

## MACHINE LEARNING AVANZATO DA ZERO

ANTONIO DI CECCO - SCHOOL OF AI

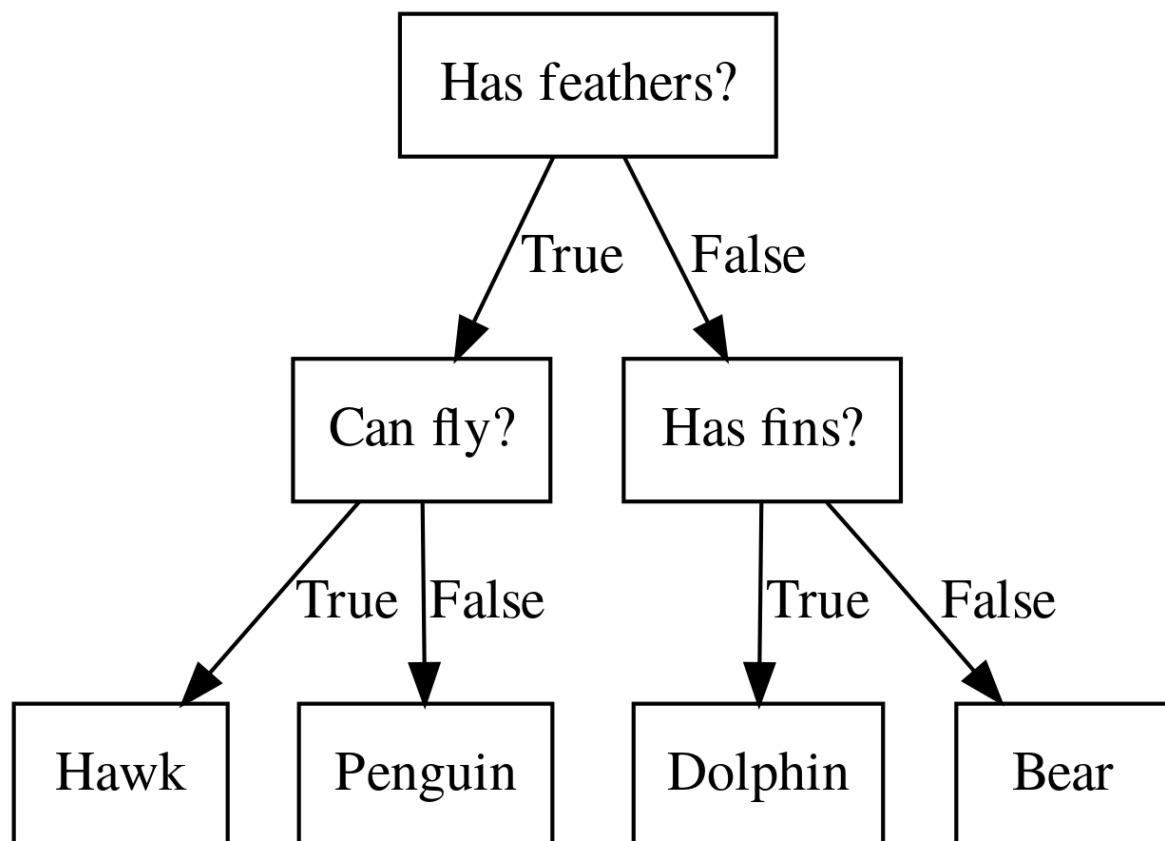
# Decision Tree, Random Forest ed Ensemble di modelli

*Jeremy Howard Approves<sup>(tm)</sup>*

## Decision Tree per la classificazione

"Akinator"

Idea: una serie di domande binarie

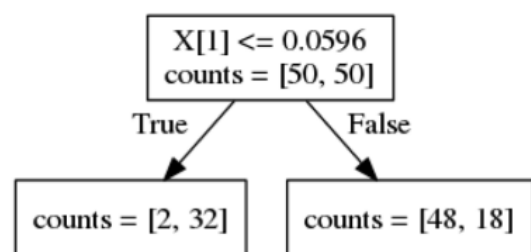
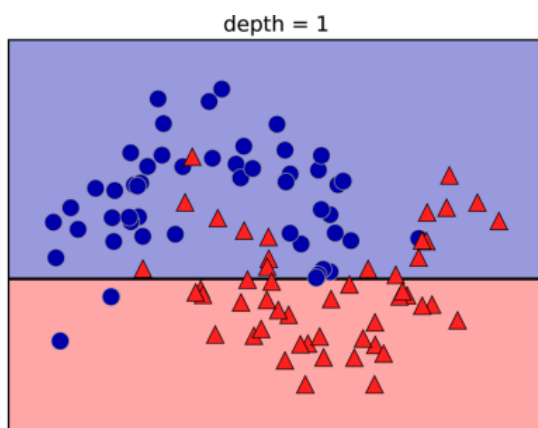


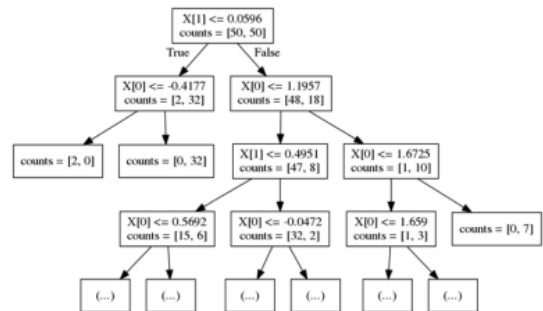
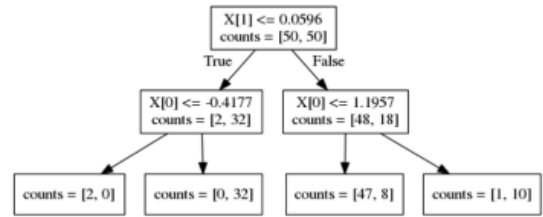
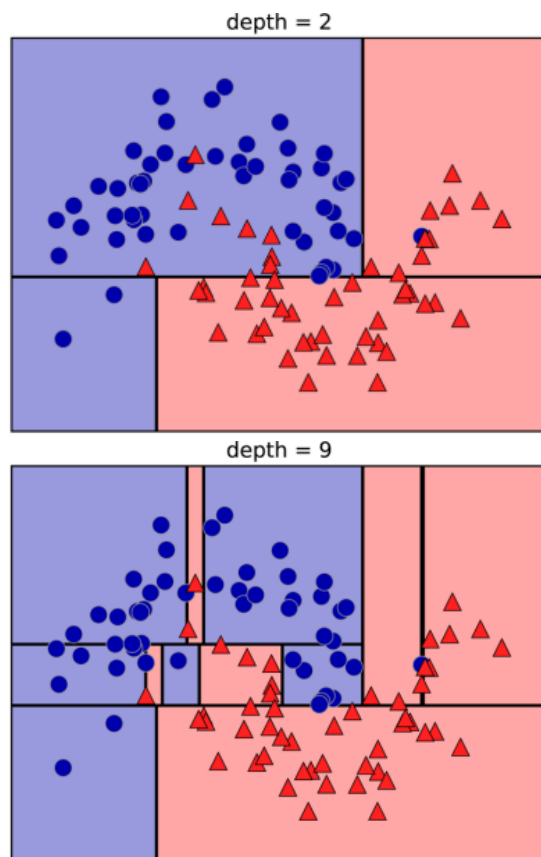
## Costruire gli alberi

---

### Feature continue:

- le "domande" sono i treshold su ogni feature
- Minimizzare l'impurità
- per evitare un mostruoso costo computazionale la ricerca dello split si fa in modo *greedy*





## Criteri (per la classificazione)

Indici di impurità (da minimizzare ad ogni split)

- Indice di Gini:

$$H_{\text{gini}}(X_m) = \sum_{k \in \mathcal{Y}} p_{mk} (1 - p_{mk})$$

- Cross-Entropy:

$$H_{\text{CE}}(X_m) = - \sum_{k \in \mathcal{Y}} p_{mk} \log(p_{mk})$$

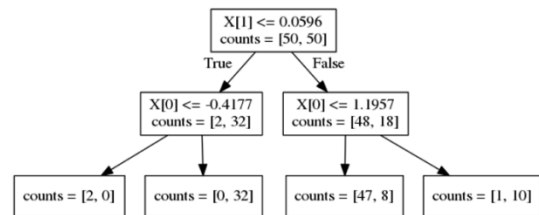
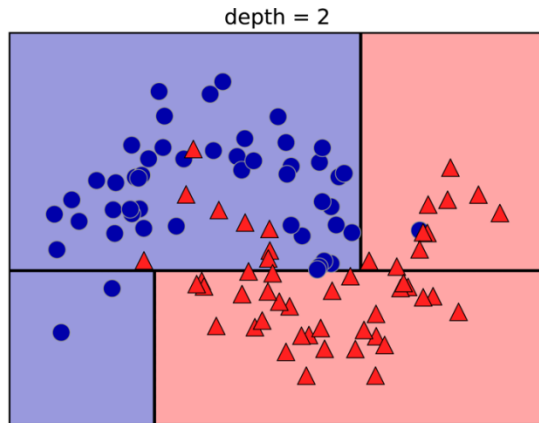
$X_m$  osservazioni nel nodo m

$y$  classi

$p_m$  distribuzione sulle classi nel nodo m

## Predizione

attraverso l'albero fino alla foglia molto veloce



## Alberi di regressione

La predizione è la media dei target nella foglia

- Predizione

$$\bar{y}_m = \frac{1}{N_m} \sum_{i \in N_m} y_i$$

Il criteri di impurity invece

- Mean Squared Error:

$$H(X_m) = \frac{1}{N_m} \sum_{i \in N_m} (y_i - \bar{y}_m)^2$$

- Mean Absolute Error:

