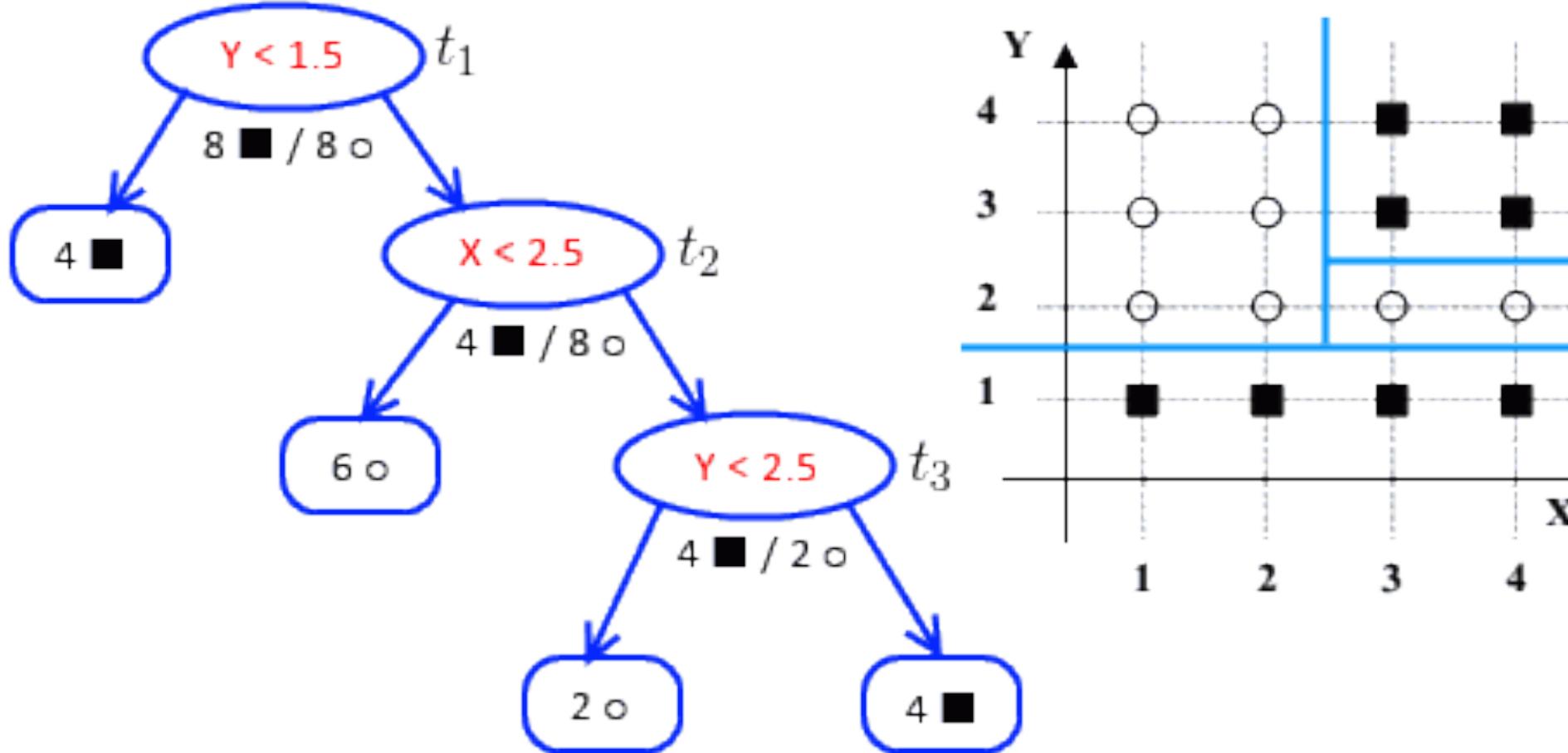
DECISION TREE PRIMA DEL PRUNING



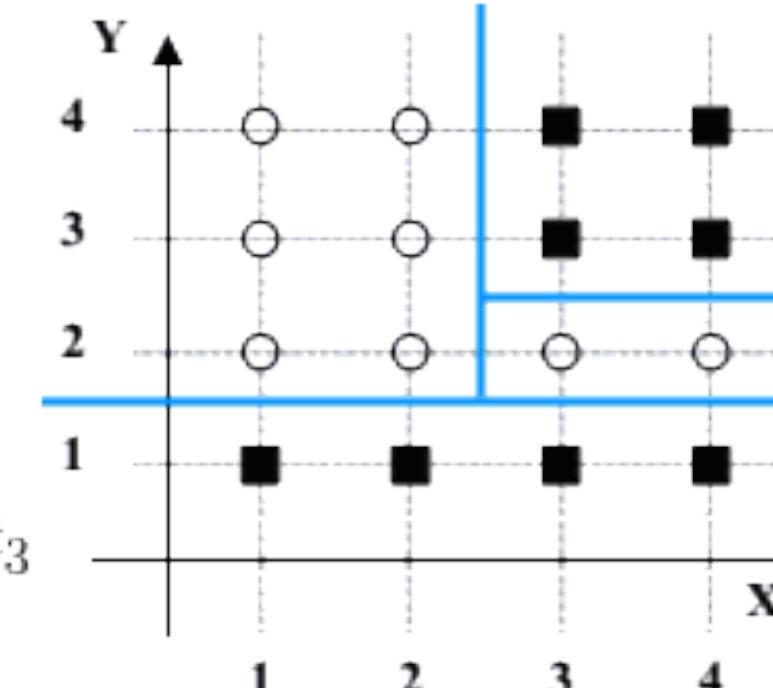
DECISION TREE DOPO IL PRUNING



ESEMPIO: COST COMPLEXITY PRUNING



tre nodi potabili: t_1 (root), t_2 , t_3



$$R(t_1) = r(t) \times p(t) = 8/16 \times 16/16 = 8/16$$

error rate nodo x frazione di dati analizzati

$$g(t_1) = (R(t_1) - R(T_{t1})) / \text{foglie-1} = (8/16 - 0) / (4-1) = 1/2 / 3 = 1/6$$

$$R(t_2) = 4/12 \times 12/16 = 1/4$$

$$R(T_{t2}) = 0$$

$$g(t_2) = (1/4 - 0) / (3-1) = 1/8$$

$$R(t_3) = 2/6 \times 6/16 = 1/8$$

$$R(T_{t3}) = 0$$

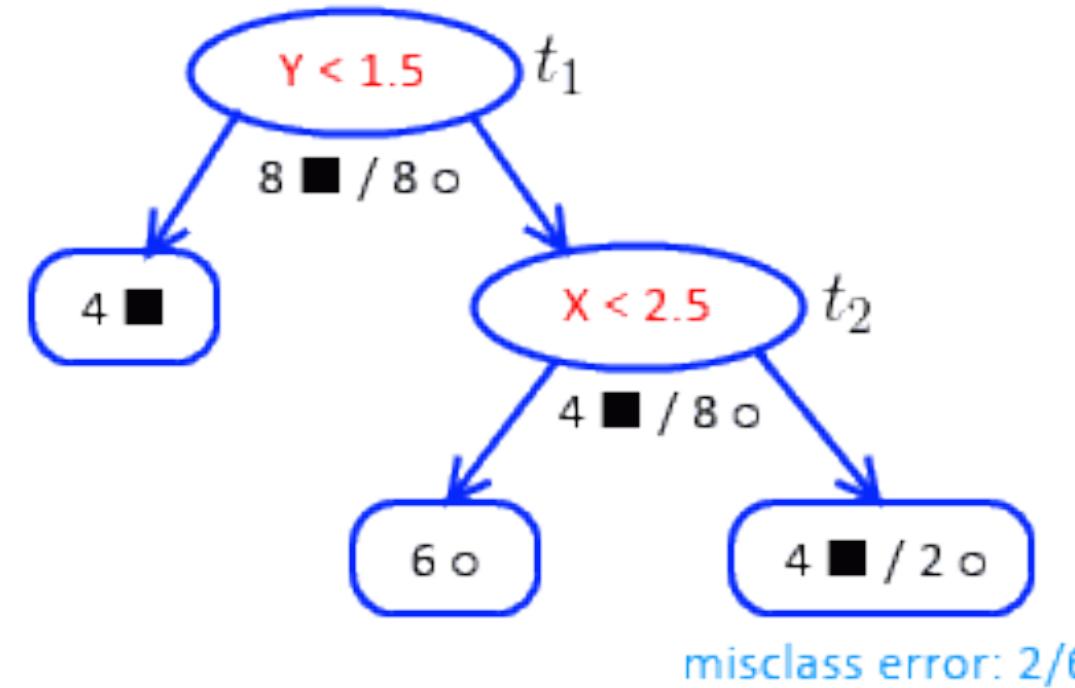
$$g(t_3) = (1/8 - 0) / (2-1) = 1/8$$

| | | |
|-------------|--|--------------------------------|
| t | $r(t) = n_1/(n_1+n_2)$ se $n_2 \geq n_1$ | $n_2/(n_1+n_2)$ se $n_1 > n_2$ |
| n_1 / n_2 | | |

