Università degli Studi di Cagliari
Corso di Laurea DSBAI

# Web Analytics e Analisi Testuale

http://agile-group.org

A.A. 2019/2020

Ing. Marco Ortu
Via Porcell 4, primo piano
mail: marco.ortu@unica.it

# Introduzione a Scikit-Learn

# Scikit-Learn

Machine Learning library

➔ Sviluppata con ***NumPy*** e ***SciPy***

Features:

➔ Classification

➔ Clustering

➔ Regression

Website

***http://scikit-learn.org/***

# Scikit-Learn

Il Machine Learning (ML) è l'arte di creare una "spiegazione" del mondo utilizzando una grande quantità di dati presi dal mondo reale.

Formalmente, ML è il campo dell'informatica che si occupa dello studio e dello sviluppo di sistemi che possono imparare dai dati.

- ➔ Model: l'insieme delle features che rappresentano un input da classificare.
- ➔ Data: i dati di input del model
- ➔ Target: il valore che si cerca di predire, è l'output del sistema
- ➔ Features: attributi del dati che sono utilizzati per la predizione
- ➔ Methods: gli algoritmi utilizzati per la predizione

# Scikit-Learn

Un problema di apprendimento considera un insieme di n campioni di dati e cerca di prevedere proprietà di dati sconosciuti.

***Supervised Classification***

Se i sistemi apprendono da dati già etichettati (***training set***) per prevedere la classe dei campioni sconosciuti (***test set***) allora si parla di ***Classificazione***. Se l'output desiderato è costituito da una o più variabili continue, allora si parla di ***Regressione***.

***Unsupervised Learning***

Tutti i campioni sono sconosciuti e i compiti principali sono:

1. raggruppare i campioni simili (***clustering***).
2. determinare la distribuzione dei campioni (***density estimation***).
3. proiettare i dati descritti in uno spazio dimensionale a molte dimensioni in uno spazio di ordine inferiore (***dimensionality reduction***).

# Scikit-Learn: Toy Data

Esattamente come NLTK anche Scikit-Learn viene fornito con un insieme di dati di esempio (come i ***corpus*** per NLTK) chiamati datasets.

Un ***dataset object*** ha i seguenti attributi (***n*** è il numero di campioni, ***m*** il numero di features):

- ***data***: Numpy 2D-array, di dimensione nxm.
- ***feature_names***: una lista di dimensione m che contiene i nomi delle features.
- ***target***: un array di dimensione n contenente contenente i numeri associati alle classi di campioni.
- ***target_names***: una lista di dimensione n che contiene i nomi delle classi

# Scikit-Learn: Toy Data

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> iris.data
array([[ 5.1, 3.5, 1.4, 0.2],
[ 4.9, 3. , 1.4, 0.2],
[ 4.7, 3.2, 1.3, 0.2],
…
>>> iris.feature_names
['sepal length (cm)',
'sepal width (cm)',
'petal length (cm)',
'petal width (cm)']
>>> iris.target_names
array(['setosa', 'versicolor', 'virginica'], dtype='|S10')
```

# Scikit-Learn: Toy Data

```
>>> from sklearn import datasets as ds
>>>  cats = ['alt.atheism', 'sci.space']
>>> text = ds.fetch_20newsgroups(subset ='train', categories = cats)
>>> text.target_names
['alt.atheism', 'sci.space']
>>> print text.filenames
['.../scikit_learn_data/20news_home/20news-bydate-train/alt.atheism/51312'
 '.../scikit_learn_data/20news_home/20news-bydate-train/sci.space/60929'
 '.../scikit_learn_data/20news_home/20news-bydate-train/sci.space/61239'
 ...
```

Il metodo ***fetch_20newsgroups()*** permette di ottenere un dataset pronto all'uso, ovvero con le features già pronte.

# Scikit-Learn: Text Processing

I dati di testo richiedono una preparazione speciale prima di poter essere utilizzati per la modellazione predittiva.

Il testo deve essere analizzato per rimuovere le ***stop-words*** per poi passare alla tokenizzazione.

Quindi i ***token*** devono essere codificati come numeri interi o in virgola mobile da utilizzare come input per un algoritmo di apprendimento automatico, passaggio chiamato ***feature extraction*** (o ***vectorization***).

La libreria di scikit-learn offre strumenti facili da usare per eseguire sia la tokenizzazione che la feature extraction dei dati di tipo testuale.

# Scikit-Learn: Text Processing

**Bag-of-Words Model**

Non possiamo lavorare direttamente con il testo quando si usano algoritmi di apprendimento automatico.

Invece, abbiamo bisogno di convertire il testo in numeri.

Potremmo voler eseguire la classificazione dei documenti, quindi ogni documento è un "input" e un'etichetta di classe è l'output del nostro algoritmo predittivo. Gli algoritmi prendono i vettori di numeri come input, quindi è necessario convertire i documenti in vettori di numeri a lunghezza fissa.

Un modello semplice ed efficace per pensare ai documenti di testo in machine learning è chiamato il modello ***Bag-of-Words***, o ***BoW***.

Il modello è semplice in quanto rigetta tutte le informazioni sull'ordine nelle parole e si focalizza sulle ***occorrenze*** delle parole in un documento.

# Scikit-Learn: Text Processing

**Bag-of-Words Model**

Si assegna ad ogni parola un numero univoco. Quindi qualsiasi documento può essere codificato come un vettore a lunghezza fissa con la lunghezza del vocabolario di parole conosciute.

Il valore in ciascuna posizione nel vettore potrebbe essere riempito con un ***conteggio*** o una ***frequenza*** di ciascuna parola nel documento codificato.

Questo è il modello ***BoW***, in cui ci occupiamo solo di schemi di codifica che rappresentano se le parole sono presenti o il grado in cui sono presenti nei documenti codificati senza alcuna informazione sull'ordine.

La libreria di scikit-learn fornisce 3 diversi schemi che possiamo usare.

# Scikit-Learn: Text Processing

**Vectorizer**

- **fit([...])**
- **transform([...])**

***CountVectorizer*** ***TfidfVectorizer*** **Hashing*Vectorizer***

# Scikit-Learn: Text Processing

**Word Counts with CountVectorizer**

***CountVectorizer*** fornisce un modo semplice per sia ***tokenizzare*** una raccolta di documenti di testo e costruire un vocabolario di parole conosciute, ma anche per codificare nuovi documenti utilizzando quel vocabolario.

- ➔ Creare un'istanza della classe ***CountVectorizer***.
- ➔ Chiama la funzione fit () per imparare un vocabolario da uno o più documenti.
- ➔ Chiama la funzione transform () su uno o più documenti secondo necessità per codificarli come un vettore.

Un vettore codificato viene restituito con una lunghezza pari all'intero vocabolario e un numero intero per il numero di volte in cui ogni parola è comparsa nel documento.

# Scikit-Learn: Text Processing

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> text = ["The quick brown fox jumped over the lazy dog."]
>>> vectorizer = CountVectorizer()
>>> vectorizer.fit(text)
>>> print(vectorizer.vocabulary_)
{u'brown': 0, u'lazy': 4, u'jumped': 3, u'over': 5, u'fox': 2, u'dog': 1, u'quick': 6, u'the': 7}
>>> vector = vectorizer.transform(text)
>>> print(vector.shape)
(1, 8)
>>> print(type(vector))
<class 'scipy.sparse.csr.csr_matrix'>
>>> print(vector.toarray())
[[1 1 1 1 1 1 1 2]]
```

# Scikit-Learn: Text Processing

È importante sottolineare che lo stesso ***vectorizer*** può essere utilizzato su documenti che contengono parole non incluse nel vocabolario.

Queste parole sono ignorate e non viene fornito alcun conteggio nel vettore risultante.

Ad esempio, di seguito è riportato un esempio di utilizzo del ***vectorizer*** sopra per codificare un documento con una parola nel vocab e una parola che non lo è.

```
>>> text2 = ["the puppy"]
>>> vector = vectorizer.transform(text2)
>>> print(vector.toarray())
[[0 0 0 0 0 0 0 1]]
```

# Scikit-Learn: Text Processing

**Word Frequencies with TfidfVectorizer**

I conteggi delle parole sono un buon punto di partenza, ma sono molto basilari.

Un problema con i conteggi semplici è che alcune parole come "***The***" appariranno molte volte e i loro grandi conteggi non saranno molto significativi nei vettori codificati.

Un'alternativa è quella di calcolare le frequenze delle parole, e il metodo di gran lunga più popolare è chiamato ***TF-IDF***. Questo è un acronimo che sta per "***Term Frequency*** - ***Inverse Document Frequency***" che sono le metriche risultanti assegnate a ciascun ***token***.

***Term Frequency***: riassume la frequenza con cui una determinata parola appare all'interno di un documento.

***Inverse Document Frequency***: questo ridimensiona le parole che occorrono molto in più documenti.

# Scikit-Learn: Text Processing

**Word Frequencies with TfidfVectorizer**

***TfidfVectorizer*** consente di ***tokenizzare*** i documenti, imparare il vocabolario e il calcolo della ***TF-IDF*** del documento e consente di codificare nuovi documenti.

In alternativa, se si ha già un ***CountVectorizer*** addestrato, lo si può usare con un ***TfidfTransformer*** per calcolare semplicemente le frequenze inverse del documento e avviare i documenti di codifica.

E' lo stesso processo di creazione, adattamento e trasformazione che viene utilizzato con il ***CountVectorizer***.