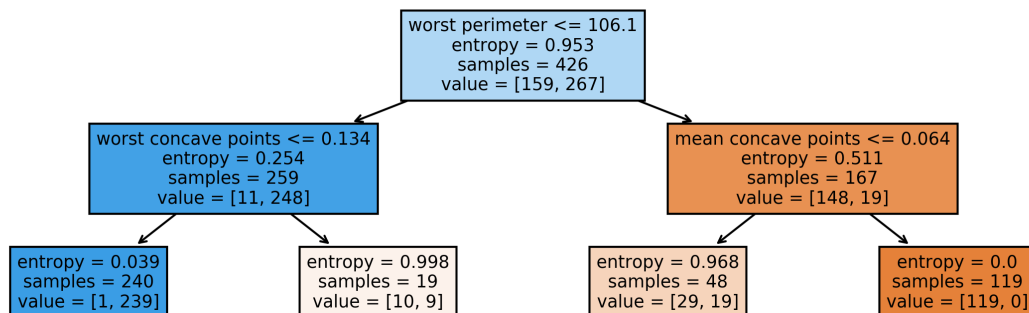
$$H(X_m) = \frac{1}{N_m} \sum_{i \in N_m} |y_i - \bar{y}_m|$$

## Visualizzare gli alberi con sklearn

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    stratify=cancer.target,
                                                    random_state=0)
```

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=2)
tree.fit(X_train, y_train)
```

```
from sklearn.tree import plot_tree
tree_dot = plot_tree(tree, feature_names=cancer.feature_names)
```