$$R(T_{t1}) = 0$$

tutte le foglie sono pure i.e. non ci sono eventi mal classificati

potiamo il nodo con minimo $g(t)$ e nel caso di parità quello che taglia meno nodi: i.e. t_3



$$R(t_1) = 8/16 \times 16/16 = 1/2$$

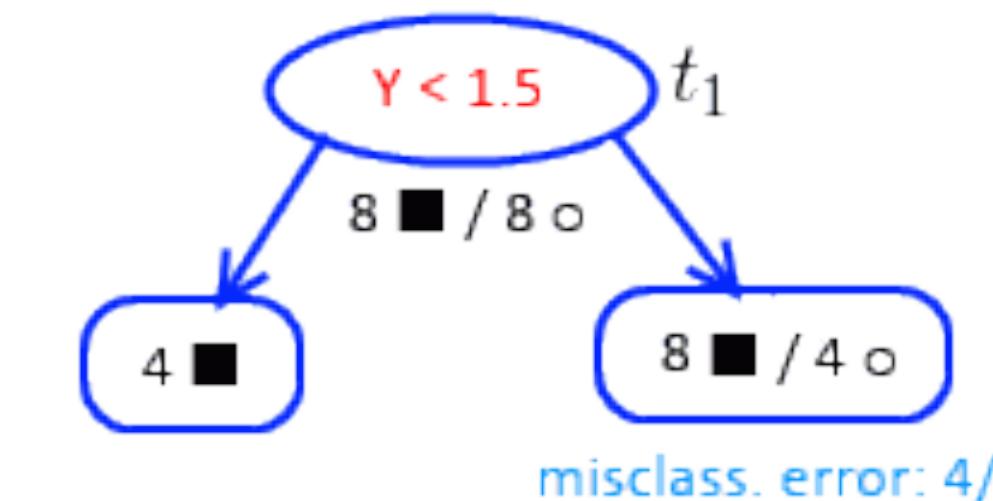
$$R(T_{t1}) = 2/16 = 1/8$$

$$g(t_1) = (1/2 - 1/8) / (3-1) = 3/16$$

$$R(t_2) = 4/12 \times 12/16 = 1/4$$

$$R(T_{t2}) = 2/16 = 1/8$$

$$g(t_2) = (1/4 - 1/8) / (2-1) = 1/8$$



$$R(t_1) = 8/16 \times 16/16 = 1/2$$

$$R(T_{t1}) = 4/16 = 1/4$$

$$g(t_1) = (1/2 - 1/4) / (2-1) = 1/4$$

ESEMPIO DECISION TREE CON SCIKIT-LEARN

[Link to scikit-learn example](#)

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

# Parameters
n_classes = 3
plot_colors = "ryb"
plot_step = 0.02

# Load data
iris = load_iris()          dataset IRIS usuale ...
```



```

for pairidx, pair in enumerate([[0, 1], [0, 2], [0, 3],
                           [1, 2], [1, 3], [2, 3]]):
    # We only take the two corresponding features
    X = iris.data[:, pair]
    y = iris.target

    # Train
    clf = DecisionTreeClassifier().fit(X, y)

    # Plot the decision boundary
    plt.subplot(2, 3, pairidx + 1)

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                          np.arange(y_min, y_max, plot_step))
    plt.tight_layout(h_pad=0.5, w_pad=0.5, pad=2.5)

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    cs = plt.contourf(xx, yy, Z, cmap=plt.cm.RdYlBu)

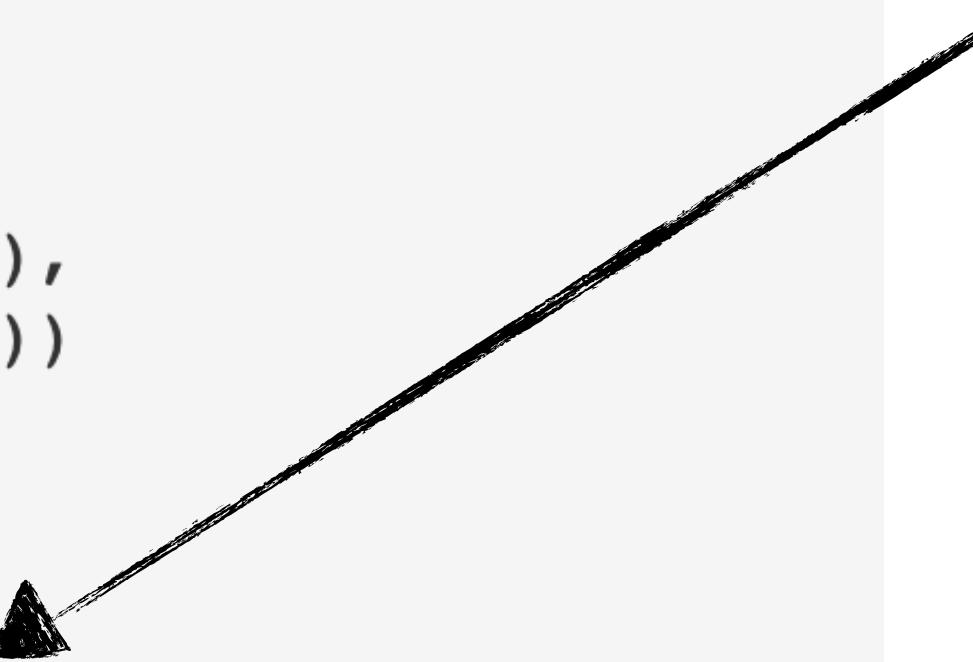
    plt.xlabel(iris.feature_names[pair[0]])
    plt.ylabel(iris.feature_names[pair[1]])

    # Plot the training points
    for i, color in zip(range(n_classes), plot_colors):
        idx = np.where(y == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=color, label=iris.target_names[i],
                    cmap=plt.cm.RdYlBu, edgecolor='black', s=15)

plt.suptitle("Decision surface of a decision tree using paired features")
plt.legend(loc='lower right', borderpad=0, handletextpad=0)
plt.axis("tight")
plt.show()