# Scikit-Learn: Text Processing

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> text = ["The quick brown fox jumped over the lazy dog.",
            "The dog.",
            "The fox"]
>>> vectorizer = TfidfVectorizer()
>>> vectorizer.fit(text)
>>> print(vectorizer.vocabulary_)
{u'brown': 0, u'lazy': 4, u'jumped': 3, u'over': 5, u'fox': 2, u'dog': 1, u'quick': 6, u'the': 7}
>>> print(vectorizer.idf_)
[1.69314718 1.28768207 1.28768207 1.69314718 1.69314718 1.69314718
 1.69314718 1.        ]
>>> vector = vectorizer.transform([text[0]])
>>> print(vector.shape)
(1, 8)
>>> print(vector.toarray())
[[0.36388646 0.27674503 0.27674503 0.36388646 0.36388646 0.36388646
  0.36388646 0.42983441]]
```

# Scikit-Learn: Text Processing

Dai documenti viene appreso un vocabolario di 8 parole e ad ogni parola viene assegnato un indice intero univoco nel vettore di output.

Le ***idf*** sono calcolate per ogni parola nel vocabolario, assegnando il punteggio più basso di 1.0 alla parola più frequentemente osservata: "***The***" all'indice 7.

Infine, il primo documento è codificato come array sparse a 8 elementi e possiamo rivedere i punteggi finali di ogni parola con valori diversi per "the", "fox" e "dog" dalle altre parole del vocabolario.

# Scikit-Learn: Text Processing

**Hashing with HashingVectorizer**

Calcolare il numero di occorrenze di un tokens e  frequenze può essere molto utile, ma una limitazione di questi metodi è che il vocabolario può diventare molto grande.

Questo, a sua volta, richiederà grandi vettori per la codifica dei documenti e imporrà grandi requisiti alla memoria e rallenterà gli algoritmi.

Si può utilizzare un **hash** unidirezionale per convertirle in numeri interi.

Il ***vantaggio*** è che non è richiesto alcun vocabolario e si può scegliere un vettore a lunghezza fissa arbitraria.

Lo ***svantaggio*** è che l'***hash*** è una funzione a senso unico, quindi non c'è modo di convertire la codifica in una parola (che potrebbe non avere importanza per molte attività di apprendimento supervisionate).

La funzione di hashing impiegata è la versiona signed a 32 Bit dell'algoritmo **Murmurhash3.**

# Scikit-Learn: Text Processing

**Hashing with HashingVectorizer**

La classe ***HashingVectorizer*** implementa questo approccio che può essere utilizzata per parole “hash coerenti”, quindi tokenizza e codifica i documenti secondo necessità.

Scegliamo una dimensione vettoriale arbitraria a lunghezza fissa pari a 20 (default 2^20). Ciò corrisponde all'intervallo della funzione hash, in cui valori grandi (come 2^20) possono provocare collisioni di hash.

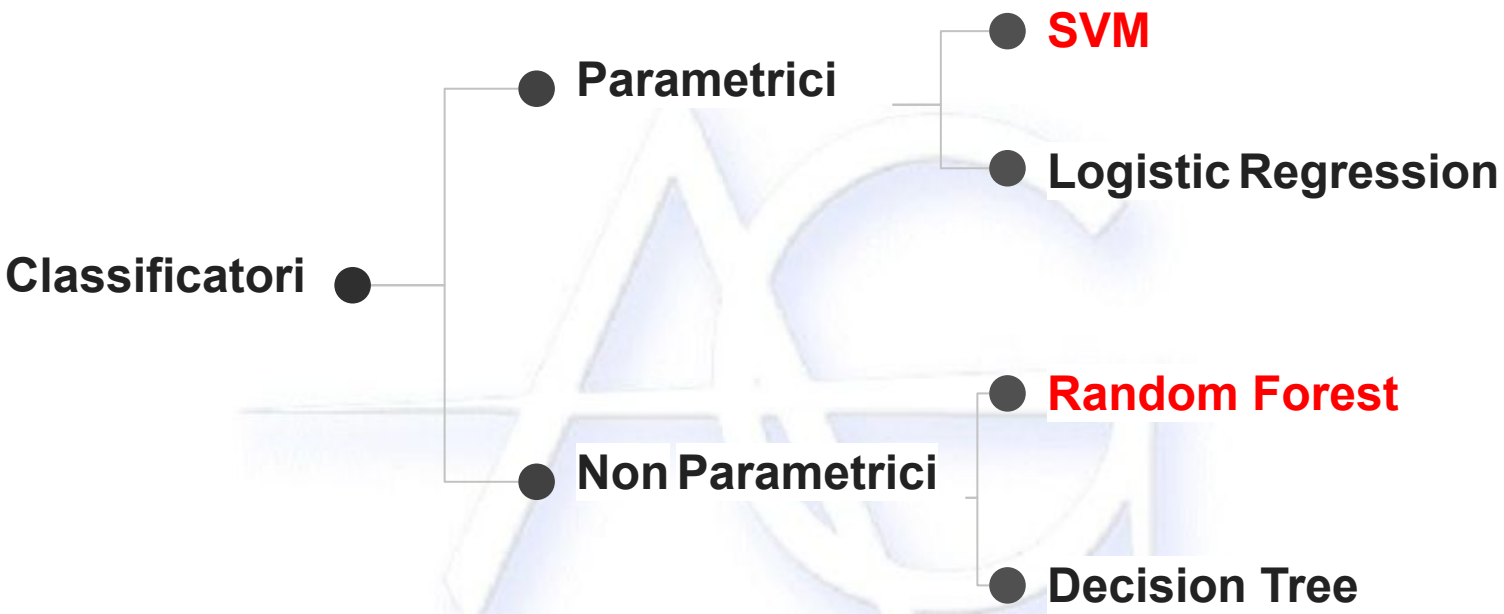
Per ogni termine viene calcolato l’hash e vengono poi contati gli hash uguali (stesso termine a meno di collisioni).

Si noti che questo vettore non richiede una chiamata per adattarsi ai documenti dei dati di addestramento. Invece, dopo l'istanziazione, può essere utilizzato direttamente per avviare la codifica dei documenti.

# Scikit-Learn: Text Processing

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> text = ["The quick brown fox jumped over the lazy dog."]
>>> vectorizer = HashingVectorizer(n_features=20)
>>> vector = vectorizer.transform(text)
>>> print(vector.shape)
(1, 20)
>>> print(vector.toarray())
[[ 0.         0.         0.         0.         0.         0.33333333
   0.        -0.33333333  0.33333333  0.         0.         0.33333333
   0.         0.         0.        -0.33333333  0.         0.
  -0.66666667  0.        ]]
```

# Scikit-Learn: Text Processing



# Scikit-Learn: Support Vector Machine

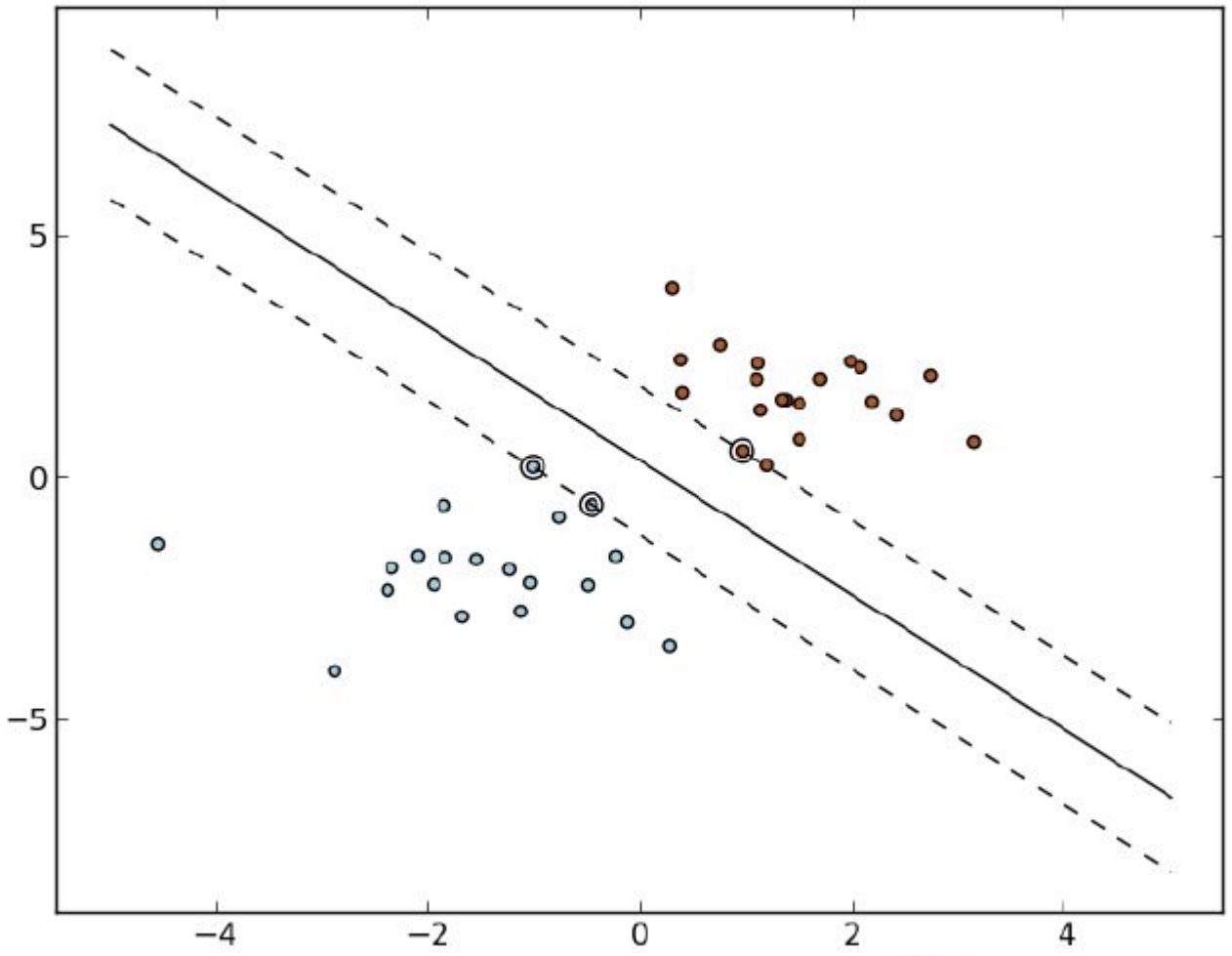
I Support Vector Machines (SVM) sono algoritmi di apprendimento che analizzano i dati al fine di massimizzare i margini di classificazione.

Dato un insieme di esempi, ciascuno contrassegnato come appartenente a una categoria distinta, l'algoritmo di addestramento SVM crea un modello che assegna nuovi esempi ad una categoria, rendendolo un classificatore lineare binario non-probabilistico.

Un SVM costruisce un iperpiano (o un insieme di iperpiani) in uno spazio ad alta dimensione.

Intuitivamente, una buona separazione viene raggiunta dall'iperpiano che ha la **maggiore distanza** dai punti dati di addestramento più vicini di qualsiasi classe (il cosiddetto margine funzionale), poiché in generale maggiore è il margine più basso è l'errore di generalizzazione del classificatore.