## Tuning dei parametri

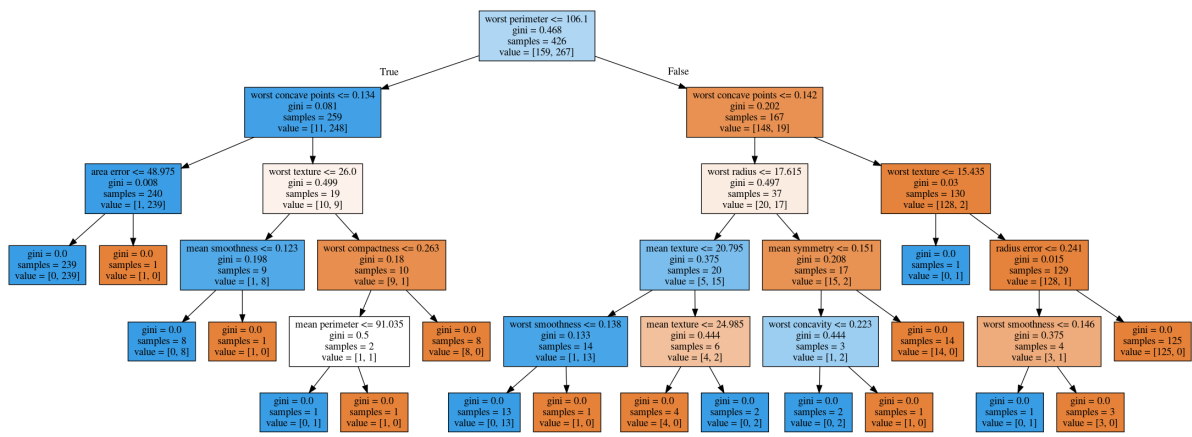
### Pre-pruning vs post-pruning

### Pre-pruning: Limita la dimensione dell'albero

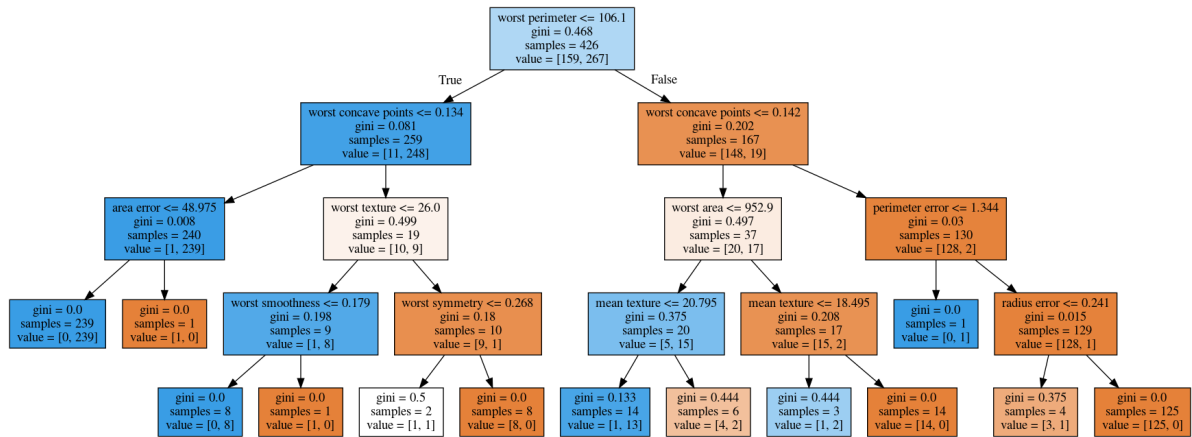
*esempi*

- max\_depth
- max\_leaf\_nodes
- min\_samples\_split
- min\_impurity\_decrease

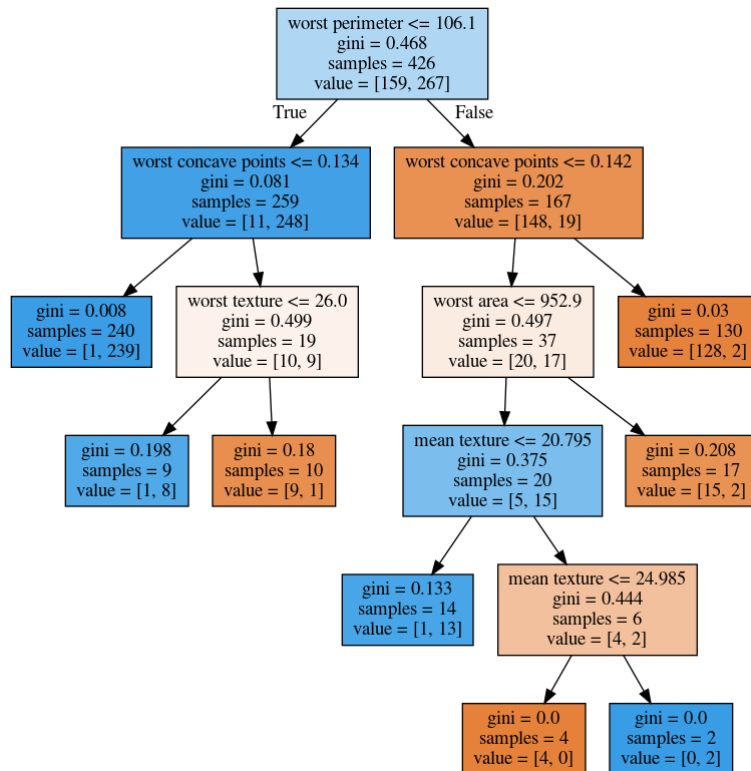
### Nessun pruning



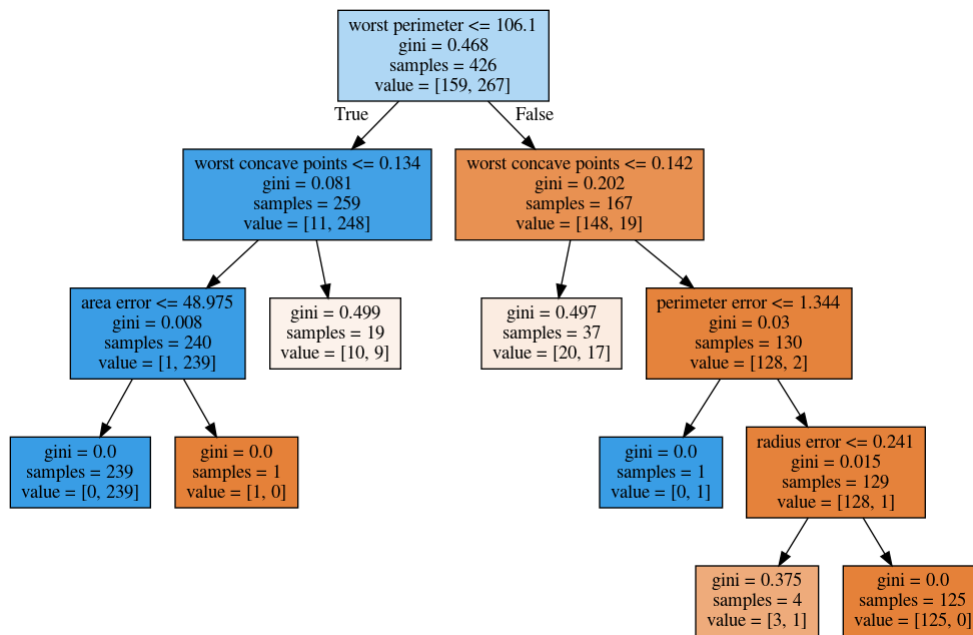
max\_depth = 4



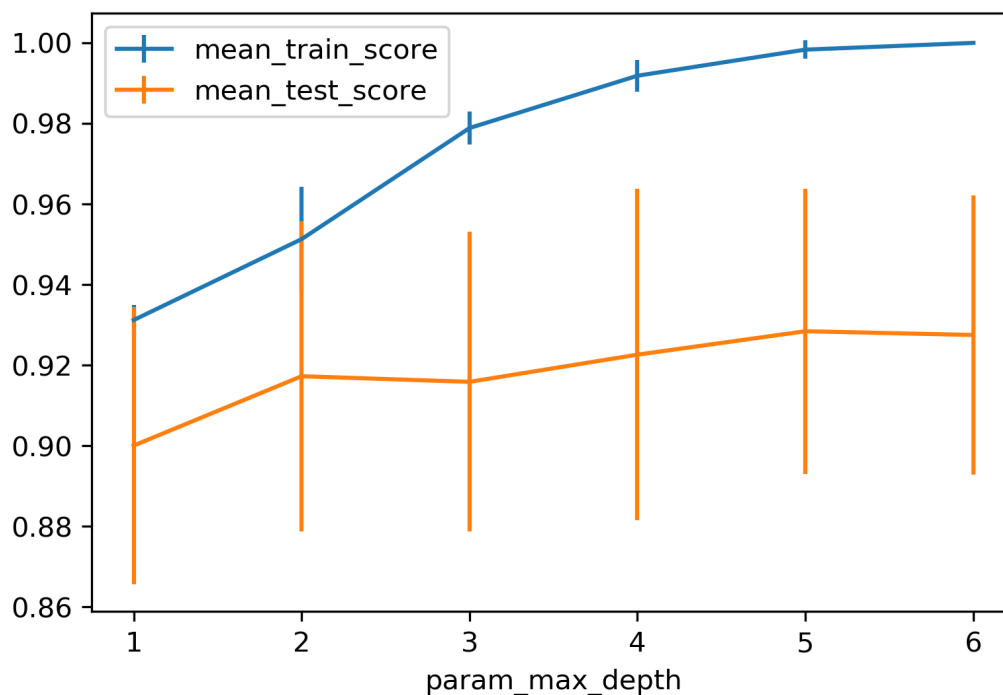
max\_leaf\_nodes = 8



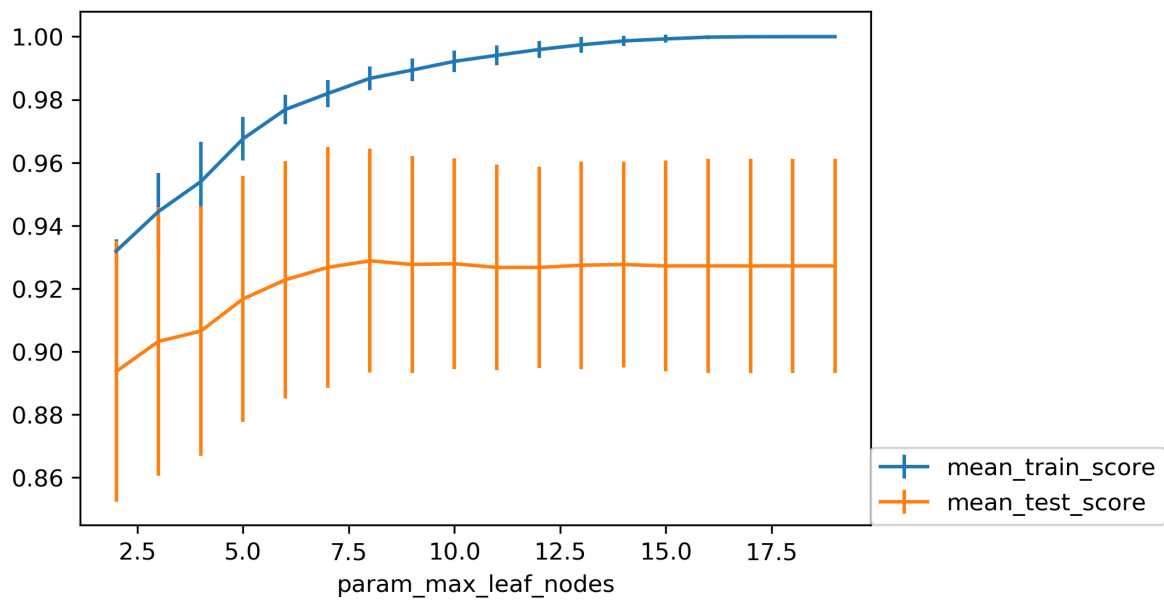
min\_samples\_split = 50



```
from sklearn.model_selection import GridSearchCV
param_grid = {'max_depth':range(1, 7)}
grid = GridSearchCV(DecisionTreeClassifier(random_state=0),
                    param_grid=param_grid, cv=10)
grid.fit(X_train, y_train)
```



```
param_grid = {'max_leaf_nodes':range(2, 20)}
```



## Post pruning

### Cost Complexity Pruning

$$R_{\alpha}(T) = R(T) + \alpha|T|$$

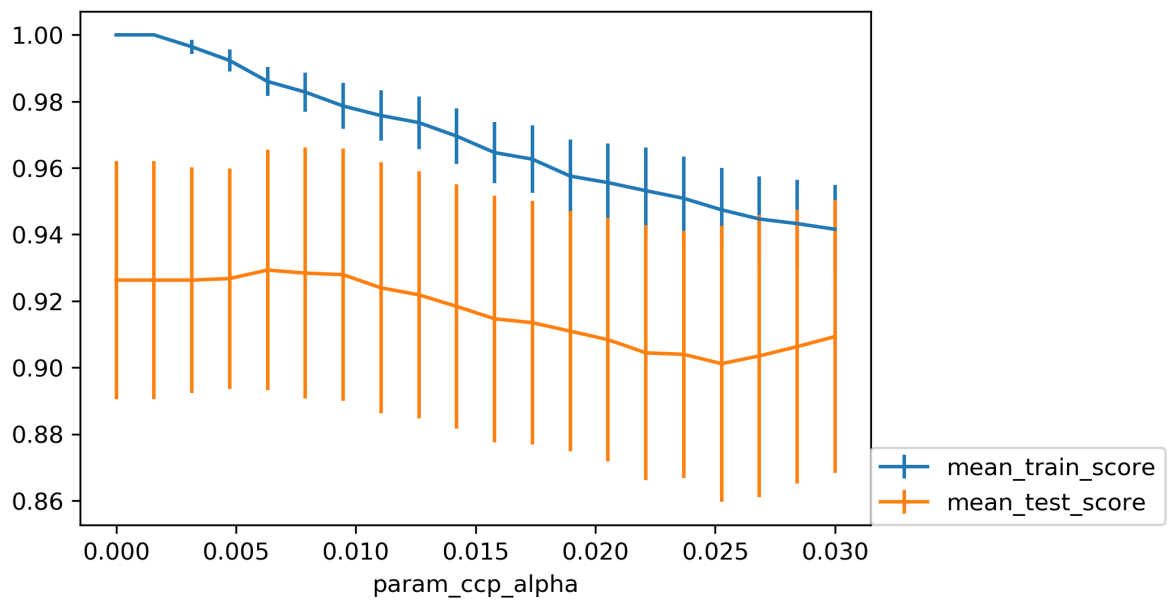
$R(T)$  is total leaf impurity

$|T|$  is number of leaf nodes

$\alpha$  is free parameter.

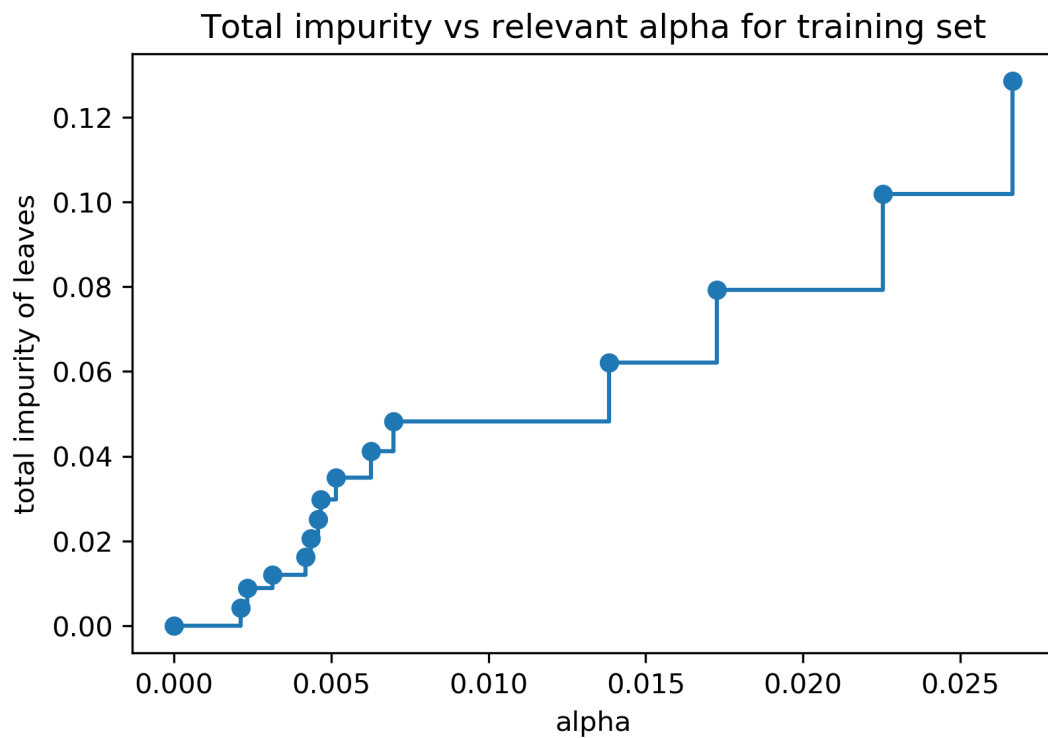
```
param_grid = {'ccp_alpha': np.linspace(0., 0.03, 20)}
```