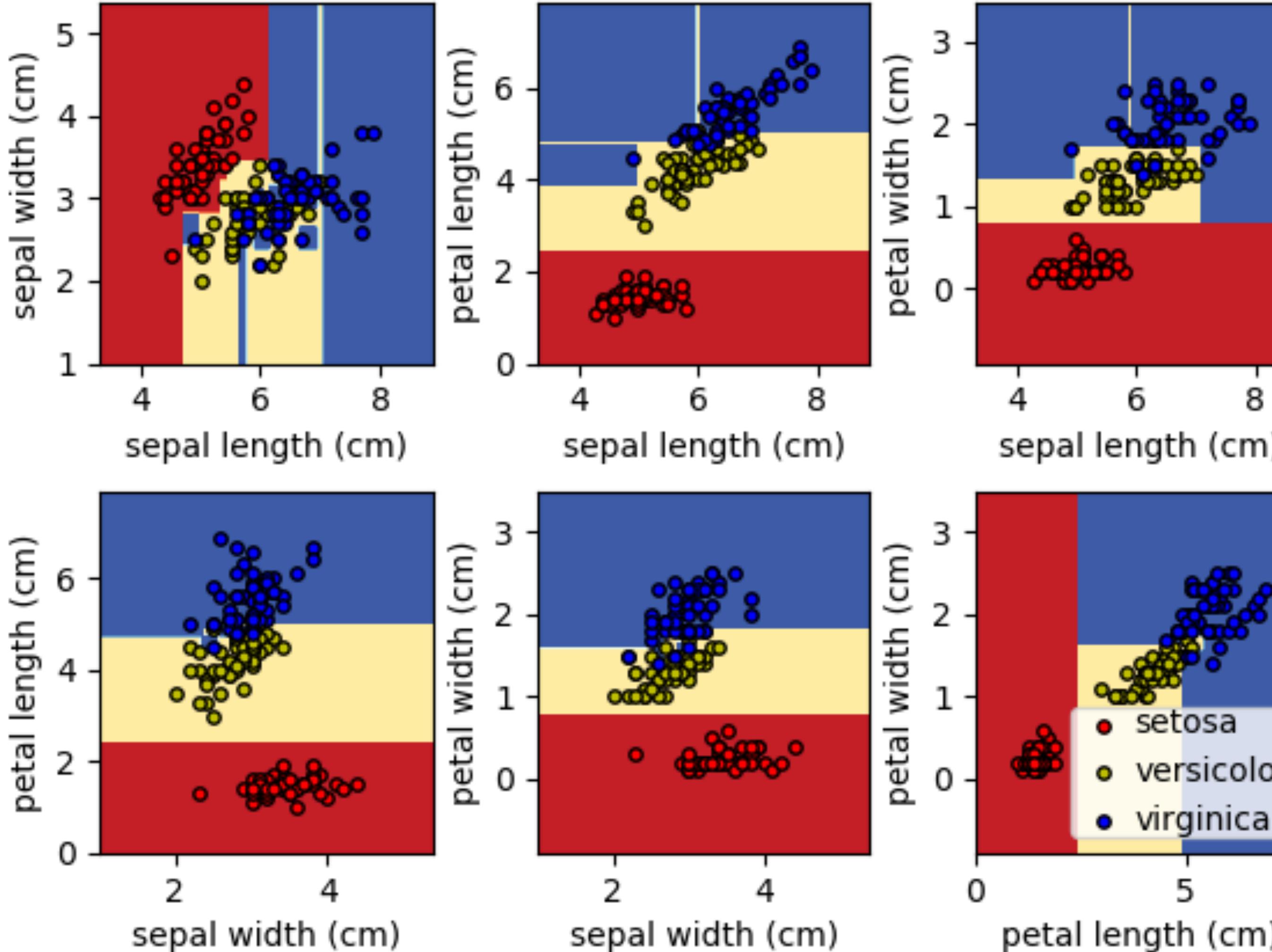
```

per ogni coppia di feature nel dataset iris
addestriamo un decision tree e ne grafichiamo le superfici di separazione



SUPERFICI DI SEPARAZIONE DEL DECISION TREE

Decision surface of a decision tree using paired features



ENSEMBLE DI CLASSIFICATORI

- Una delle idee più efficaci e largamente applicate nel moderno machine learning è legata all'uso di **ensemble method** che combinano le predizioni di più modelli deboli per migliorare le prestazioni predittive
- **Friedman-Popescu ansatz:** dato un set di tanti classificatori individualmente deboli, possiamo costruirne uno più performante mediando sull'ensamble di tali classificatori
- Spiegazione (discussione dettagliata: Louppe (2013) arXiv:1407.7502):
 - consideriamo un dataset $T\{\mathbf{x}_i, y_i\}$ $i=1\dots N$, ed assumiamo che i dati veri siano generati da un modello rumoroso del tipo:

$$y = f(\mathbf{x}) + \epsilon$$

con ϵ distribuita come $N(0, \sigma_\epsilon)$

- consideriamo una procedura statistica (tipo regressione dei minimi quadrati) che ci permetta di costruire un predittore $\hat{y} = g(\mathbf{x}; T)$ che fornisce una predizione del modello per un nuova feature \mathbf{x} una volta addestrato il modello usando il dataset T usando come loss function $L(T) = \sum(y_i - g(\mathbf{x}_i; T))^2$
- se ora immaginiamo di estrarre M dataset T_i della stessa dimensione di T e per ognuno costruiamo $g(\mathbf{x}; T_i)$, i predittori differiranno a causa degli effetti stocastici dovuti al noise, quindi possiamo interpretare le g come una variabile random di cui possiamo calcolare il valore atteso, si dimostra che l'errore atteso di generalizzazione di ogni modello è dato da:

$$E = Bias^2 + Var + Noise$$

misura la deviazione asintotica ($N \rightarrow \infty$) del nostro estimatore dal valore vero

tiene conto delle fluttuazioni addizionali dovute al fatto che $N \neq \infty$

tiene conto del rumore intrinseco nei dati dovuto a ϵ e che nessun estimatore statico può evitare



ENSEMBLE DI CLASSIFICATORI

- se ora consideriamo l'ensamble di classificatori:

$$g(\mathbf{x}_i) = \frac{1}{M} \sum_{m=1}^M g(\mathbf{x}_i; T_m)$$

- si dimostra che:

$$Var = \rho(\mathbf{x})\sigma_g^2 + \frac{1 - \rho(\mathbf{x})}{M}\sigma_g^2$$

ρ : grado correlazione tra i modelli

σ_g^2 : varianza del singolo modello randomizzato

- per cui usando grandi ensemble (M grande) possiamo ridurre il termine di Varianza e per ensemble completamente random con modelli totalmente correlati la riduzione diventa massimale
 - il termine di Bias atteso rimane invece lo stesso del singolo modello quando $\rho = 0$
 - in altre parole usando il predittore aggregato possiamo abbattere le fluttuazioni dovute alla dimensione finita del training set
-
- NOTA: nella pratica (BAGGING, BOOSTING) $\rho \neq 0$ e la procedura aumenta il bias ma il contributo è piccolo / trascurabile rispetto alla diminuzione della varianza



BAGGING E FORESTE RANDOM

- Bagging (Bootstrap AGGregation):
- è uno dei primi e più semplici metodi basati su ensemble:
 - basato su una procedura di bootstrap/re-sampling:
 - si parte dal sample T e si generano M nuovi sample campionando T (con replacement (i.e. eventi duplicati sono ammessi))
 - si addestrano M modelli di classificazione $g_i(\mathbf{x})$ ($i=1,\dots,M$) uno per ogni sample
 - classificatore finale da media dei predictor: $g(\mathbf{x}) = 1/M \sum g_i(\mathbf{x})$
 - se i $g_i(\mathbf{x})$ sono alberi di decisioni binarie, $g(\mathbf{x})$ viene chiamato una **Random Forest**
- NOTA: se ogni predittore $g_i(\mathbf{x})$ predice una classe di appartenenza $j \in [1,2,\dots,C]$ allora:

$$g(\mathbf{x}) = argmax_j \sum_{i=1}^M I[g_i(\mathbf{x}) = j]$$

$I[\dots] = 1$ se $g=j$; 0 altrimenti



TECNICA DEL BOOTSTRAP IN STATISTICA

- supponiamo di avere un set finito di n dati $D = \{x_1, \dots, x_n\}$ e di voler costruire l'intervallo di confidenza per un estimatore del nostro campione (per esempio la media del campione)

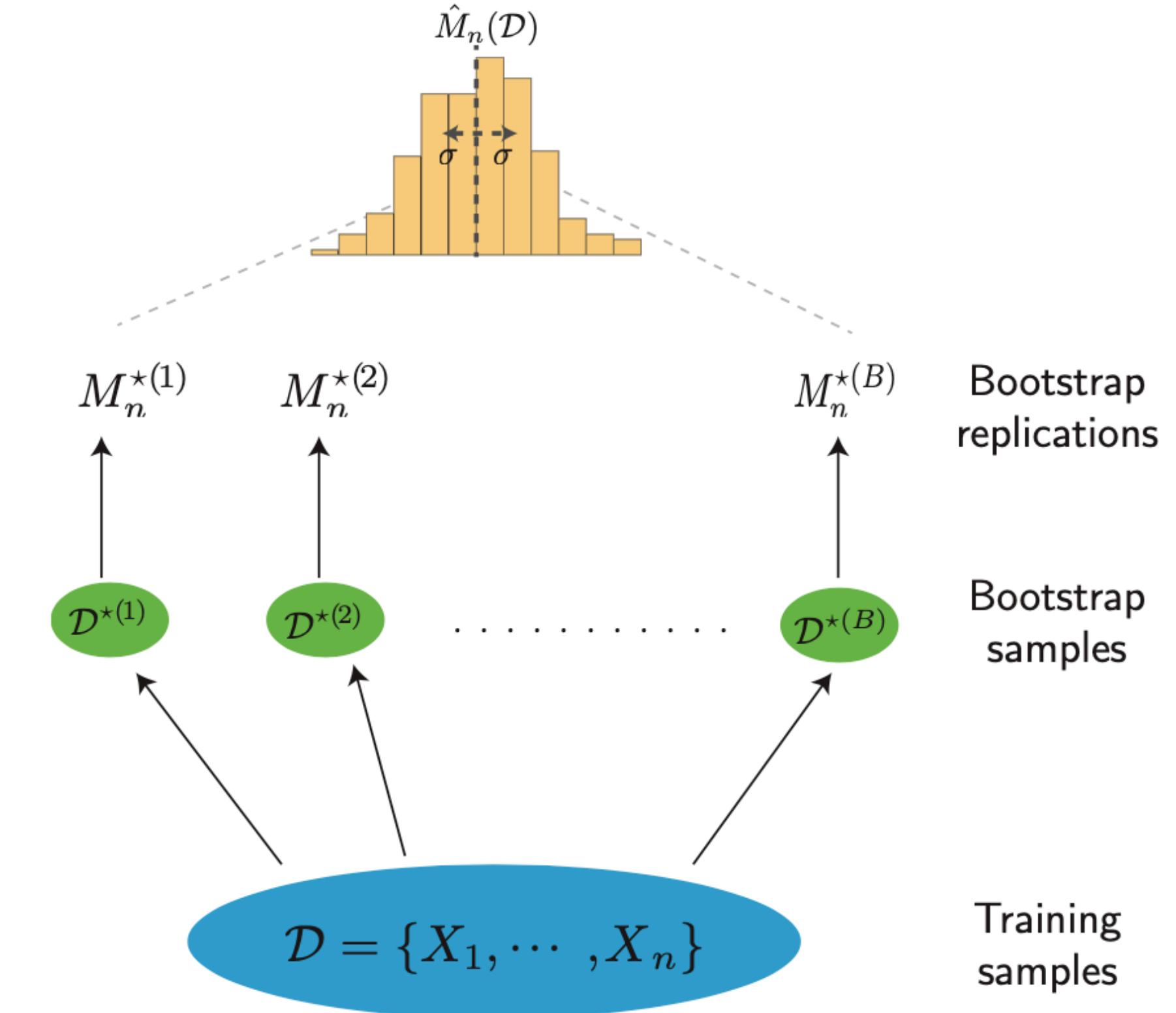
- per fare questo selezioniamo n punti con replacement da D e costruiamo $D^{*(1)} \leftarrow \text{bootstrap sample}$

- ripetiamo la procedura B volte: $D^{*(1)}, \dots, D^{*(B)}$

- consideriamo ora le mediane di bootstrap: $M_n^{*(k)} = \text{mediana}(D^{*(k)})$

- la varianza della distribuzione delle mediane di bootstrap è data da:

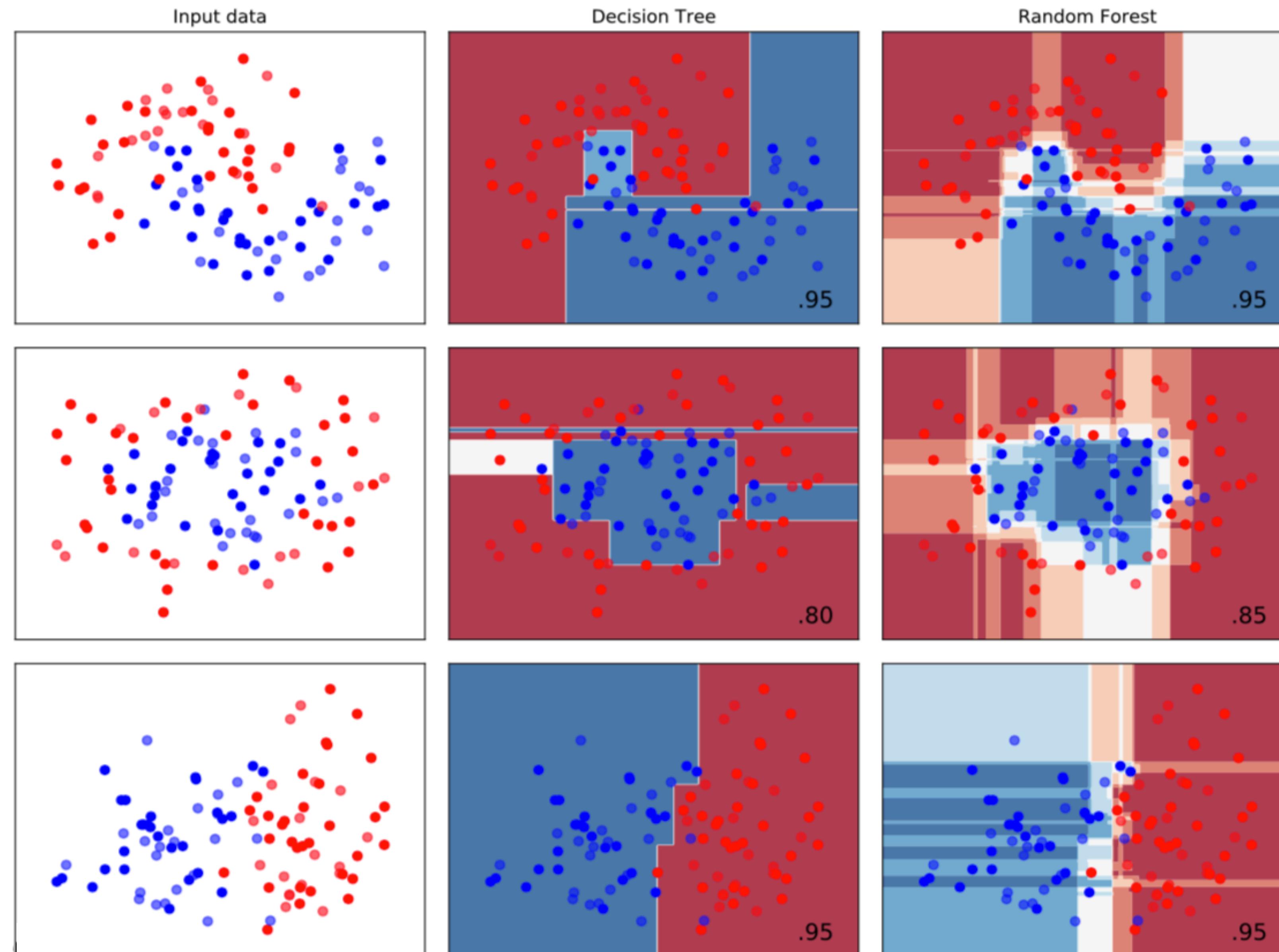
$$\tilde{Var}_B(M_n) = \frac{1}{B-1} \sum_{k=1}^B (M_n^{*(k)} - \langle M_n^* \rangle)^2 \quad \langle M_n^* \rangle = \frac{1}{B} \sum_{k=1}^B M_n^{*(k)}$$



- Si dimostra (Bickel, Freedman) che per $n \rightarrow \infty$ la distribuzione delle stime bootstrap è una gaussiana centrata in $M_n(D) = \text{mediana}(x_1, \dots, x_n)$ con deviazione standard proporzionale a $1/\sqrt{n}$



ESEMPIO RANDOM FOREST COSTRUITA VIA BAGGING



RANDOM FOREST PER FEATURE IMPORTANCE IN SCIKIT-LEARN

https://www.dropbox.com/s/3mkdtbpamszv7z/Feature_importance_ordering_DcisionForests_scikit-learn.ipynb?dl=0

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier

# Build a classification task using 3 informative features
X, y = make_classification(n_samples=1000,
                            n_features=10,
                            n_informative=3,
                            n_redundant=0,
                            n_repeated=0,
                            n_classes=2,
                            random_state=0,
                            shuffle=False)

# Build a forest and compute the feature importances
forest = RandomForestClassifier(n_estimators=250,
                                 random_state=0)

forest.fit(X, y)
importances = forest.feature_importances_
std = np.std([tree.feature_importances_ for tree in forest.estimators_],
             axis=0)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(X.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))
```