# Scikit-Learn: Support Vector Machine



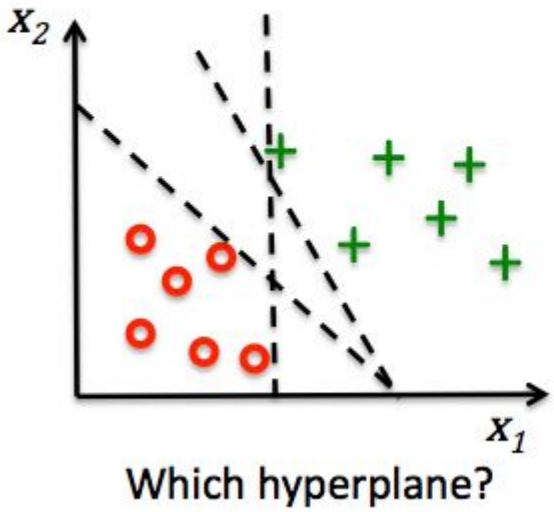
# Scikit-Learn: Support Vector Machine

- ***SVC***, ***NuSVC*** e ***LinearSVC*** sono classi in grado di eseguire una classificazione multi-classe su un insieme di dati.

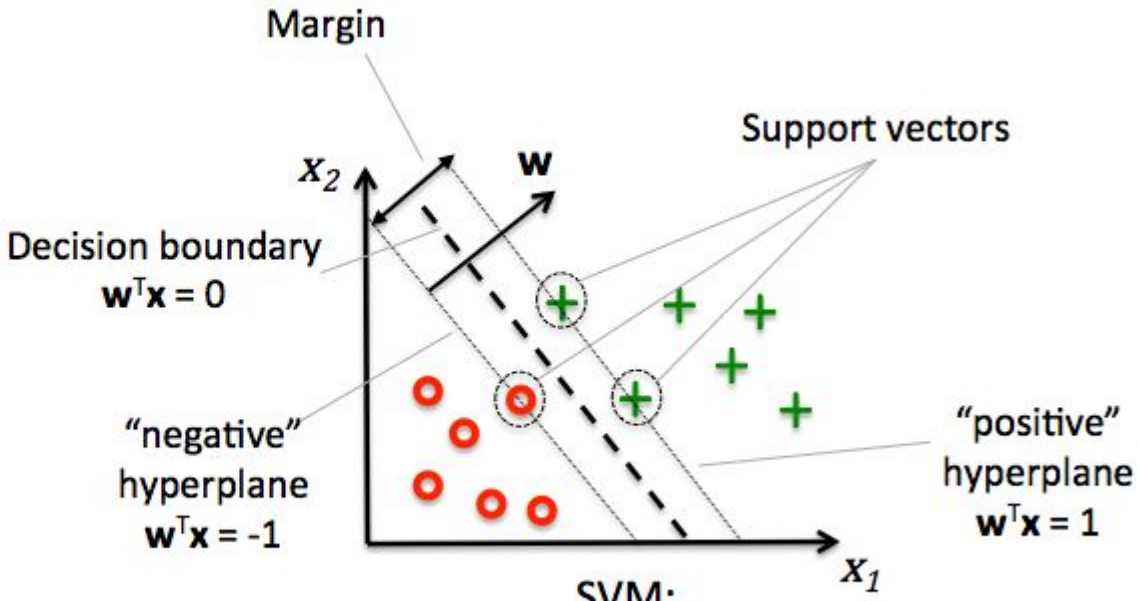
- ***SVC*** e ***NuSVC*** sono metodi simili, ma accettano serie di parametri leggermente differenti e hanno diverse formulazioni matematiche (documentazione qui).

- ***LinearSVC*** è un'altra implementazione di ***Support Vector Classification*** per il caso di un ***kernel*** lineare.

# Scikit-Learn: Support Vector Machine




Which hyperplane?




SVM:
Maximize the margin

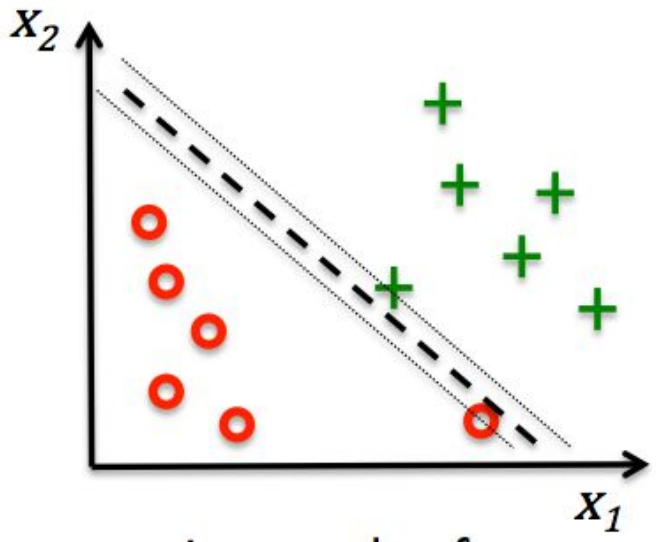
$$\mathbf{w}^T\mathbf{x}^{(i)} \geq 1 - \xi^{(i)} \text{ if } y^{(i)} = 1$$

$$\mathbf{w}^T\mathbf{x}^{(i)} < -1 + \xi^{(i)} \text{ if } y^{(i)} = -1$$

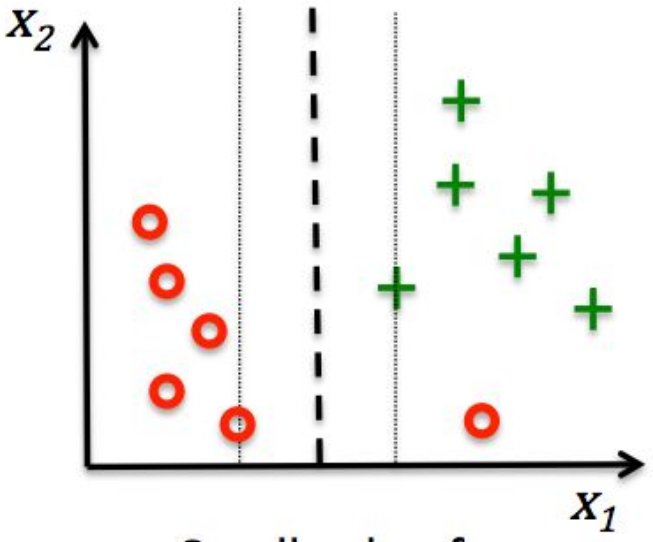
$$\frac{1}{2}\|\mathbf{w}\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

# Scikit-Learn: Support Vector Machine

Il parametro C controlla la penalizzazione dell'errata classificazione, valori alti corrispondono ad una penalizzazione maggiore. Il valore di C è un compromesso tra bias e varianza.




Large value for parameter *C*




Small value for parameter *C*

# Scikit-Learn: Support Vector Machine

Esempio di classificazione

```
from sklearn import svm
X = [[0, 0], [1, 1]]
y = [0, 1]
clf = svm.SVC()
clf.fit(X, y)
print(clf.predict([[2., 2.]]))
>>> [1]
```

# Scikit-Learn: Support Vector Machine

La funzione decisionale di un SVM dipende da alcuni sottoinsiemi dei dati di addestramento, chiamati vettori di supporto.

Alcune proprietà di questi vettori di supporto possono essere trovate con i seguenti metodi:

```
X = [[0, 0], [1, 1]]

y = [0, 1]

clf = svm.SVC()

clf.fit(X, y)

print(clf.predict([[2., 2.]]))

print(clf.support_vectors_ #  restituisce i support vectors: [[0. 0.] [1. 1.]])

print(clf.support_ # restituisce gli indici support vectors: [0 1])

print(clf.n_support_ # restituisce il numero di support vectors per classe: [1 1])
```

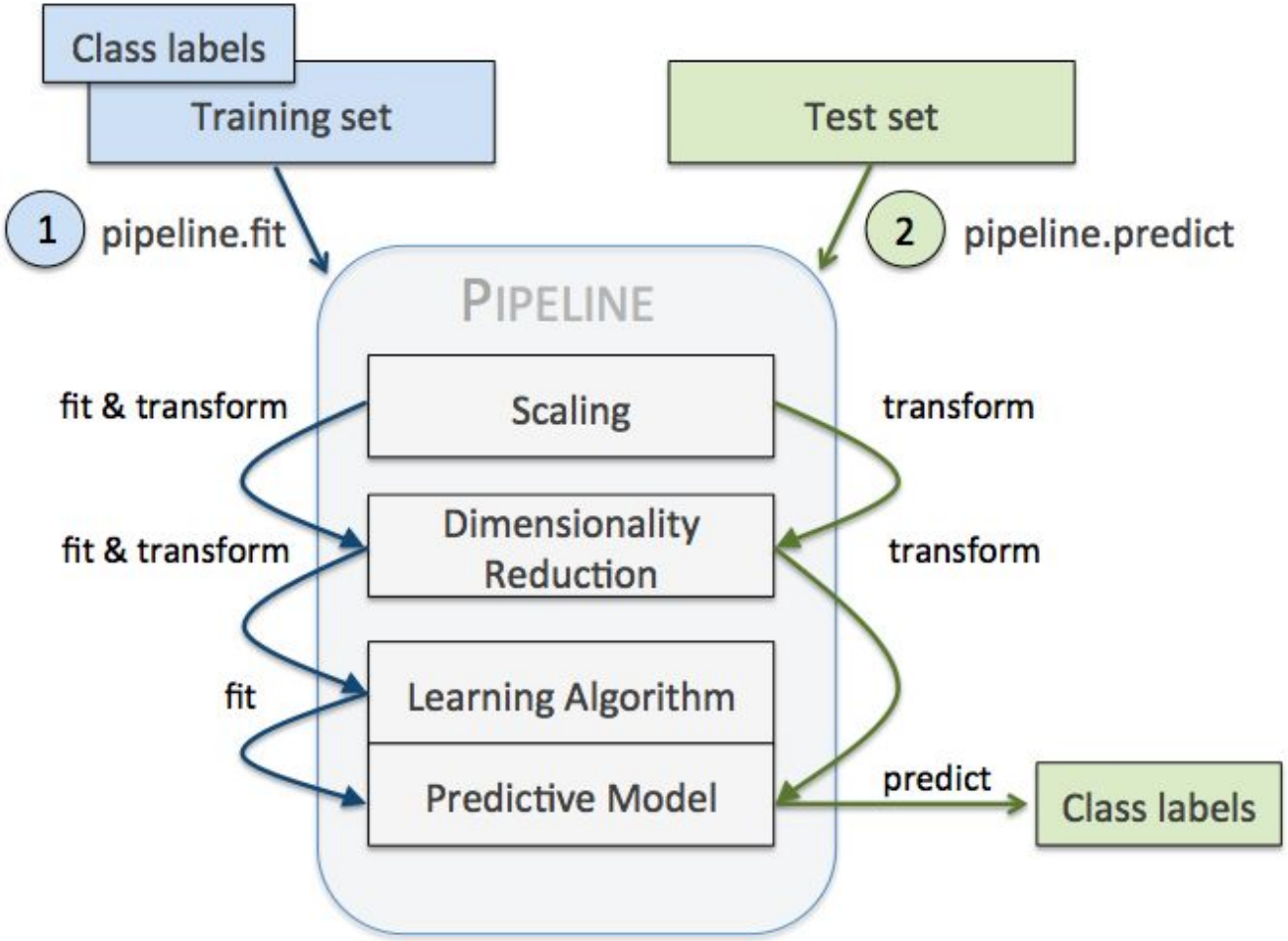
# Scikit-Learn: Support Vector Machine

Classificazioni multi-classe: SVC e NuSVC implementano il "one-against-one" approccio per la classificazione multi-classe. Se C è il numero di classi, allora I classificatori C (C - 1) / 2 sono costruiti e ognuno elabora i dati di due classi:

```
X = [[0], [1], [2], [3]]
Y = [0, 1, 2, 3]
clf = svm.SVC()
clf.fit(X, Y)
dec = clf.decision_function([[0.3]])
print(dec.shape[1])  # 4 classes
print(clf.predict([[0.3]])) # 0
```