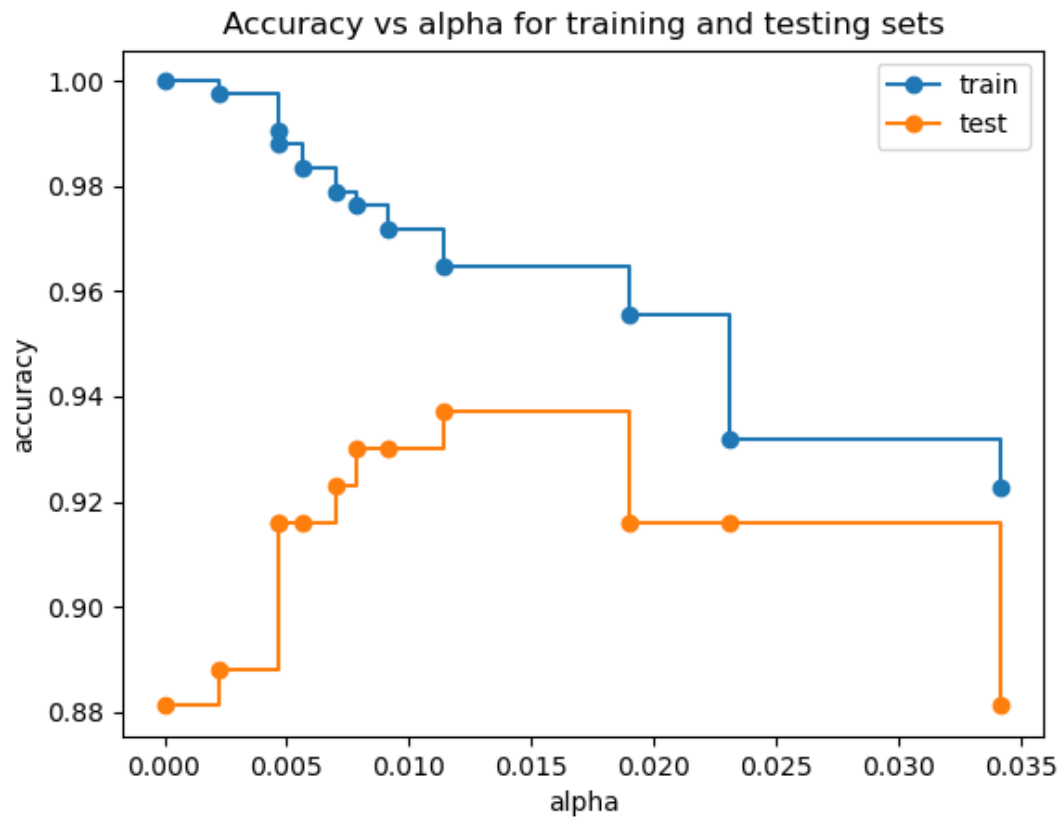




## Pruning efficiente

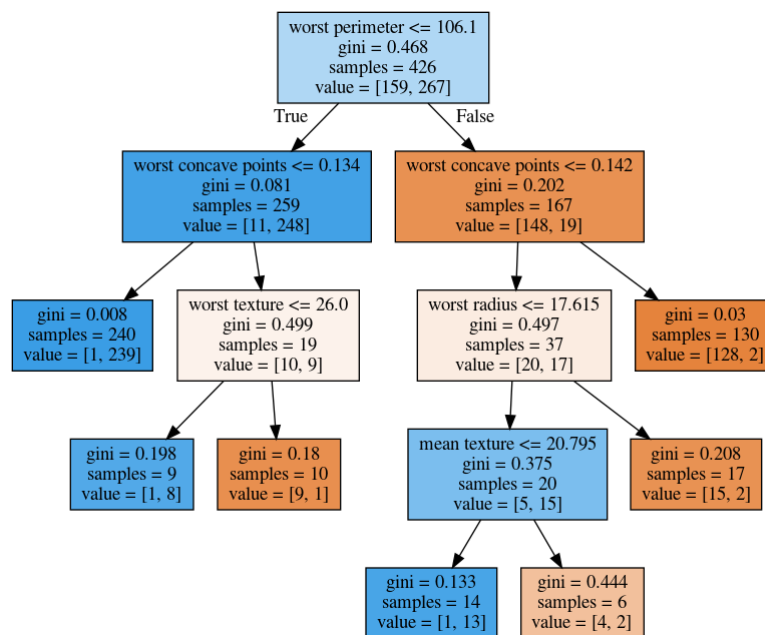
```
clf = DecisionTreeClassifier(random_state=0)
path = clf.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```



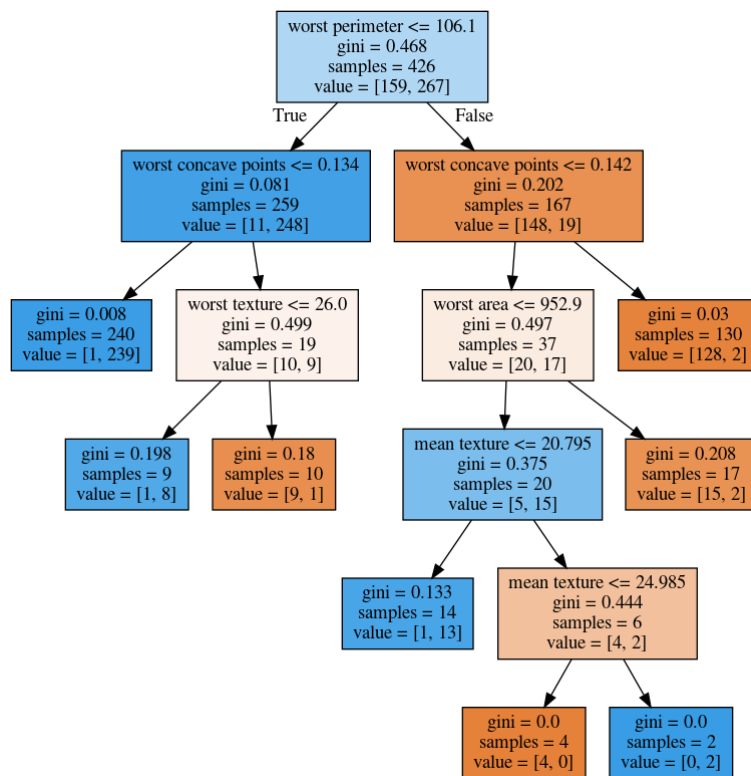


## Confronto post-pruned vs pre-pruned

### Cost-complexity pruning



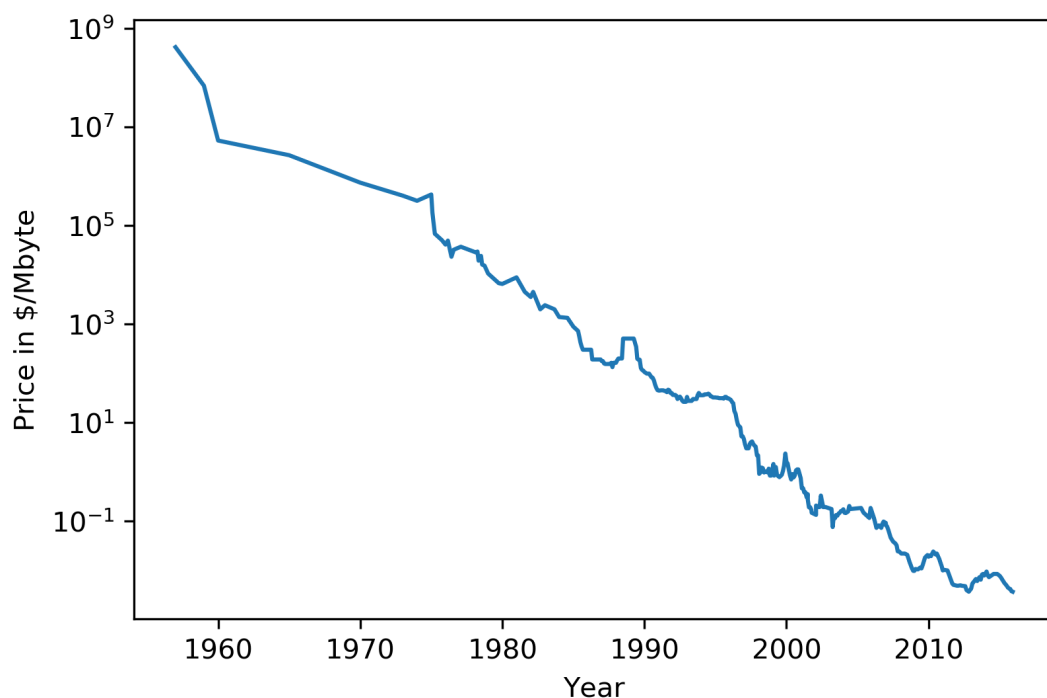
### Max leaf nodes search

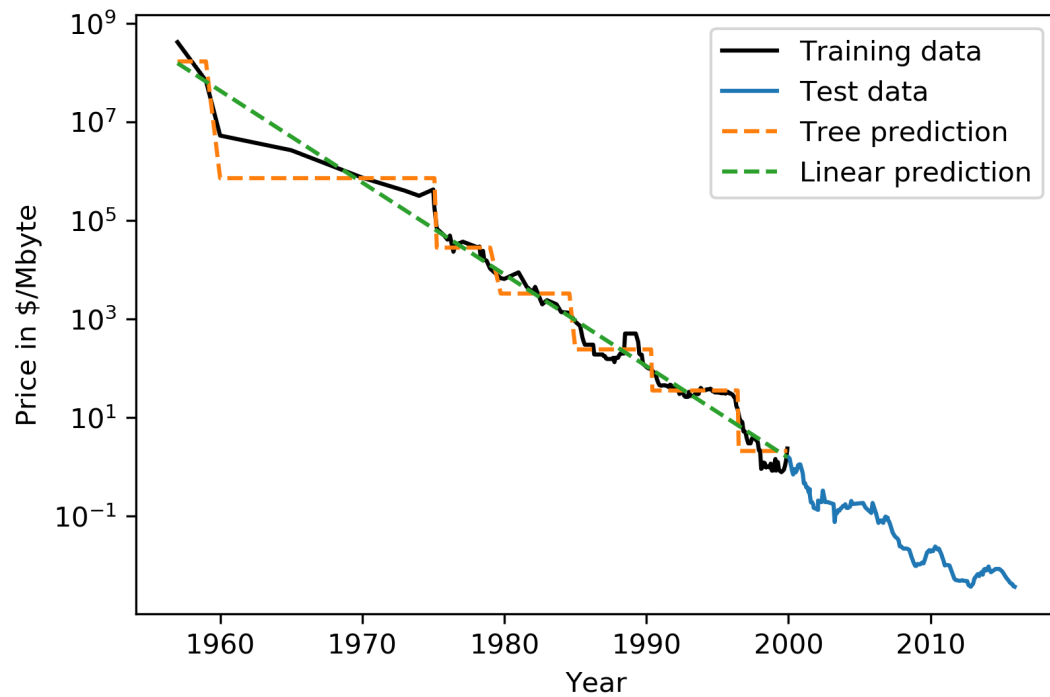


## Estrapolazione

- Molti modelli in Machine Learning sono nati per interpolare e non sanno estrapolare bene
  - estrapolare è estendere la predizione lontano dall "*nuvola*" dei dati
- OSSERVAZIONE: C'è differenza tra estrapolazione e generalizzazione

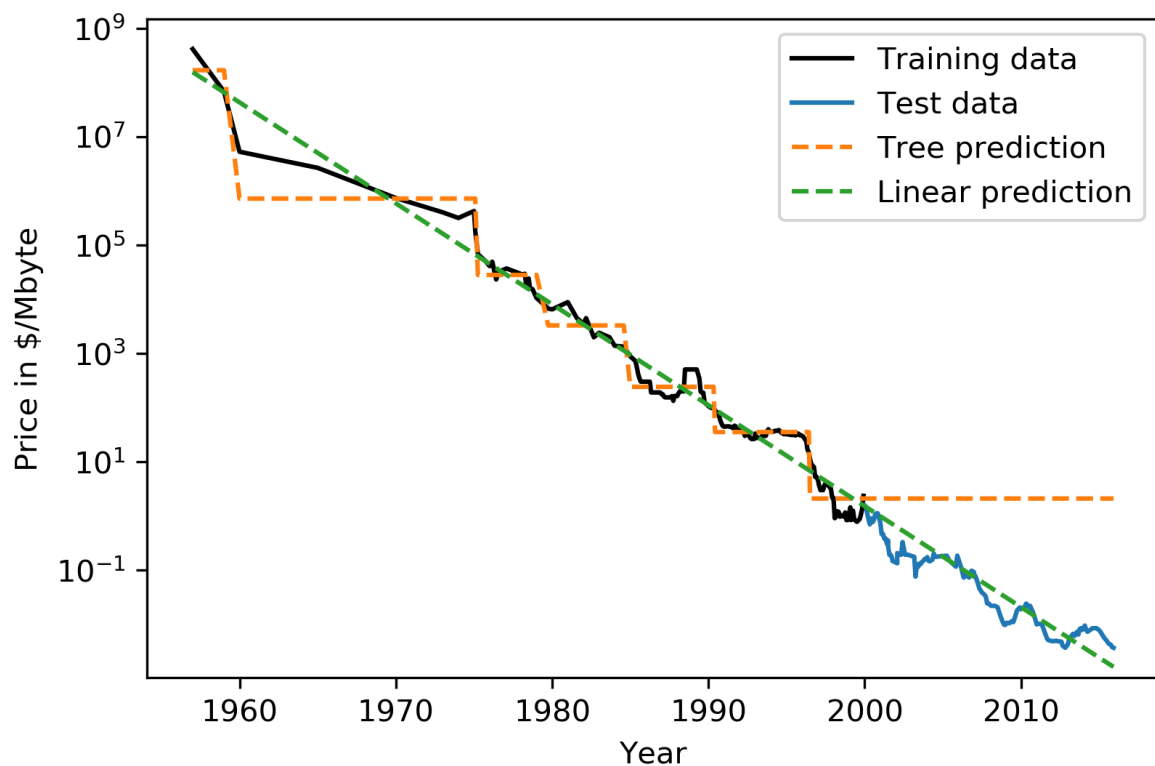
Prendiamo una serie temporale





“gli alberi non sanno estrapolare”

I Decision Tree sono come il KNN effettuano medie dei valori vicini (quelli presenti nella stessa foglia)