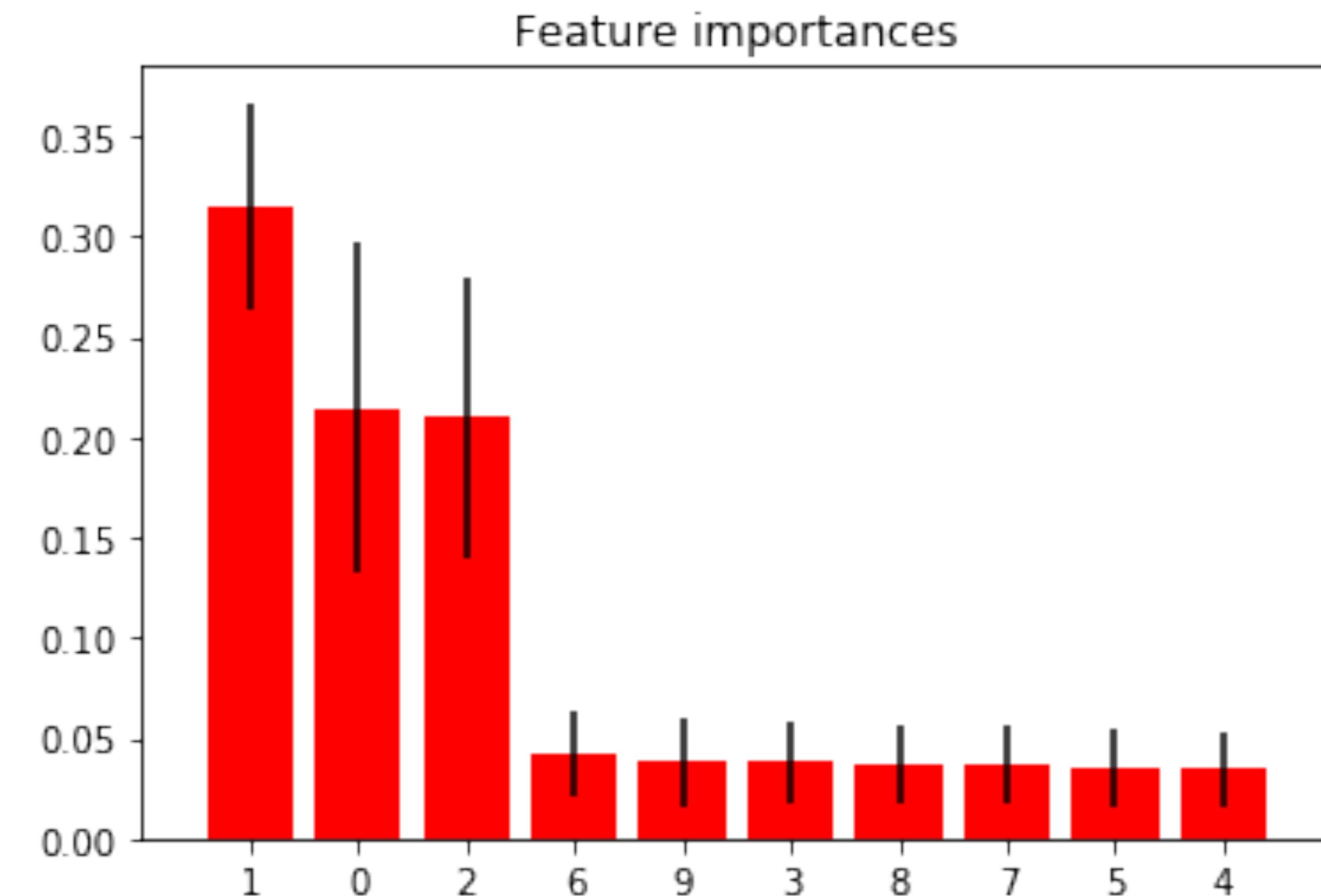
costruisce un toy sample con 10 features
(di cui solo 3 informative) e usa una
foresta di decisioni per ordinare le
feature in termini di importanza

feature_importance_: sfrutta l'idea che le
feature che vengono usate per prime nello
sviluppo del tree sono quelle più informative
e contribuiscono alla predizione finale per
una frazione più grande del campione di
training

```
# Plot the feature importances of the forest
plt.figure()
plt.title("Feature importances")
plt.bar(range(X.shape[1]), importances[indices],
        color="r", yerr=std[indices], align="center")
plt.xticks(range(X.shape[1]), indices)
plt.xlim([-1, X.shape[1]])
plt.show()
```