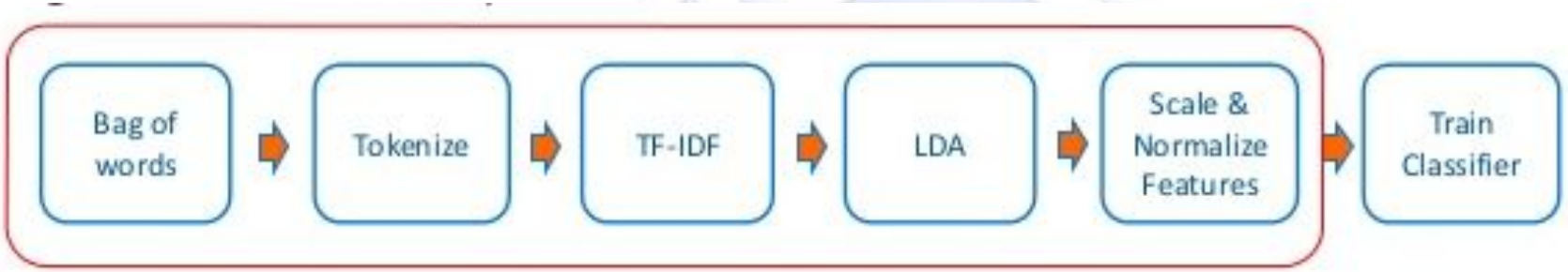
# Scikit-Learn: Support Vector Machine

Pipelines



# Scikit-Learn: Support Vector Machine

Pipelines



# Scikit-Learn: Support Vector Machine

Creiamo una ***pipeline*** per un full-processing che, partendo da testo semplice, permette di classificare le recensioni dei film presenti nel ***movie_reviews*** corpus.

```
from nltk.corpus import movie_reviews
documents = [(movie_reviews.raw(fileid), category)
        for category in movie_reviews.categories()
        for fileid in movie_reviews.fileids(category)]
random.shuffle(documents)
documents = documents[:100]
```

Prendiamo le prime 100 reviews per abbreviare i tempi di calcolo.

# Scikit-Learn: Support Vector Machine

Utilizziamo la funzione ***train_test_split*** di sklearn per ottenere train e test set.

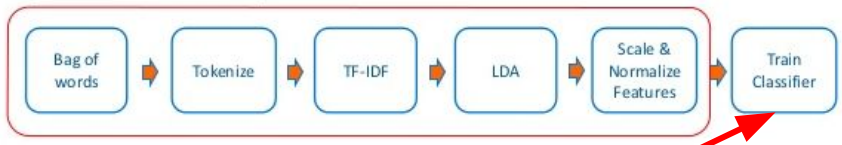
```
xData = [doc[0] for doc in documents]
yData = [doc[1] for doc in documents]
xTrain, xTest, yTrain, yTest = train_test_split (
   xData, yData,
   test_size=0.33,
   random_state=42
)
```

***test_size*** è la percentuale di suddivisione tra train e test set, in questo caso 70 e 30.

# Scikit-Learn: Support Vector Machine

Ora costruiamo la nostra pipeline.

```
classifier = Pipeline([
  ('feature_vect',  TfidfVectorizer(strip_accents='unicode',
                    tokenizer=word_tokenize,
                    stop_words='english',
                    decode_error='ignore',
                    analyzer='word',
                    norm='l2',
                    ngram_range=(1, 2)
                    )),
  ('clf', SVC(probability=True, C=10, shrinking=True, kernel='linear'))
])
```



# Scikit-Learn: Support Vector Machine

Ora possiamo addestrare il nostro classificatore:

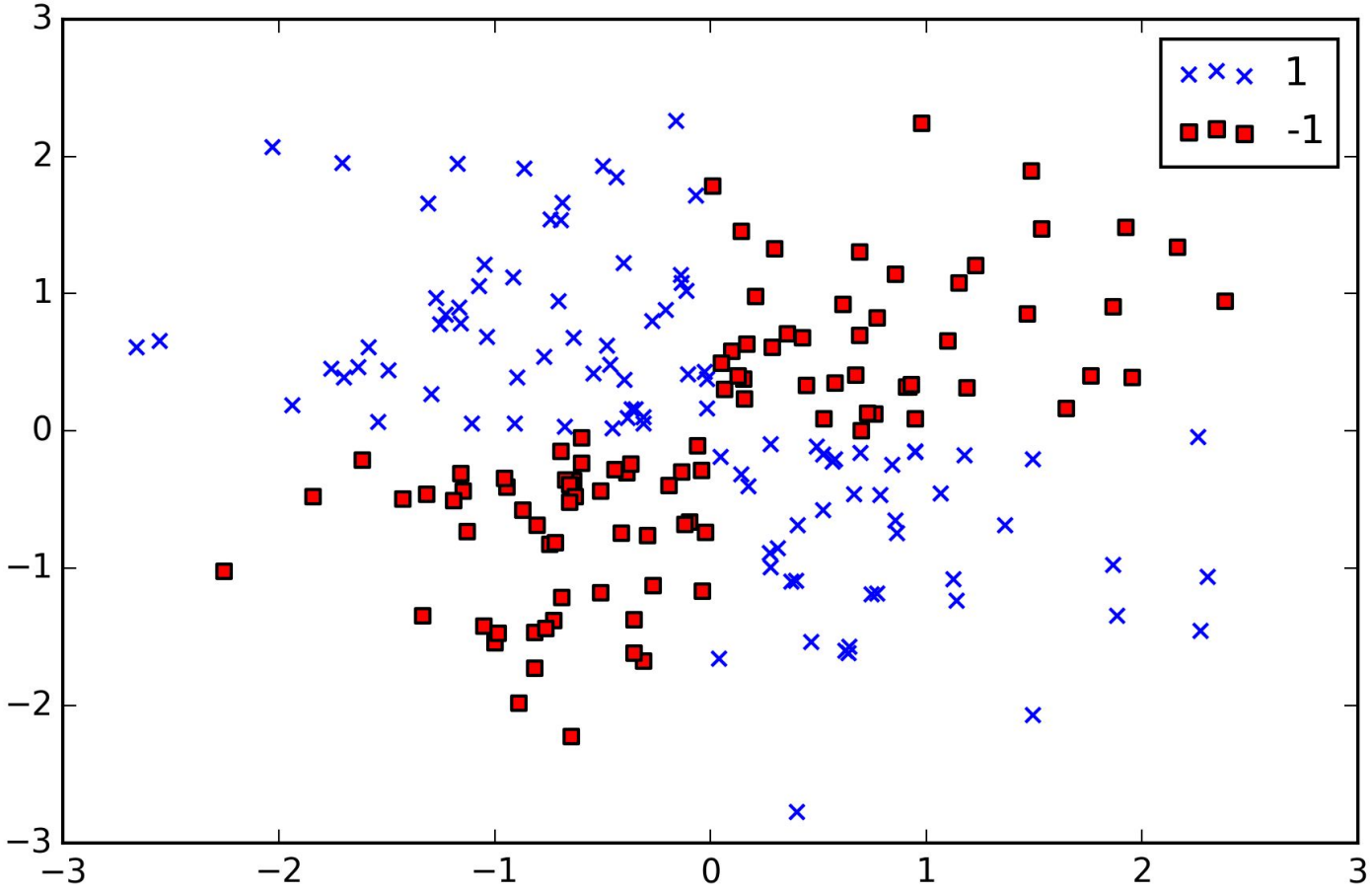
```
classifier.fit(xTrain, yTrain)
```

e verificare le performance.

```
predicted = classifier.predict(xTest)
print(accuracy_score(yTest, predicted))
print(precision_recall_fscore_support(yTest, predicted))
print(classification_report(yTest, predicted))
```