- non riescono a vedere il trend

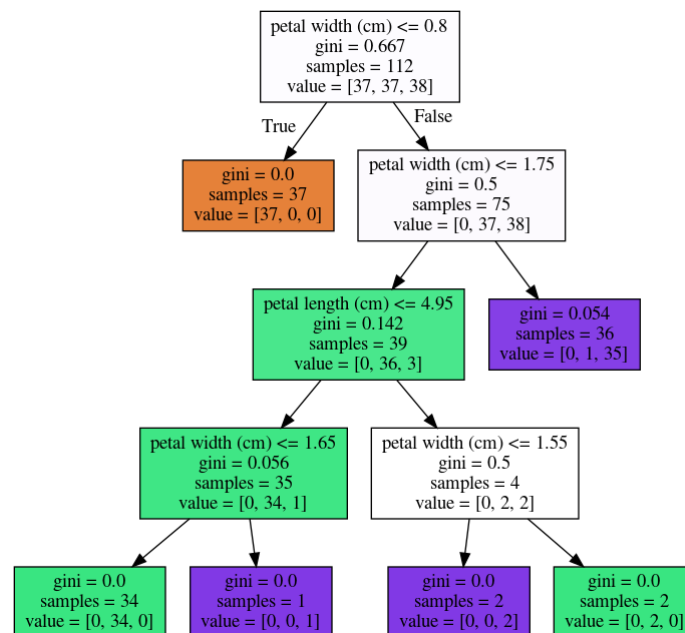
- Come posso ovviare a questa situazione?

## Relazione con il Nearest Neighbors

- Predice la media dei vicini che siano k, una  $\epsilon$ -palla o una foglia.
- Gli alberi sono molto più veloci in predizione.
- Entrambi non riescono ad estrapolare
- 1KK e Decision Tree non regolarizzati hanno un *perfetto overfit*

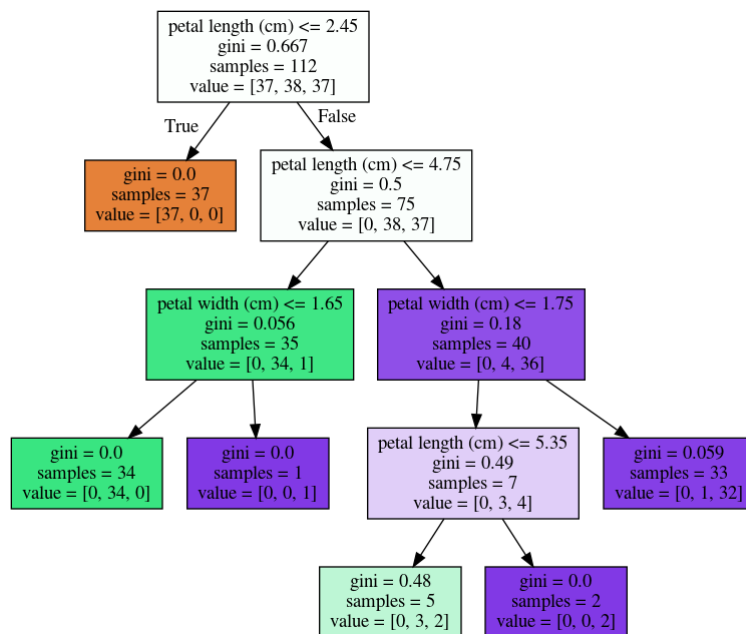
## Instabilità

```
x_train, x_test, y_train, y_test = train_test_split(
    iris.data, iris.target, stratify=iris.target, random_state=0)
tree = DecisionTreeClassifier(max_leaf_nodes=6)
tree.fit(X_train, y_train)
```



cambio solo il seme

```
x_train, x_test, y_train, y_test = train_test_split(
    iris.data, iris.target, stratify=iris.target, random_state=1)
tree = DecisionTreeClassifier(max_leaf_nodes=6)
tree.fit(X_train, y_train)
```

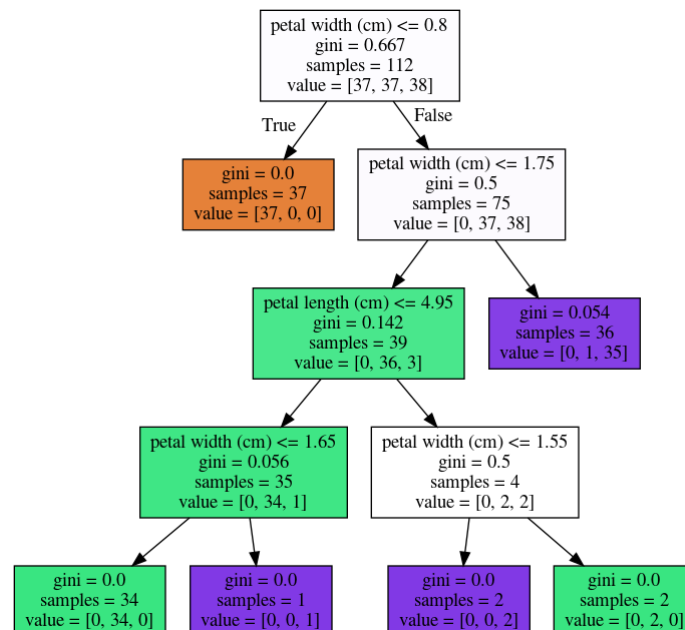


## Feature importance