Feature ranking:

1. **feature 1** (0.315051)
2. **feature 0** (0.214699)
3. **feature 2** (0.209859)
4. **feature 6** (0.041458)
5. **feature 9** (0.038024)
6. **feature 3** (0.037885)
7. **feature 8** (0.037067)
8. **feature 7** (0.036401)
9. **feature 5** (0.035074)
10. **feature 4** (0.034483)



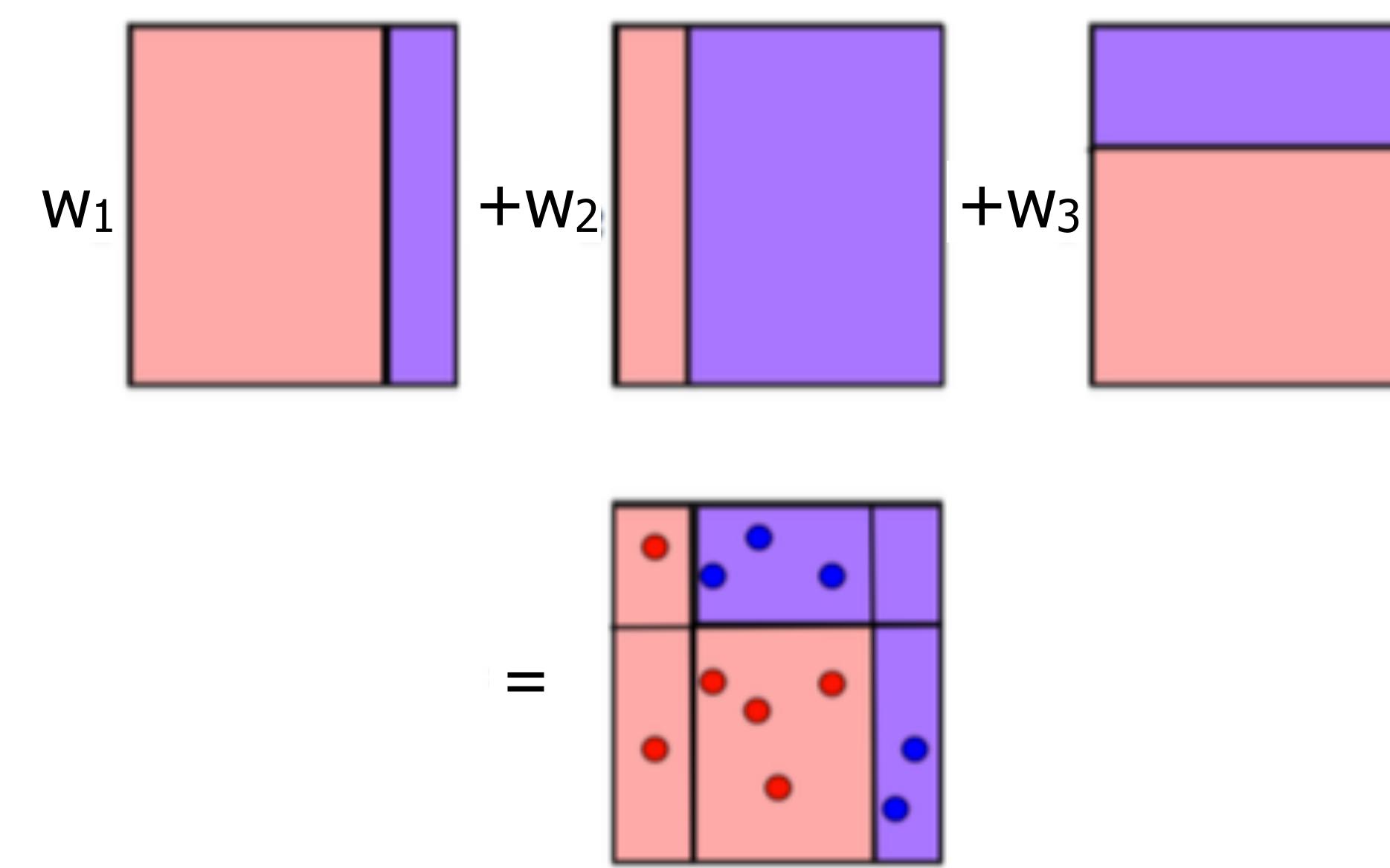
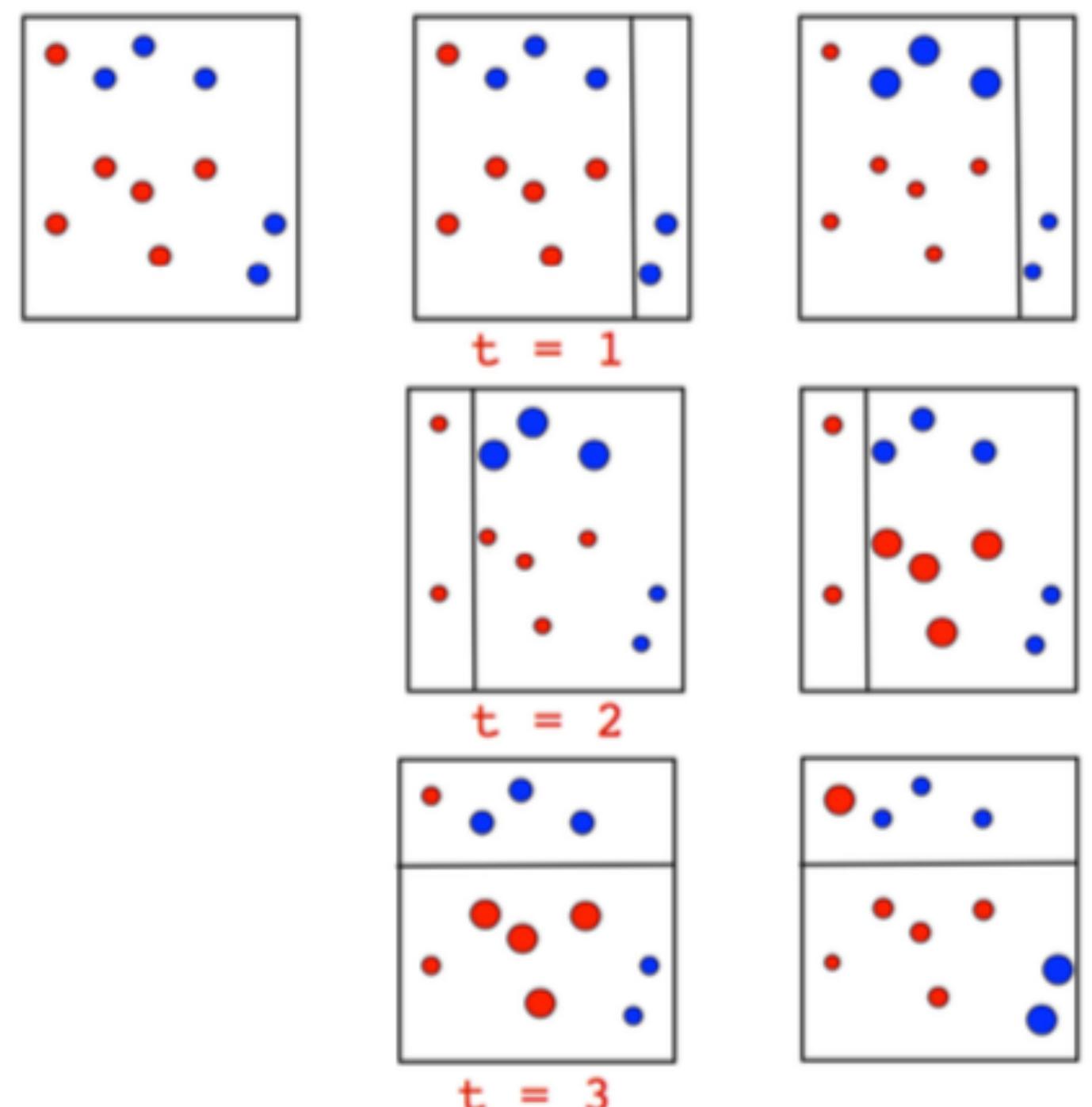
BOOSTING

- Altra idea per migliorare le prestazioni di un ensemble di classificatori (Freund, Shapire, Ho)
 - l'approccio è simile a quello del Bagging, ma:
 - nel training dell'i-esimo predittore: $g_i(x)$ si sfruttano le conoscenze sulle prestazioni ottenute con i precedenti $N-i$ predittori
- vari algoritmi di boosting disponibili:
 - Adaptive Boost (ADA-Boost)
 - Gradient Boost (GBoost)
 - Extreme Gradient Boost (XGBoost): i.e. gradient boost + regolarizzazione + ottimizzazione per GPU ...