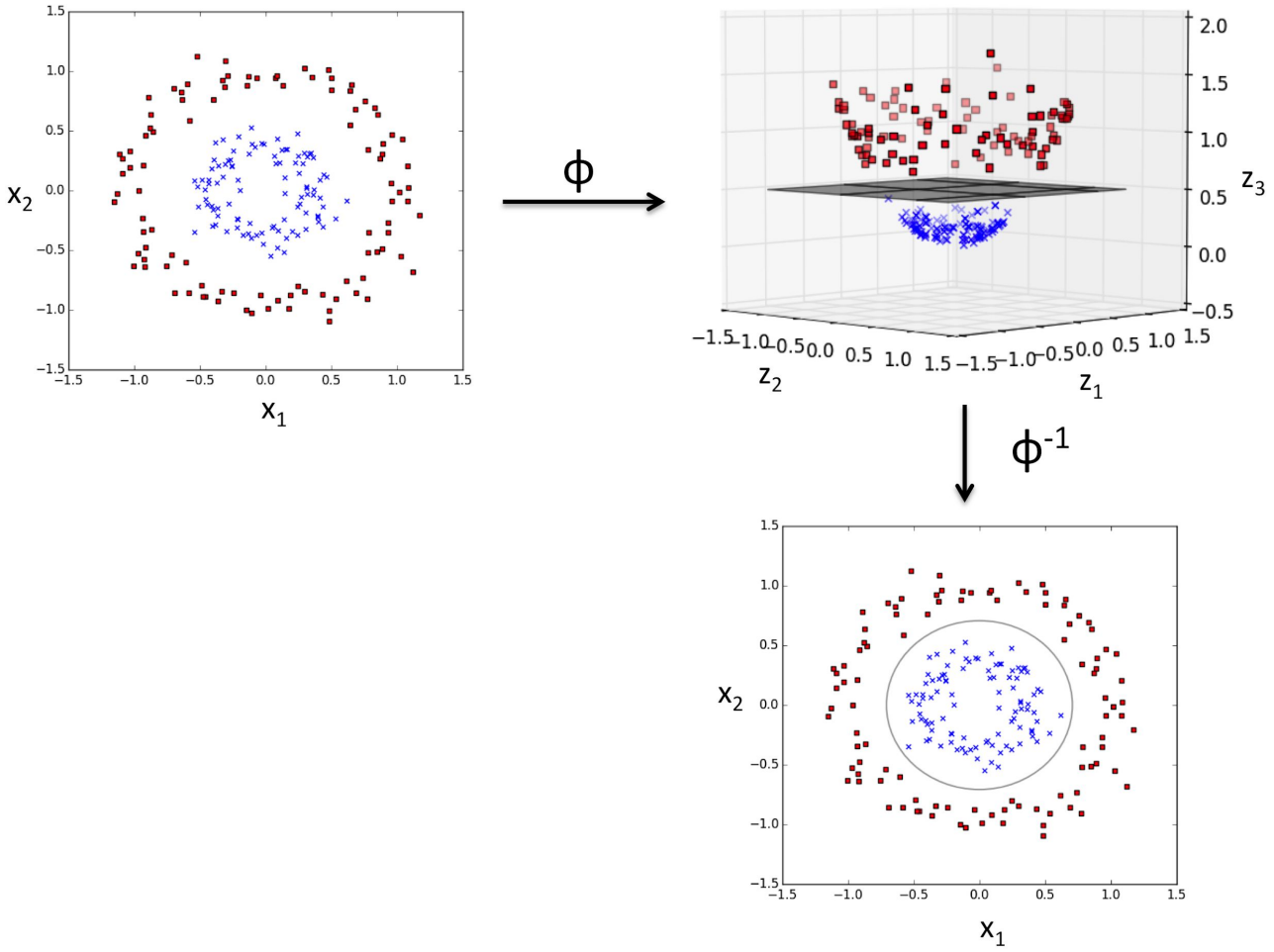
# Scikit-Learn: Support Vector Machine

Classi non separabili linearmente e kernel.



# Scikit-Learn: Support Vector Machine

Aumento di dimensioni, iperpiano di separazione e funzione kernel (similarità kernel).



# Scikit-Learn: Support Vector Machine

Kernel Radial Basis Function (RBF) o *gaussiano:*

$$k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \phi\left(\mathbf{x}^{(i)}\right)^T \phi\left(\mathbf{x}^{(j)}\right)$$

$$k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \exp\left( - \frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2} \right)$$

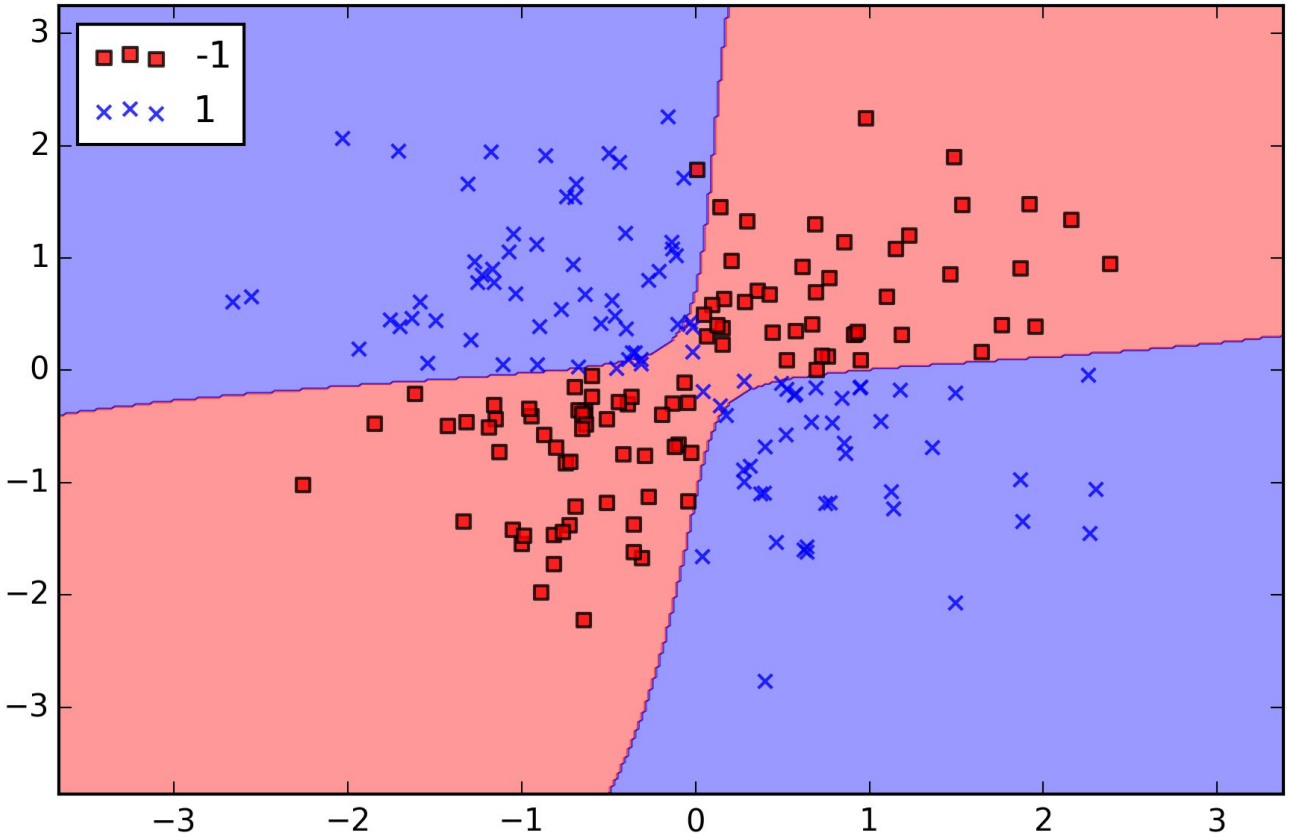
$$k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \exp\left( - \gamma \, \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2 \right)$$

Il parametro gamma γ controlla l'ampiezza della campana gaussiana nell'iperpiano di proiezione.

# Scikit-Learn: Support Vector Machine

Kernel Radial Basis Function (RBF) o *gaussiano:*

```
svm = SVC(kernel='rbf', random_state=0, gamma=0.10, C=10.0)
```

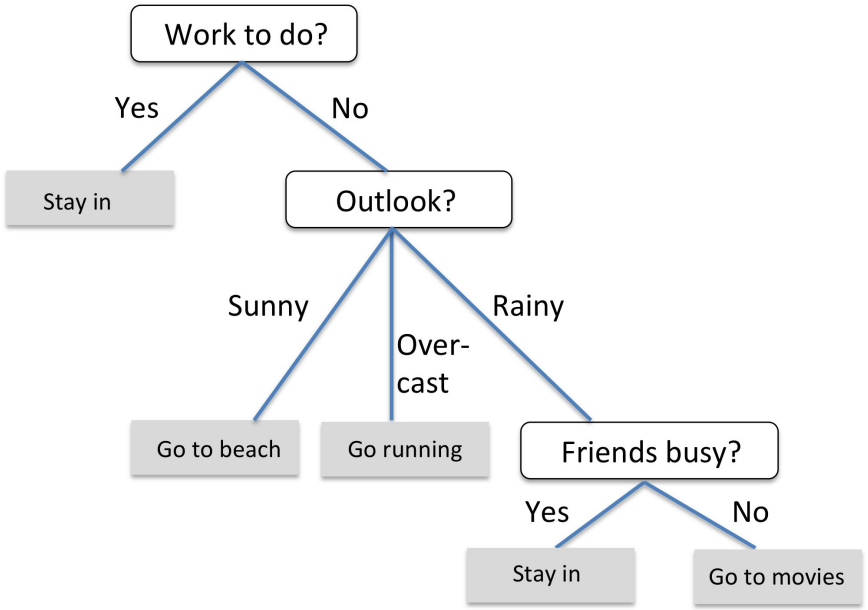


# Scikit-Learn: Decision Tree

Gli alberi decisionali sono algoritmi di classificazione **non parametrici** che ad ogni nodo prendono una "**decisione**" su quale **caratteristica** del dataset sia quella che meglio **discrimina** le due classi.

In maniera **iterattiva** si costruisce l'albero andando a creare nuovi nodi fino ad ottenere un albero le cui foglie sono costituite da insiemi omogenei di campioni appartenenti alla stessa classe.

In generale questo può portare ad alberi molto profondi (**overfitting**), generalmente si ottimizzano poi con operazioni di potatura (**pruning**)



# Scikit-Learn: Decision Tree

Per poter decidere quale, ad ogni iterazione, sia la caratteristica più discriminante bisogna definire una funzione obiettivo da ottimizzare con l'algoritmo di addestramento dell'albero. Si utilizza il concetto di ***guadagno di informazione*** che deve essere massimo ad ogni suddivisione successiva.

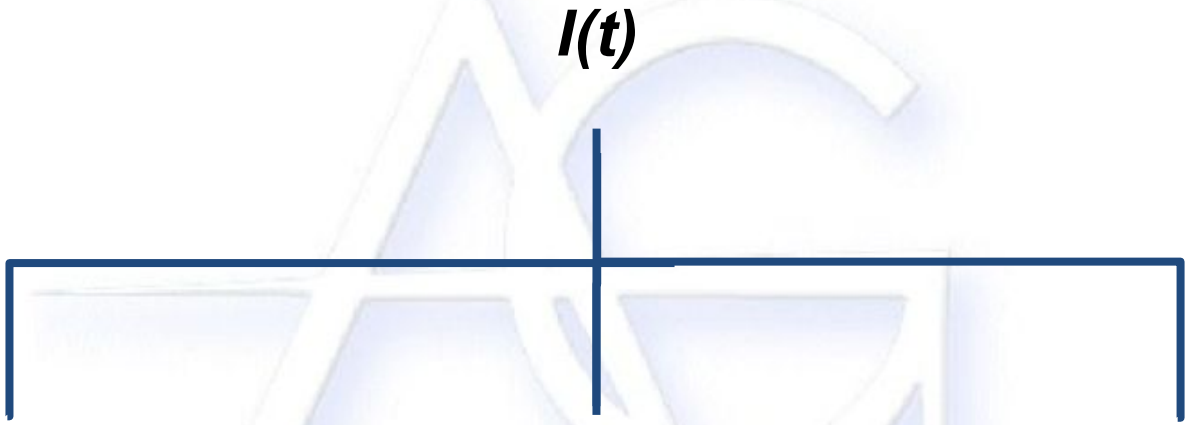
$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

Dato un nodo padre $D_p$ con *m* figli e la caratteristica *f*, l'**Information Gain** è dato dalla differenza tra l'***impurità*** $I(D_p)$ del nodo padre e la somma delle ***impurità*** dei figli $I(D_j)$. La funzione I è la misura di impurità e $N_p$ e $N_j$ sono il numero di campioni nel nodo padre e nel nodo figlio j-esimo. Per limitare la complessità Scikit-Learn utilizza alberi binari:

$$IG(D_p, f) = 1(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

# Scikit-Learn: Criterion

La libreria Scikit-Learn definisce tre misure di impurità meglio detti **criteri di suddivisione**.