```

X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, stratify=iris.target, random_state=0)
tree = DecisionTreeClassifier(max_leaf_nodes=6)
tree.fit(X_train, y_train)

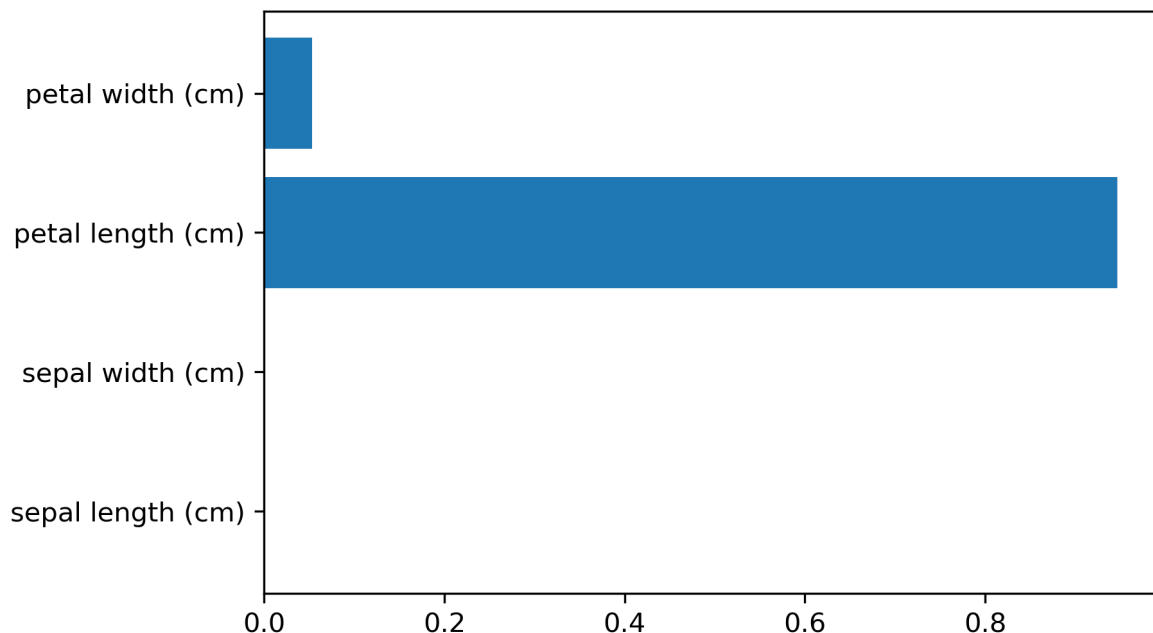
```



```

tree.feature_importances_
array([0.0, 0.0, 0.414, 0.586])

```



Ogni volta che una particolare feature è scelta nell'albero accumuli come viene diminuita la sua impurezza.

Se uso una feature più volte aggiungo le diminuzioni di purezza (gli aumenti di purezza)

Il problema principale è che l'instabilità del modello rende instabile anche la feature importance

(ci sono poi problemi che vanno oltre i limiti del modello la causalità, se diverse feature sono correlate *sostituibili tra loro* e trattare modelli con un numero enorme di feature).

---

## Dati Categorici

- Trattano i dati categorici nativamente
- Gli split vengono fatti tra insiemi diversi
- $2^n$  valori molte possibilità di split
- Possibile da fare in  $O(n\_values \cdot \log(n\_values))$  esattamente per gini index e classificazione binaria
- Viene usata un'euristica in pratica per il multiclasse
- Non c'è ancora in sklearn

---

## Predire le probabilità

- Frazione della classe nella foglia
- Senza il pruning sempre 100% certo
- Perfino con il pruning troppa sicurezza

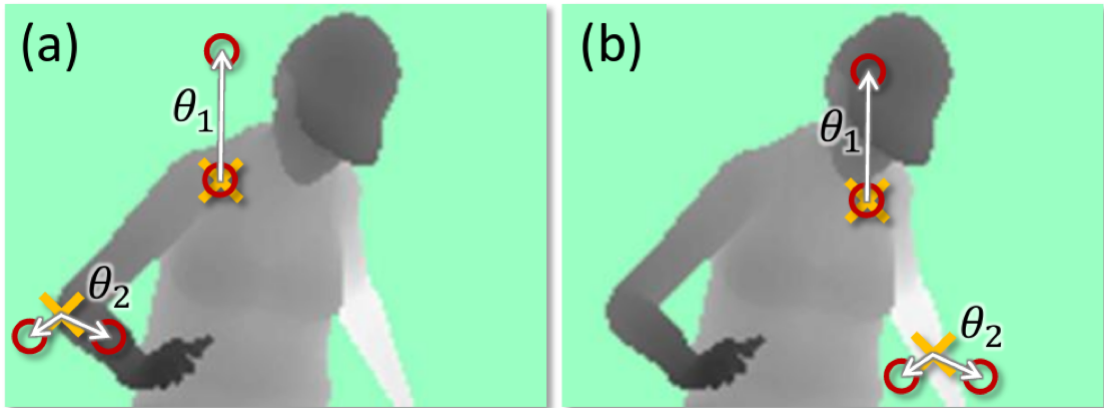
---

## Alberi di inferenza condizionale

- Scelgono il “best” split correggendo per diversi test di ipotesi
- Più “fair” verso le variabili categoriche
- c’è solo in R (sic!)

## I decision tree sono molto flessibili

(preso da Shotton et. al. Real-Time Human Pose Recognition ..)



Un'immagine di profondità per capire dove sono le differenti parti del corpo

Mi chiedo la profondità di altri pixel rispetto a un pixel di riferimento

Quindi paragono pixel o regioni di pixel

Vantaggi

- posso usare qualsiasi cosa come candidato allo split
- Computer vision: paragoni di pixel
- ad esempio alberi che hanno modelli lineari sulle foglie

## Ensemble di modelli

- Costruisco modelli differenti
- Medio i risultati
- Owen Zhang (primo su kaggle per molto tempo): modelli XGBoosting con diversi random seed
- Più modelli sono meglio se non sono correlati
- Funziona anche con le reti neurali
- Puoi fare la media di quanti modelli vuoi basta che diano buone probabilità (*calibrate*)
- Scikit-learn: VotingClassifier hard and soft voting



Soft voting vuol dire fare la media delle probabilità e prendere arg max.

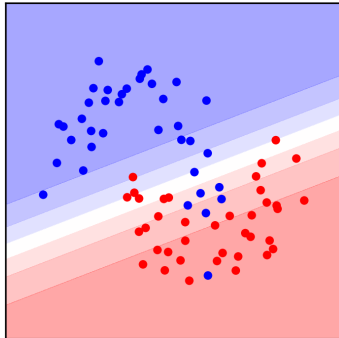
Hard voting vuol dire ognuno fa una previsione e prendiamo quello più votato

## VotingClassifier

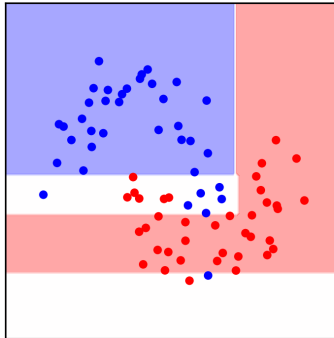
```
voting = VotingClassifier(  