ADAPTIVE (ADA)-BOOST

- ▶ lo stesso modello (per esempio un decision tree) viene addestrato più volte, ogni volta usando un campione di eventi di training ri-pesato (boosted) con pesi aumentati per gli eventi mal-classificati dai precedenti decision tree
- ▶ i.e. gli eventi che vengono classificati in modo erroneo durante la crescita di un tree, ricevono un peso maggiore nella crescita (training) del tree successivo
- ▶ il processo continua fino a quando si è raggiunto un valore predefinito per il massimo numero di alberi della foresta (tipicamente: $O(10^2-10^3)$)
- ▶ alla fine l'evento di test viene classificato sulla base della risposta data dalla media pesata dei tree della foresta



ALGORITMO ADA-BOOST IN DETTAGLIO ...

- ▶ sia dato il training set $T\{\mathbf{x}_i, y_i\} \ i=1, \dots, N$
- ▶ scegliamo un peso iniziale uguale per ogni evento del campione pari a: $w_{t=1}(\mathbf{x}_i) = 1/N \ \forall i$
- ▶ ripetiamo per $t=1, \dots, t_{\max}$ (max numero di alberi, i.e. criterio di stop)

1. costruiamo l'albero con il campione con eventi pesati: $g_t(\mathbf{x})$

2. calcoliamo la rate di errore sul campione pesato ϵ_t :

$$\epsilon_t = \sum_{i=1}^N w_t(\mathbf{x}_i) I[g_t(\mathbf{x}_i) \neq y_i]$$

3. sia: $\alpha_t = 1/2 \ln[(1-\epsilon_t)/\epsilon_t] \rightarrow +\infty, -\infty$ per $\epsilon \rightarrow 0, 1$

$I[\dots] = 1$ se $g \neq y$; 0 altrimenti

4. aggiorniamo i pesi del training set all'iterazione t :

$$w_{t+1}(\mathbf{x}_i) = w_t(\mathbf{x}_i) \frac{e^{[-\alpha_t y_i g_t(\mathbf{x}_i)]}}{Z_t} \quad i = 1, \dots, N$$

con:

$$Z_t = \sum_{i=1}^N w_t(\mathbf{x}_i) e^{[-\alpha_t y_i g_t(\mathbf{x}_i)]}$$

garantisce che l'insieme dei pesi sommi a 1



BOOSTED DECISION TREE (BDT)

- ▶ l'output dell'algoritmo sarà:

$$y_{BDT}(\mathbf{x}) = \text{sign} \left[\sum_{t=1}^{t_{max}} \alpha_t g_t(\mathbf{x}) \right]$$

$$y_{BDT}(\mathbf{x}) = \sum_{t=1}^{t_{max}} \alpha_t g_t(\mathbf{x})$$

Two-class BDT binary output:

| | |
|----------------|------------|
| $y_{BDT} = -1$ | → classe B |
| $y_{BDT} = +1$ | → classe A |

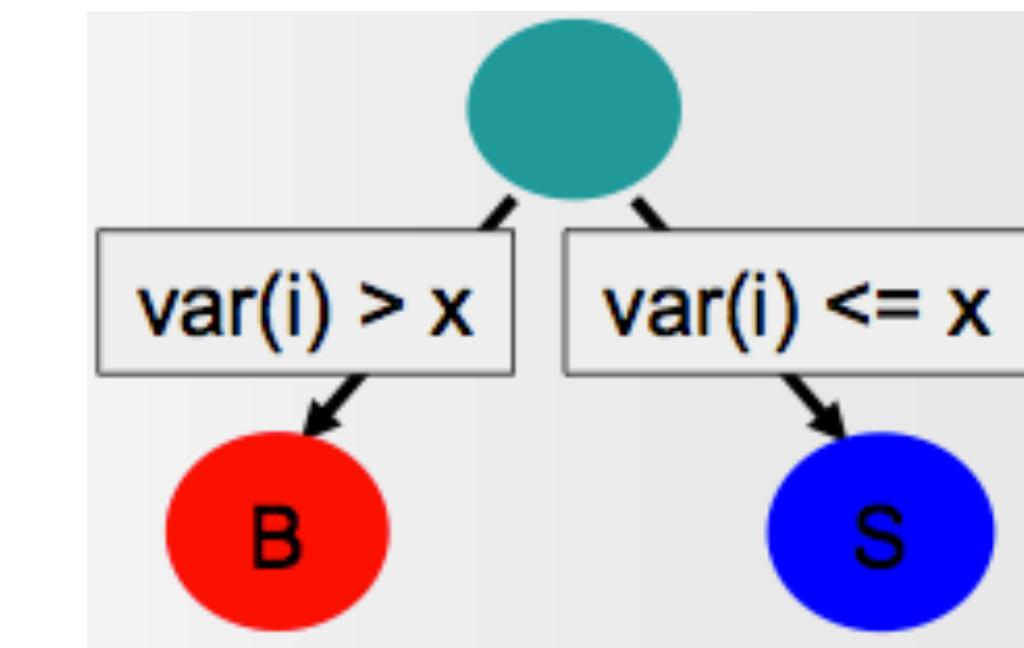
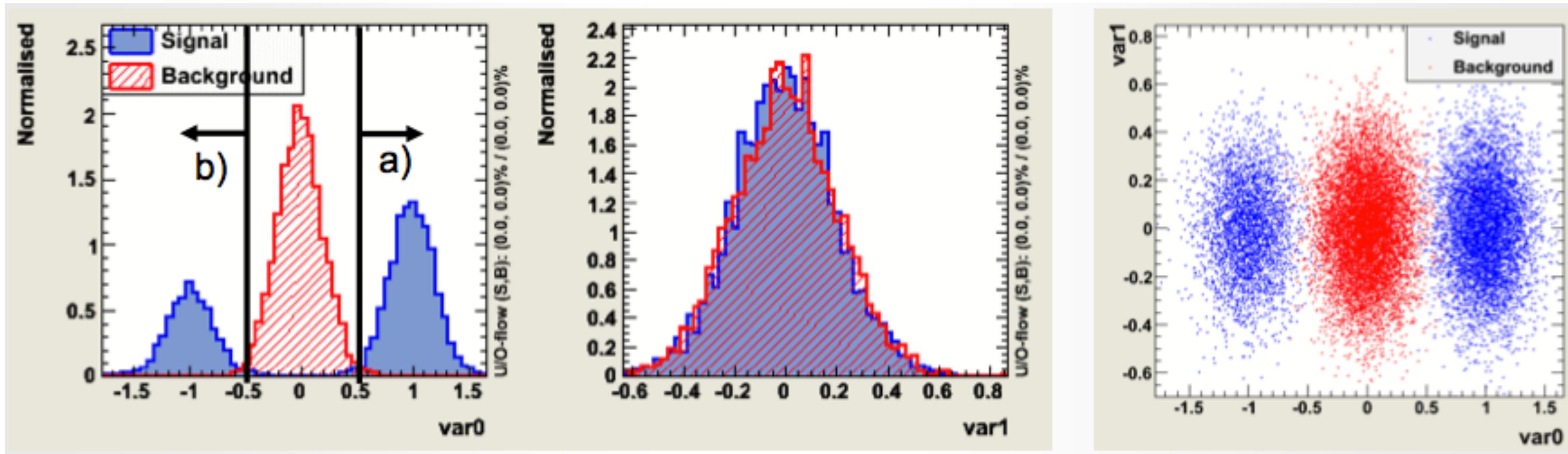
Two-class BDT output continuo:

| | |
|-------------------|------------|
| piccolo y_{BDT} | → classe B |
| grande y_{BDT} | → classe A |