$I_H(t)$ $I_G(t)$ $I_E(t)$

***Entropia*** ***Impurità di Gini*** **Errore di Classificazione**

# Scikit-Learn: Criterion

La libreria Scikit-Learn definisce tre misure di impurità meglio detti **criteri di suddivisione**:

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t)$$

$$I_G(t) = \sum_{i=1}^{c} p(i|t)(1 - p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

$$I_E(t) = 1 - max\{p(i|t)\}$$

# Scikit-Learn: Entropy

L'entropia del nodo t è definita sulla base di ***p(i|t)*** che rappresenta la proporzione di campioni che appartengono alla classe ***c***:

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t)$$

Questa definizione deriva dal **primo teorema di Shannon** fondamento della teoria dell'informazione.

$$E(X) = -\sum_{i=1}^{n} P(X = x_i) \log_2 (P(X = x_i))$$

L'entropia vale 0 se tutti i campioni di un nodo appartengono alla classe ***c*** ed è massima quando i campioni sono uniformemente distribuiti tra i nodi. Nel caso binario ***p(i=1|t)=1*** o ***p(i=0|t)=0*** allora $I_H=0$, se invece ***p(i=0|t)=p(i=0|t)=0.5*** allora $I_H=1$.

# Scikit-Learn: Gini's Impurity

L'impurità di Gini è il criterio predefinito per la maggior parte degli alberi decisionali d Scikit-Learn e si basa sull'idea di minimizzare la probabilità di errata classificazione.

$$I_G(t) = \sum_{i=1}^{c} p(i|t)(1 - p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

Come per l'entropia è massima quando le classi sono perfettamente mescolate (uniformemente distribuite) nei nodi, per ***c=2*** allor $I_G(t)=1-(0.5^2+0.5^2)=0.5$.

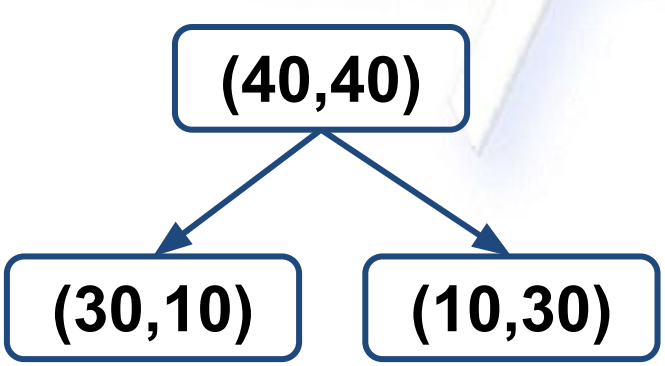
In generale il risultato è molto simile all'entropia ed è in genere indicata per la generazione dell'albero e meno per la potatura.

# Scikit-Learn: Classification Error

Utile per la potatura o pruning di un albero decisionale ma meno indicato per la sua crescita perchè risulta meno sensibile ai cambiamenti delle probabilità che i nodi appartengano a determinate classi.

$$I_E(t) = 1 - max\{p(i|t)\}$$

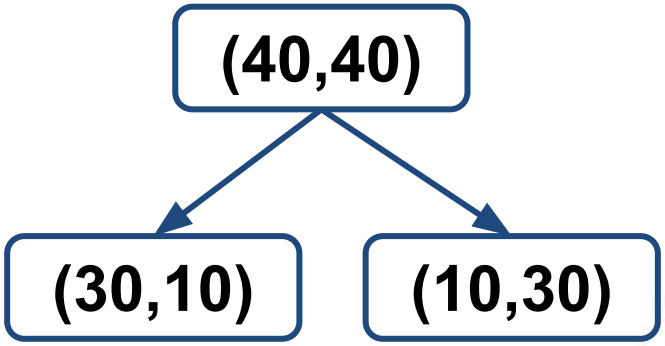
Consideriamo per esempio i due casi A e B seguenti.