    [('logreg', LogisticRegression(C=100)),  
    ('tree', DecisionTreeClassifier(max_depth=3, random_state=0))],  
    voting='soft')  
voting.fit(X_train, y_train)  
lr, tree = voting.estimators_  
tree.score(X_test, y_test), lr.score(X_test, y_test), voting.score(X_test,  
y_test)
```

0.80 0.84 0.88

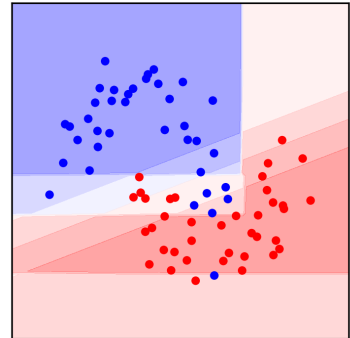
Logistic Regression acc=0.84



Decision Tree acc=0.80

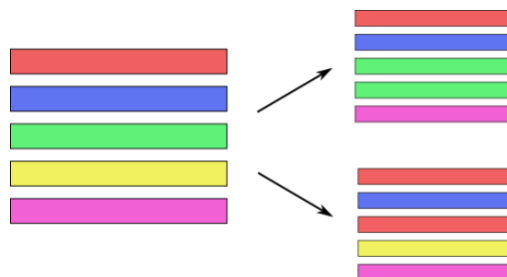


Voting Classifier acc=0.88



## Bagging (Bootstrap AGGregation)

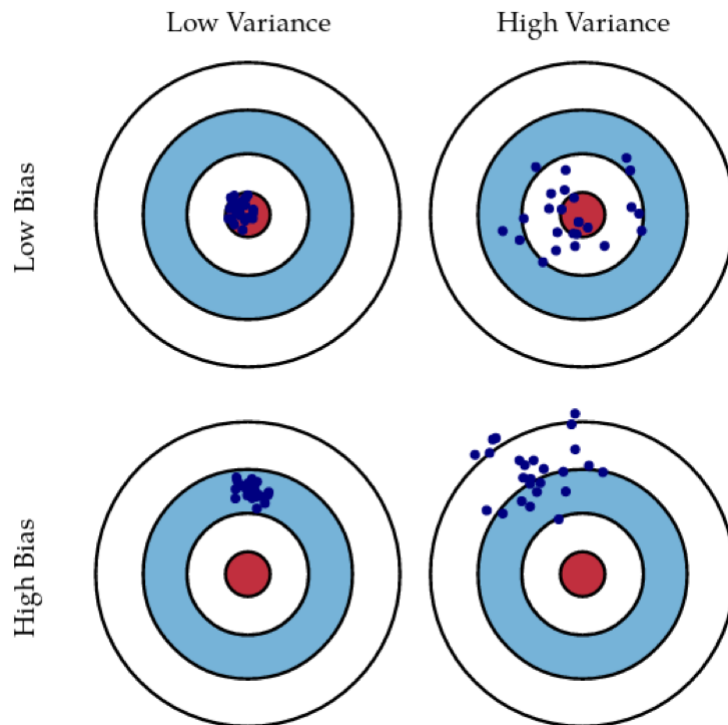
- Modo generale per costruire modelli abbastanza differenti
- Bootstrap sollevandosi tirandosi per i lacci



- BaggingClassifier, BaggingRegressor

# Bias - Variance (tradeoff)

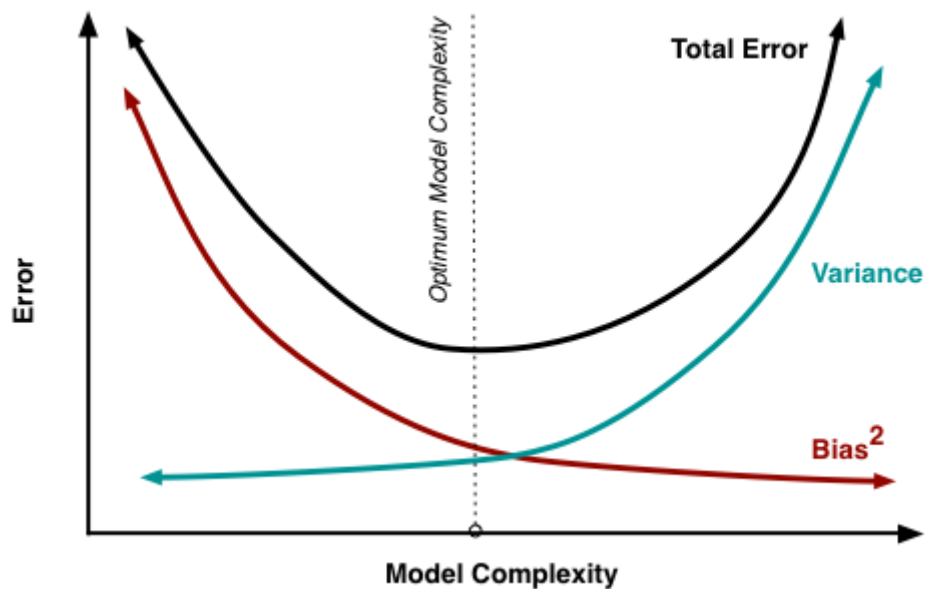
---



L'errore per una Loss quadratica ( $l_2$ ) è  $Err(x) = E[(Y - \hat{f}(x))^2]$

$$Err(x) = \left(E[\hat{f}(x)] - f(x)\right)^2 + E\left[\left(\hat{f}(x) - E[\hat{f}(x)]\right)^2\right] + \sigma_e^2$$

$$Err(x) = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$



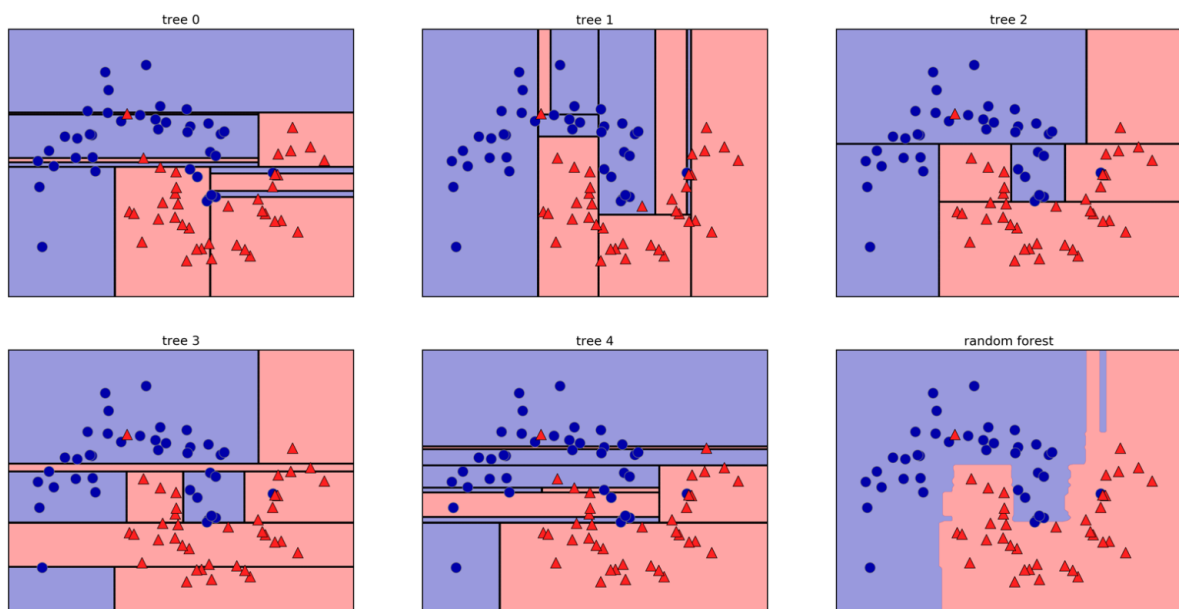
### Approfondimento.

<http://scott.fortmann-roe.com/docs/BiasVariance.html>

Si può calcolare con [http://rasbt.github.io/mlxtend/user\\_guide/evaluate/bias\\_variance\\_decomp/](http://rasbt.github.io/mlxtend/user_guide/evaluate/bias_variance_decomp/) di Raschka <-- *lookup*

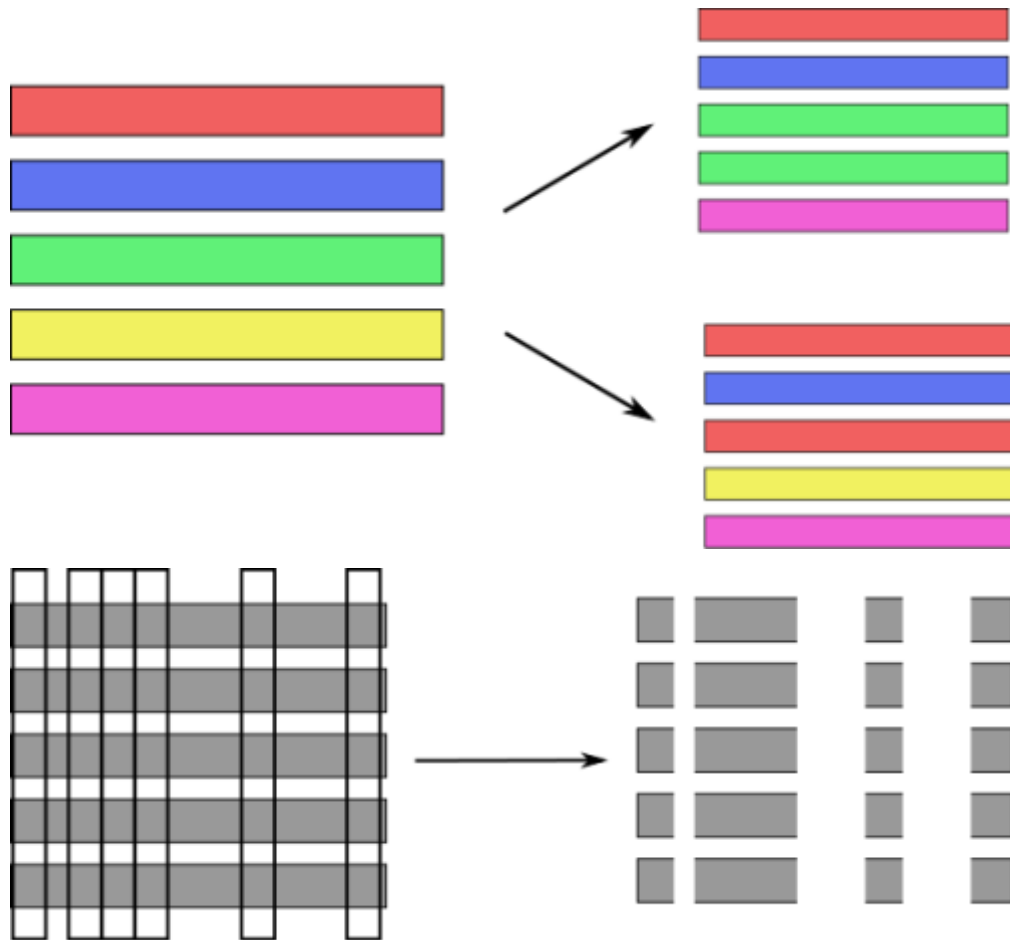
- Breiman ha mostrato che la generalizzazione dipende dalla forza dei singoli classificatori e *inversamente* dalla loro correlazione
- Trovare modelli scorrelati potrebbe essere più importante che trovare modelli forti

## Random Forest



## Randomizzare in due modi

---



- Per ogni albero:
  - campionamento bootstrap dei dati
- Per ogni split:
  - campionamento random delle feature a caso
- Più alberi sono sempre meglio

## Tuning Random Forest

---

- Il parametro più importante: `max_features`
  - circa  $\sqrt{n\_features}$  per la classificazione
  - circa `n_features` per la regressione
- `n_estimators > 100`
- Prepruning potrebbe aiutare sicuramente ridurre la dimensione del modello
- `max_depth`, `max_leaf_nodes`, `min_samples_split`

## Extremely Randomized Trees (XT)

---

- Ancora più a caso

- Il threshold è scelto a caso per ogni feature
- non usa il bootstrap
- più veloce non cerca e non mette in ordine
- ha boundary più smooth

Random forest e ERT funzionano senza preprocessing. Basta che metti il numero di estimatori abbastanza alto.

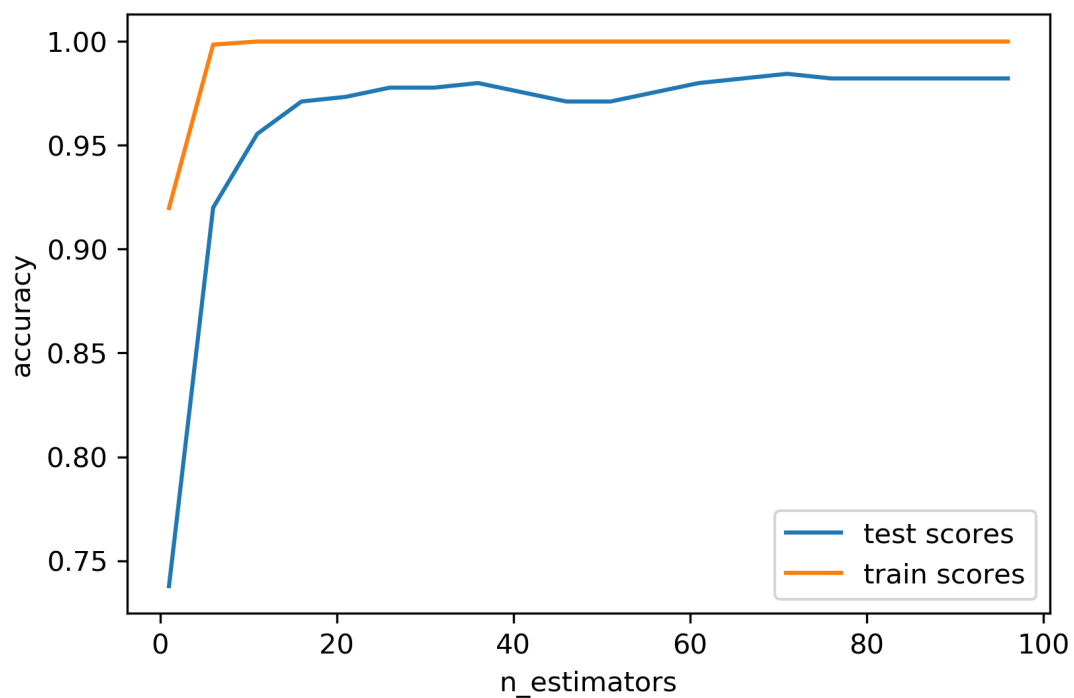
Non sarà sempre il modello migliore ma funzionano subito

## Warm-Start

- partiamo da un modello preaddestrato
- possiamo usarlo per un apprendimento **incrementale**