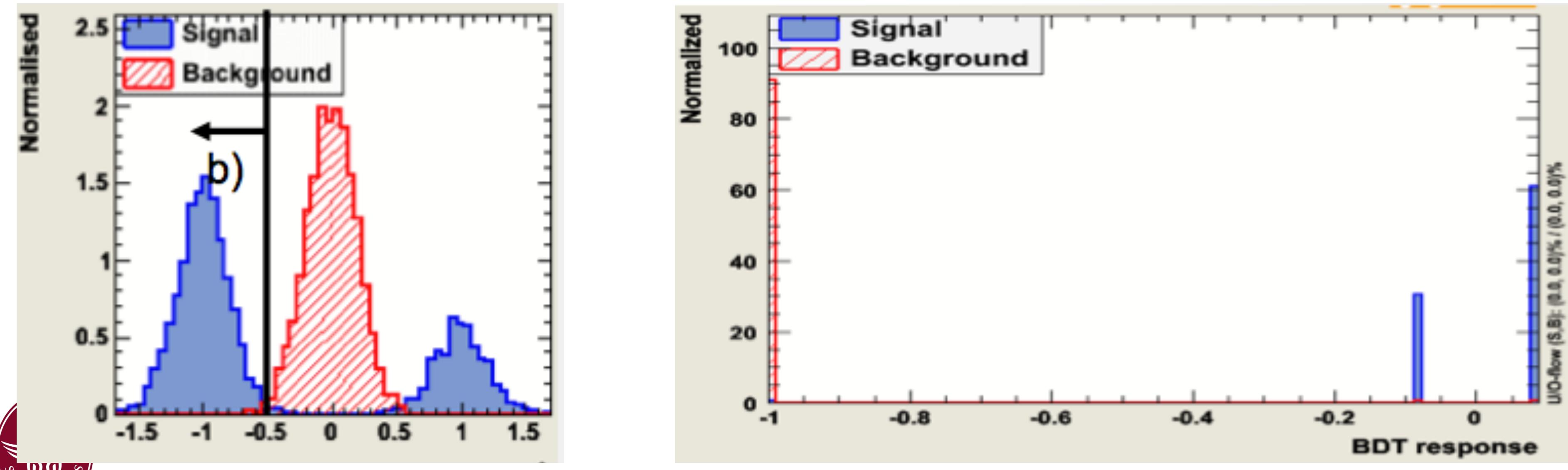
- ▶ vantaggi: tipicamente i BDT sono i classificatori più performanti “out-of-the-box”
- ▶ svantaggi: tanti tree e l'operazione di boosting implicano forte perdite di generalità con problemi di overtraining → richiede validazione e ottimizzazione accurata dei parametri



ESEMPIO TOY: ADA-BOOST “LA LAVORO”



il training di un singolo albero trova il taglio: $\text{var0} > 0.5$ (a)
nella costruzione del successivo tree il campione viene ri-pesato e il singolo albero trova il taglio: $\text{var0} < 0.5$ (b)



BDT:
combinazione dei
2 trees:

ESEMPIO CLASSIFICATORE ADA-BDT CON SCIKIT-LEARN

scikit-learn.org ADA-BDT example link

```
# Author: Noel Dawe <noel.dawe@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_gaussian_quantiles

# Construct dataset
X1, y1 = make_gaussian_quantiles(cov=2.,
                                   n_samples=200, n_features=2,
                                   n_classes=2, random_state=1)
X2, y2 = make_gaussian_quantiles(mean=(3, 3), cov=1.5,
                                   n_samples=300, n_features=2,
                                   n_classes=2, random_state=1)

X = np.concatenate((X1, X2))
y = np.concatenate((y1, -y2 + 1))
```

costruisco un dataset toy non
linearmemente separabile: due
cluster basati su gausiane
multidimensionali

classi separate da sfere
concentriche
multidimensionali con
~stesso numero di eventi
in ciascuna classe

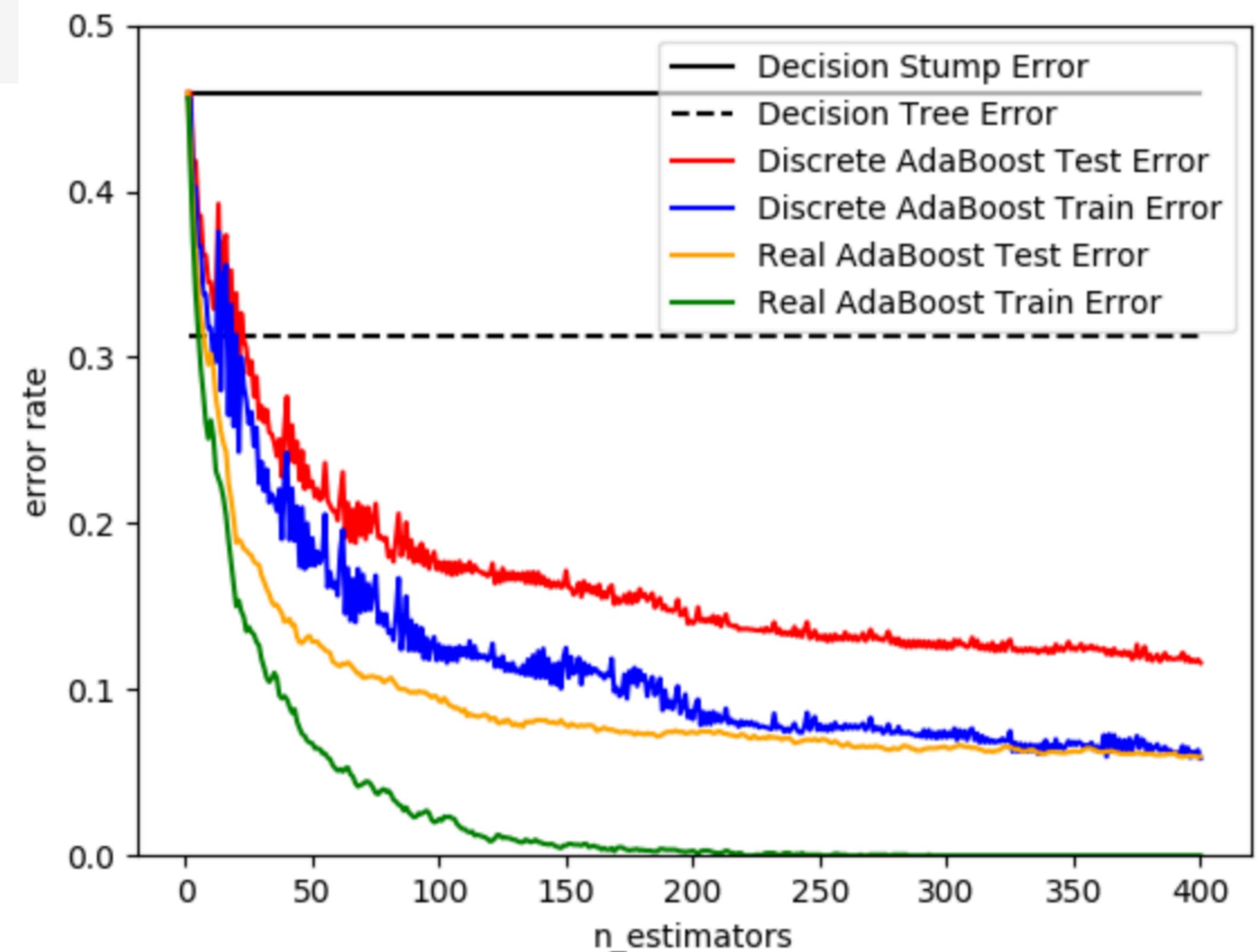


```
# Create and fit an AdaBoosted decision tree
bdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
                         algorithm="SAMME",
                         n_estimators=200)
```