A


B

# Scikit-Learn: Classification Error

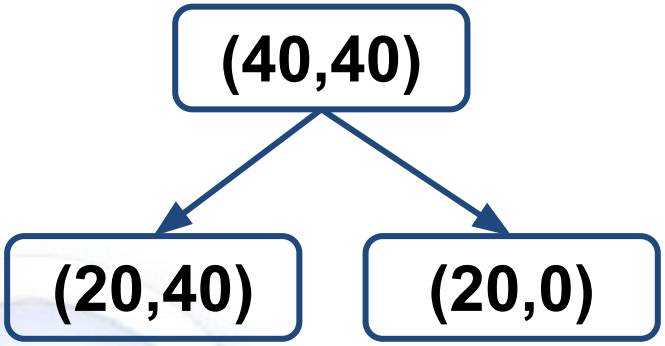



A

$I_E$=0.25

$I_G$=0.125

$I_H$=0.19



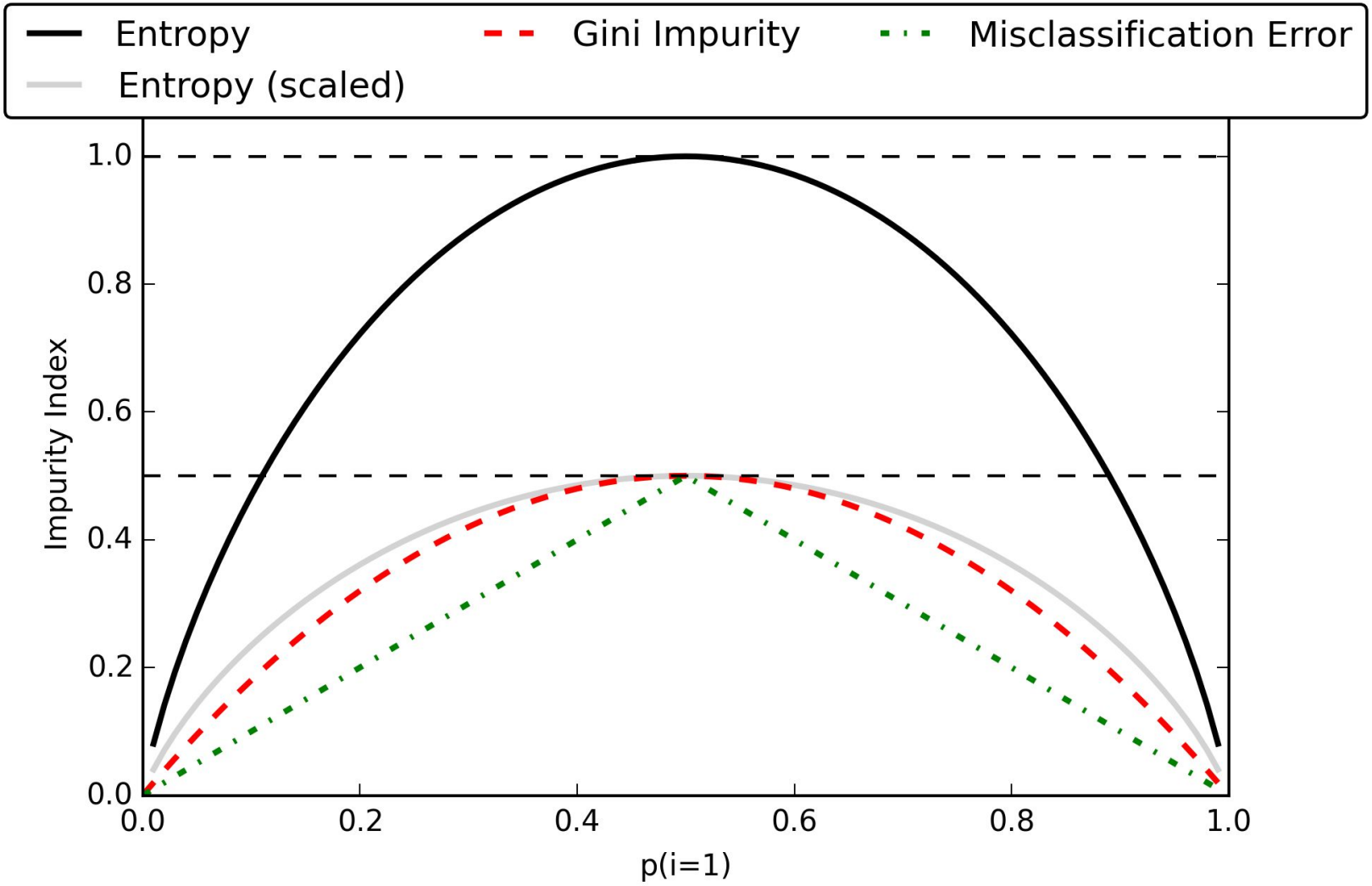

B

$I_E$=0.25

$I_G$=0.16

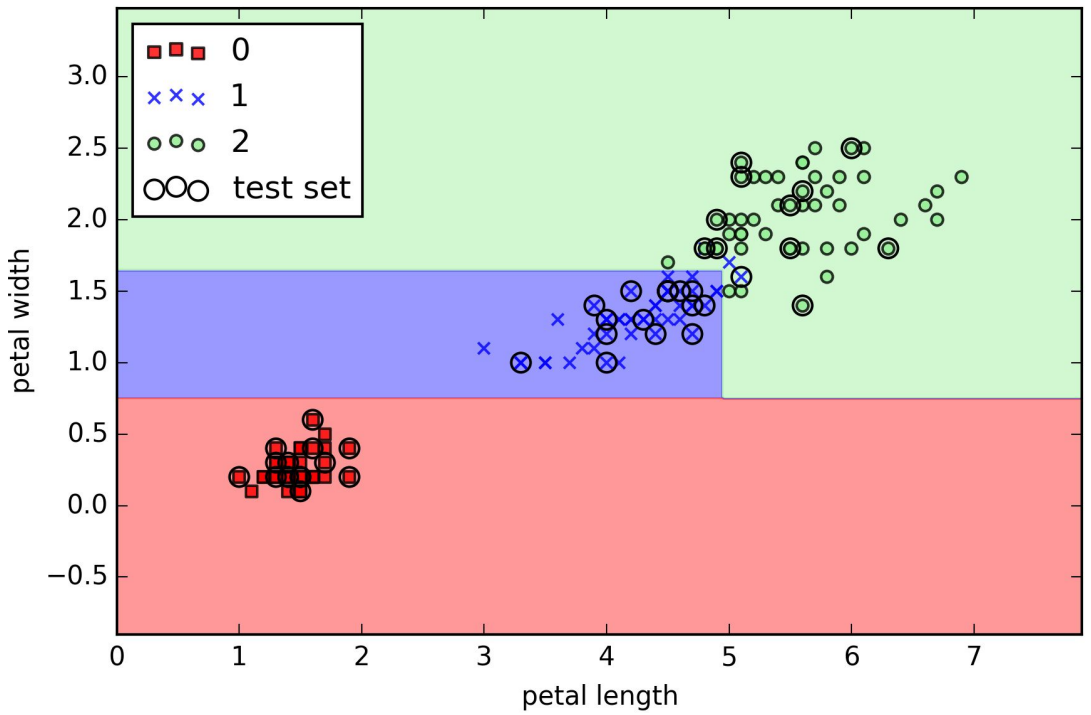
$I_H$=0.31

# Scikit-Learn: Criterion

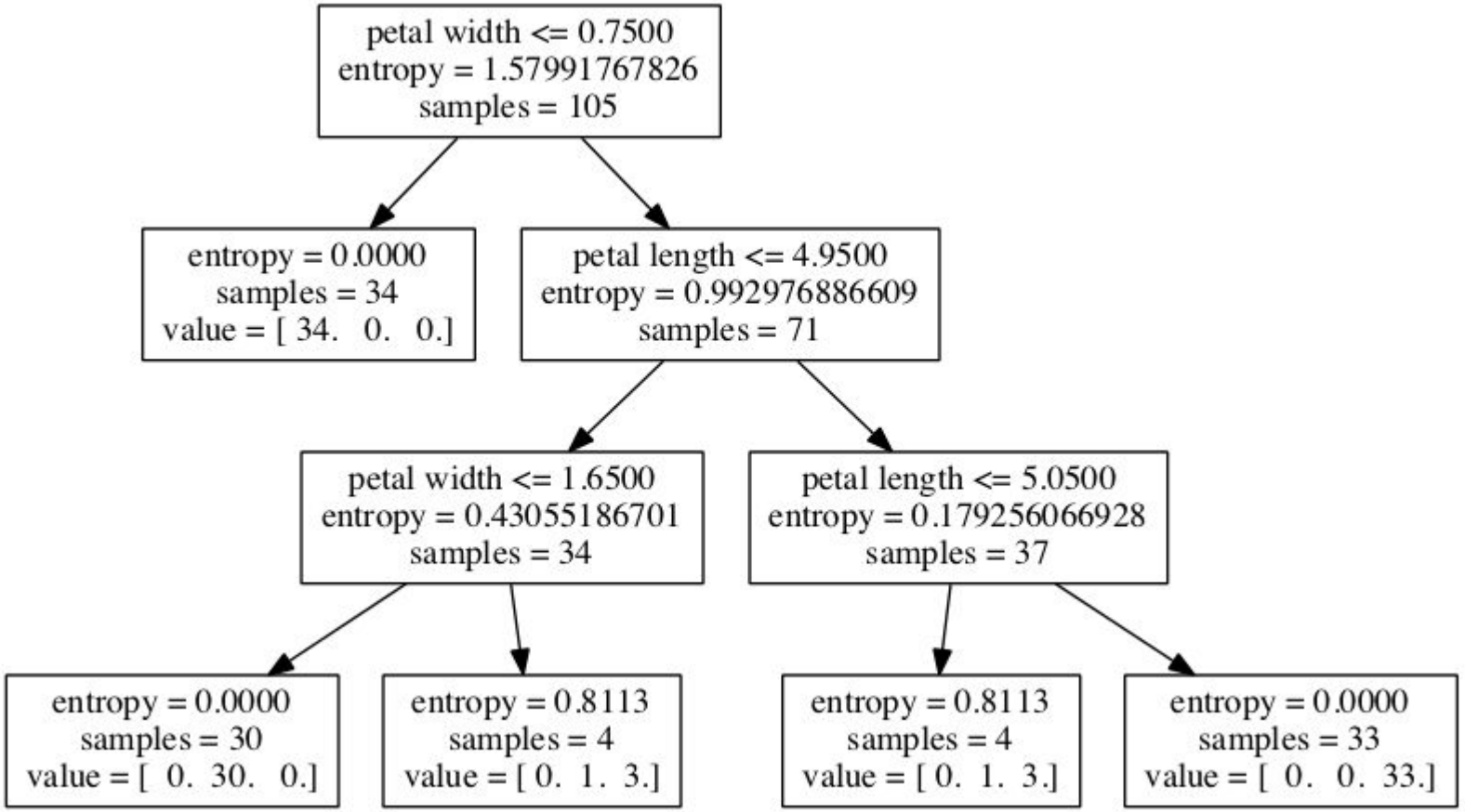


# Scikit-Learn: Decision Tree

```
iris = datasets.load_iris()

x = iris.data[:, [2, 3]]

y = iris.target

x_train, x_test, y_train, y_test = train_test_split( x, y, test_size=0.3, random_state=0)

tree = DecisionTreeClassifier(criterion=criterion, max_depth=3, random_state=0)

tree.fit(x_train, y_train)
```



# Scikit-Learn: Decision Tree



# Scikit-Learn: Random Forest

L'algoritmo è basato su alberi decisionali randomizzati. Si adatta a un numero

di classificatori dell'albero decisionale su vari sottocampioni del set di dati e

utilizza la media per migliorare l'accuratezza predittiva del classificatore.

Ogni albero nell'insieme è costruito da un campione con sostituzione (ad esempio, un campione di bootstrap) dal training set.

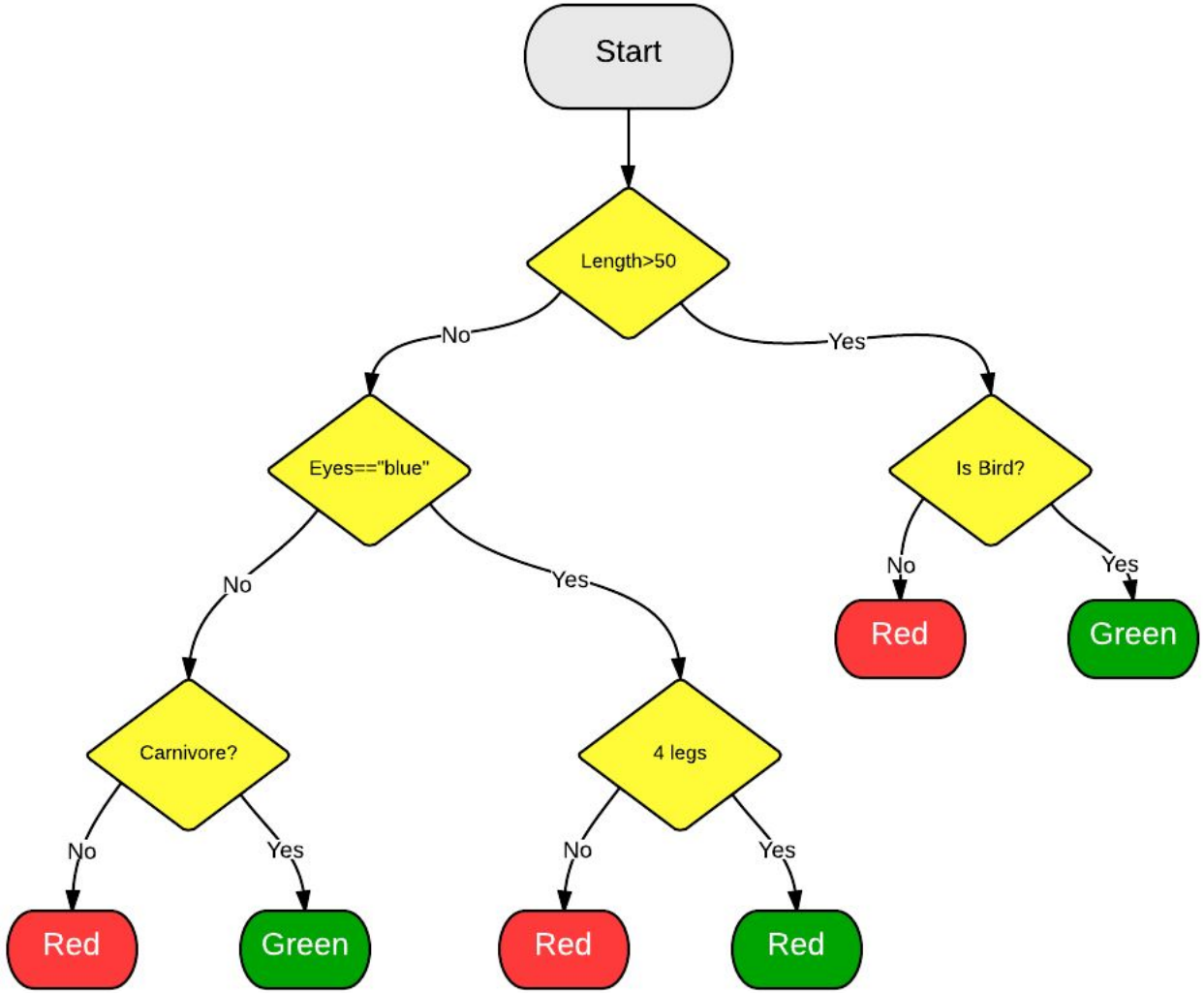
Fa parte degli algoritmi ***Ensemble Learning*** e ciò comporta la combinazione di diversi modelli per risolvere un singolo problema di predizione.

Funziona generando più classificatori che imparano e fanno previsioni in modo indipendente.

Tali previsioni sono quindi combinate in una singola (*super*) previsione che *dovrebbe* essere migliore della previsione fatta da un qualsiasi classificatore.