```
train_scores = []
test_scores = []

rf = RandomForestClassifier(warm_start=True)
estimator_range = range(1, 100, 5)
for n_estimators in estimator_range:
    rf.n_estimators = n_estimators
    rf.fit(X_train, y_train)
    train_scores.append(rf.score(X_train, y_train))
    test_scores.append(rf.score(X_test, y_test))
```



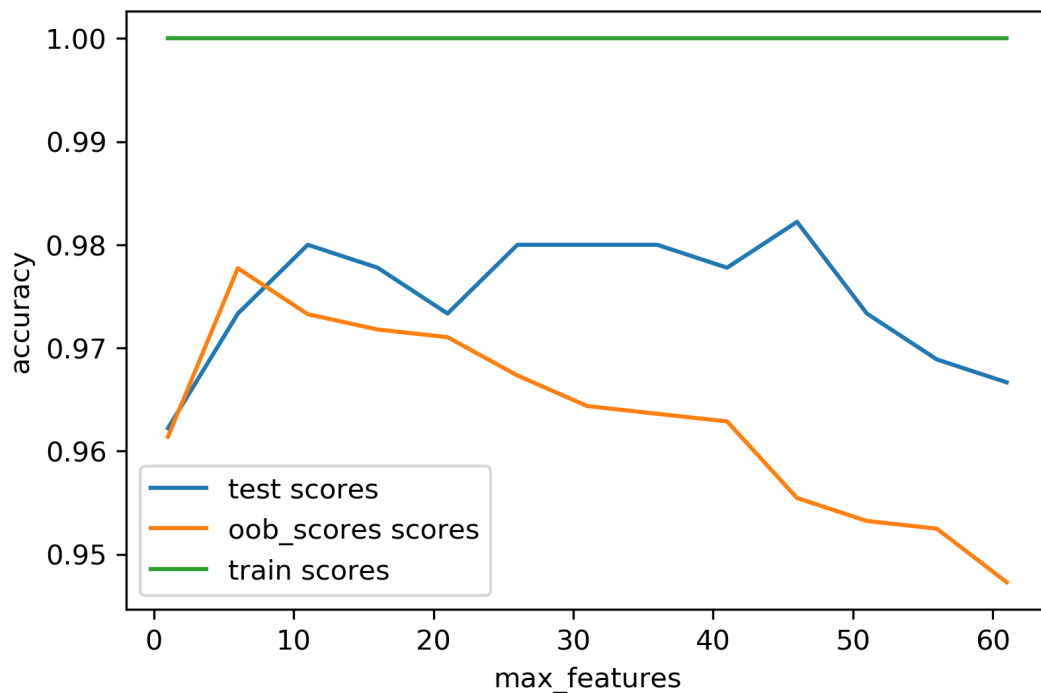
# Stime Out-of-bag

- Ogni tree usa circa ~66% dei dati
- Possiamo valutarlo sul resto!
- Facciamo previsioni sull' out-of-bag, average, score.
- Ogni predizione è una media su differenti sottoinsiemi di alberi

```
train_scores = []
test_scores = []
oob_scores = []

feature_range = range(1, 64, 5)
for max_features in feature_range:
    rf = RandomForestClassifier(max_features=max_features, oob_score=True,
                               n_estimators=200, random_state=0)

    rf.fit(X_train, y_train)
    train_scores.append(rf.score(X_train, y_train))
    test_scores.append(rf.score(X_test, y_test))
    oob_scores.append(rf.oob_score_)
```

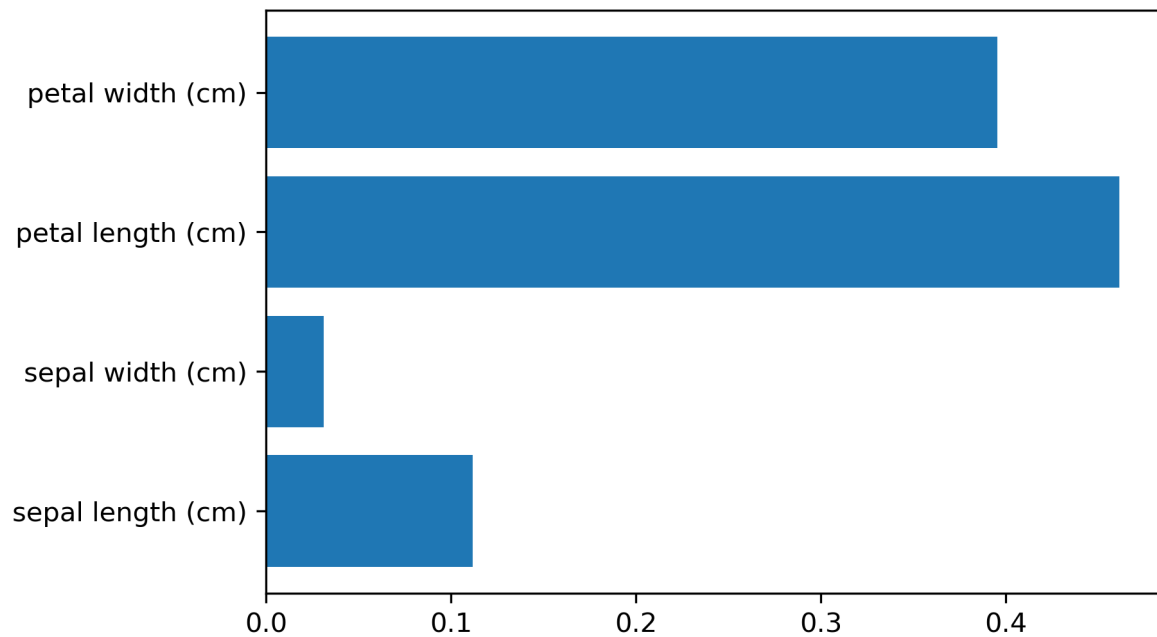


- Permette di avere un risultato paragonabile alla cross validation senza farla
- OOB score in generale è **peggiore** del validation score.
- Se avviene il contrario siamo in un caso di overfitting o shift del modello

## Variable Importance

```
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, stratify=iris.target, random_state=1)
rf = RandomForestClassifier().fit(X_train, y_train)
rf.feature_importances_
plt.barh(range(4), rf.feature_importances_)
plt.yticks(range(4), iris.feature_names);
```

```
array([ 0.126,  0.033,  0.445,  0.396])
```



Com'è calcolata la feature importance?

Quando una particolare feature viene usata guardo alla diminuzione di impurità le aggrego e alla fine le normalizzo a somma 1