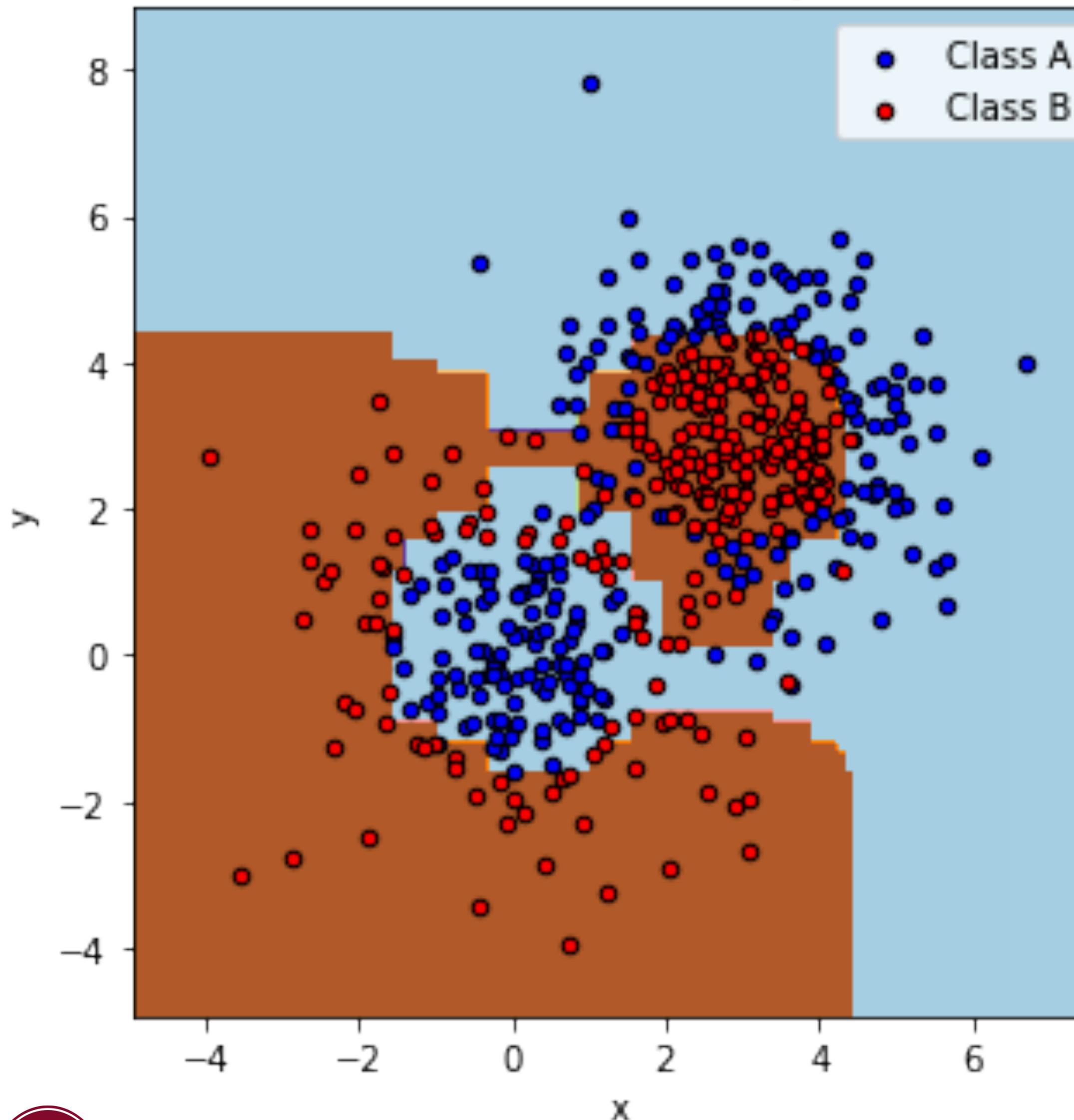
```
bdt.fit(X, y)
```

SAMME: discrete AdaBoost adapts based on errors in predicted class labels

SAMME.R: real AdaBoost adapt bases on the predicted class probabilities (faster)



Decision Boundary



```
plot_colors = "br"
plot_step = 0.02
class_names = "AB"

plt.figure(figsize=(10, 5))

# Plot the decision boundaries
plt.subplot(121)
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                      np.arange(y_min, y_max, plot_step))

z = bdt.predict(np.c_[xx.ravel(), yy.ravel()])
z = z.reshape(xx.shape)
cs = plt.contourf(xx, yy, z, cmap=plt.cm.Paired)
plt.axis("tight")

# Plot the training points
for i, n, c in zip(range(2), class_names, plot_colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1],
                c=c, cmap=plt.cm.Paired,
                s=20, edgecolor='k',
                label="Class %s" % n)

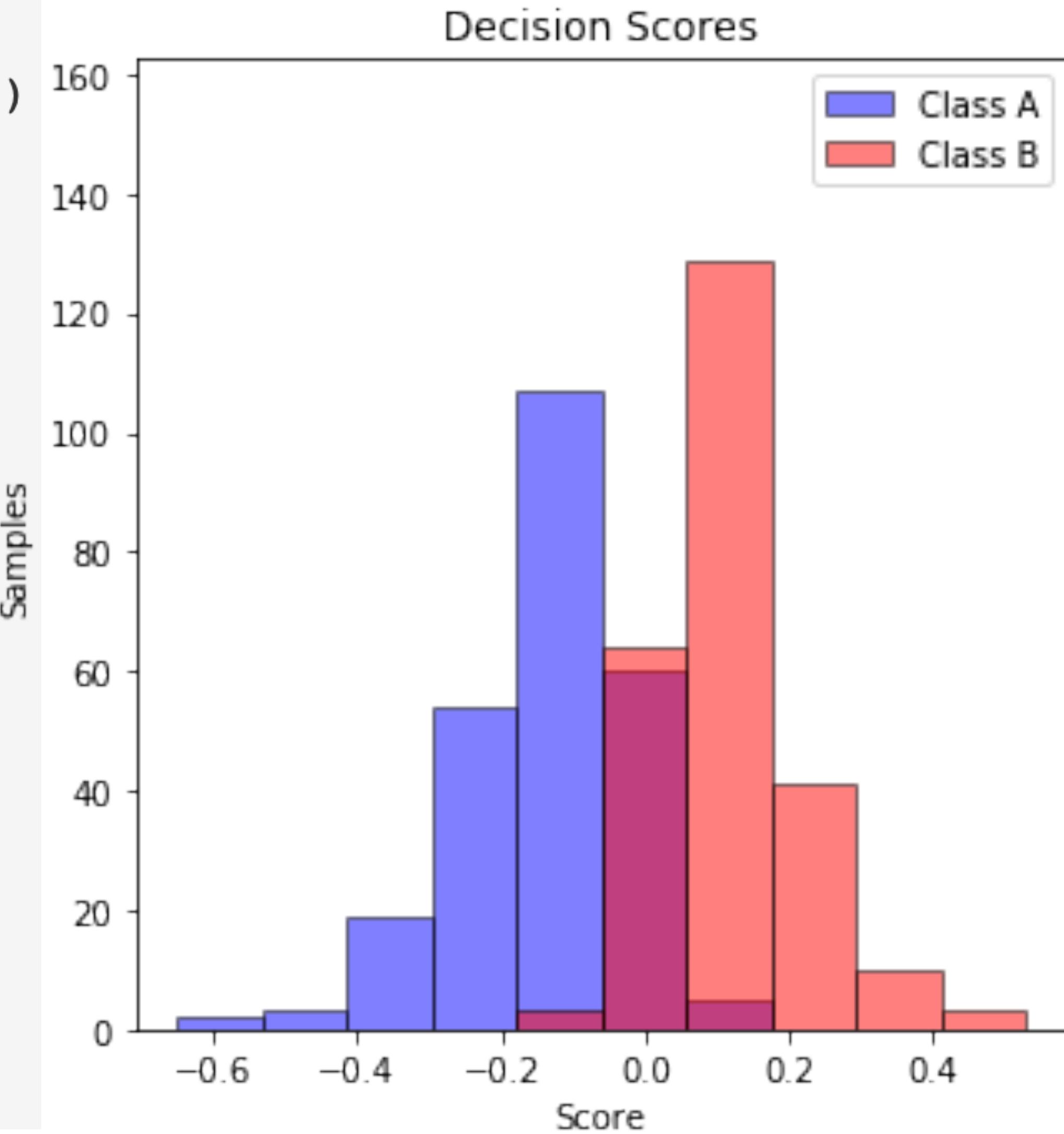
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.legend(loc='upper right')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Decision Boundary')
```

```

# Plot the two-class decision scores
twoclass_output = bdt.decision_function(X)
plot_range = (twoclass_output.min(), twoclass_output.max())
plt.subplot(122)
for i, n, c in zip(range(2), class_names, plot_colors):
    plt.hist(twoclass_output[y == i],
             bins=10,
             range=plot_range,
             facecolor=c,
             label='Class %s' % n,
             alpha=.5,
             edgecolor='k')
x1, x2, y1, y2 = plt.axis()
plt.axis((x1, x2, y1, y2 * 1.2))
plt.legend(loc='upper right')
plt.ylabel('Samples')
plt.xlabel('Score')
plt.title('Decision Scores')

plt.tight_layout()
plt.subplots_adjust(wspace=0.35)
plt.show()

```



ESEMPIO DECISION TREE AND ADA-BDT PER REGRESSIONE CON SCIKIT-LEARN

https://scikit-learn.org/stable/auto_examples/ensemble/plot_adaboost_regression.html

```
# Author: Noel Dawe <noel.dawe@gmail.com>
#
# License: BSD 3 clause

# importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor

# Create the dataset
rng = np.random.RandomState(1)
X = np.linspace(0, 6, 100)[:, np.newaxis]
y = np.sin(X).ravel() + np.sin(6 * X).ravel() + rng.normal(0, 0.1, X.shape[0])

# Fit regression model
regr_1 = DecisionTreeRegressor(max_depth=4)

regr_2 = AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
                          n_estimators=300, random_state=rng)

regr_1.fit(X, y)
regr_2.fit(X, y)
```

Confronto di un regressore basato su decision tree regressor vs un regressore ADA-BDT su un campione 1D da distribuzione sinusoidale. rumore gaussiano

```

# Predict
y_1 = regr_1.predict(X)
y_2 = regr_2.predict(X)

# Plot the results
plt.figure()
plt.scatter(X, y, c="k", label="training samples")
plt.plot(X, y_1, c="g", label="n_estimators=1", linewidth=2)
plt.plot(X, y_2, c="r", label="n_estimators=300", linewidth=2)
plt.xlabel("data")
plt.ylabel("target")
plt.title("Boosted Decision Tree Regression")
plt.legend()
plt.show()

```