Non richiede che le features siano normalizzate a differenza dei classificatori basati sulla minimizzazione dei pesi associati agli errori.

# Scikit-Learn: Random Forest



# Scikit-Learn: Random Forest

Una ***Random Forest*** è un'aggregazione di altri modelli, ma quali tipi di modelli aggrega? La Random Forest aggrega Decision Trees. Un albero decisionale è composto da una serie di decisioni che possono essere utilizzate per classificare un'osservazione in un set di dati.

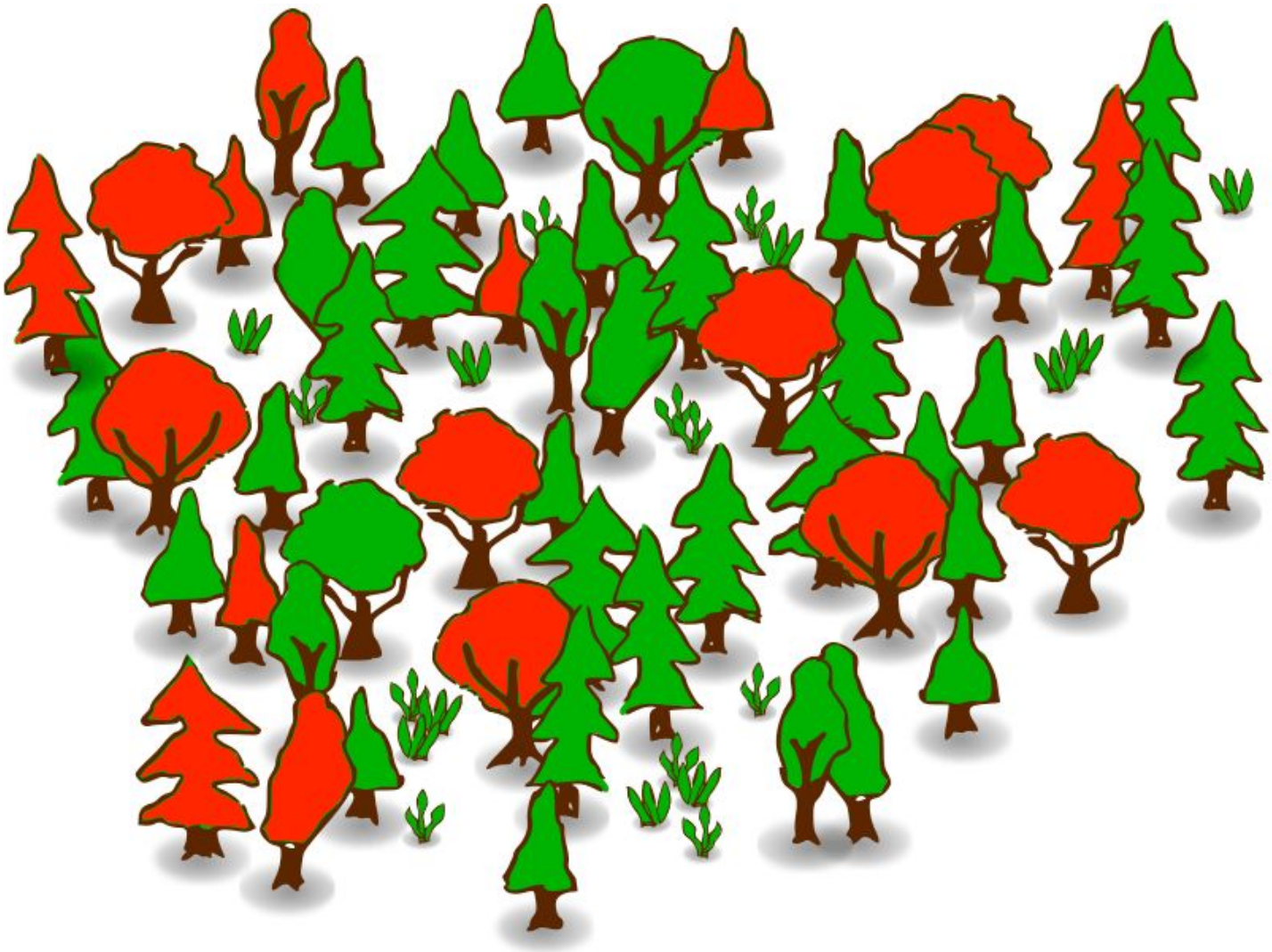
L'algoritmo per generare una ***Random Forest*** creerà automaticamente un insieme di alberi decisionali casuali.

Poiché gli alberi sono generati casualmente, la maggior parte non sarà così significativa per l'apprendimento del problema di classificazione / regressione (forse il 99,9% degli alberi).

Quindi, a che servono 10000 cattivi (probabilmente) modelli ? Si scopre che in realtà non sono così utili.

Ma ciò che è utile sono i ***pochi*** alberi decisionali veramente buoni che ha generato insieme a quelli cattivi.

# Scikit-Learn: Random Forest



# Scikit-Learn: Random Forest

Quando si effettua una previsione, la nuova osservazione viene spostata verso il basso in ogni albero decisionale e viene assegnato un valore / etichetta prevista.

Una volta che ciascuno degli alberi nella foresta ha riportato il valore / etichetta previsti, le previsioni vengono calcolate e il voto di modalità di tutti gli alberi viene restituito come previsione finale.

Semplicemente, il 99,9% degli alberi irrilevanti produce previsioni che si trovano su tutta la mappa e si annullano a vicenda.

Le previsioni della minoranza di alberi sono in cima al rumore e producono una buona previsione.

# Scikit-Learn: Random Forest

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

Parametri:

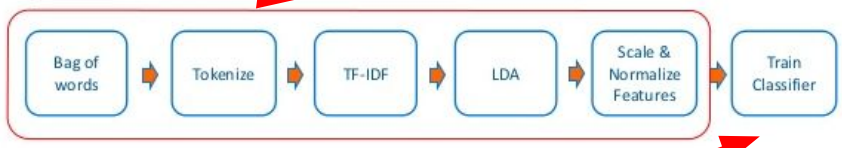
- ***n_estimators*** *(optional, default=10)*: numero di alberi della foresta.
- ***max_features*** *(optional, default='auto')*: numero di feature da considerare per il best split (float, 'sqrt', 'log2', 'None', 'auto')
- ***bootstrap*** (*optional; default=True)*: se utilizzare degli esempi per fare il bootstrap iniziale

# Scikit-Learn: Random Forest

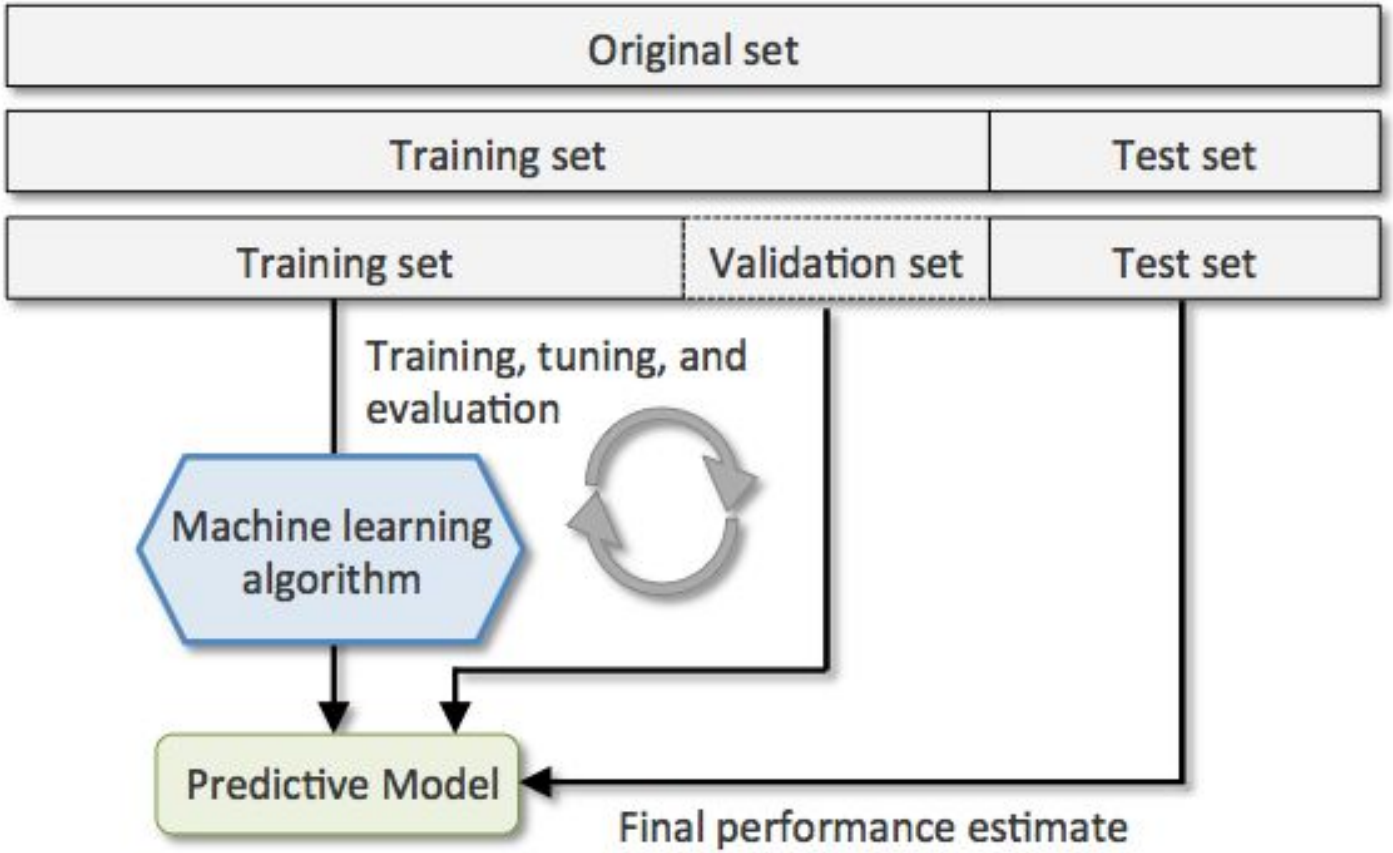
```
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data[:, :2]
y = iris.target
rf = RandomForestClassifier(n_estimators=10)
rf.fit(X, y)
rf.predict([[5.4, 1.5], [3.6, 5.1]])
```

# Scikit-Learn: Random Forest

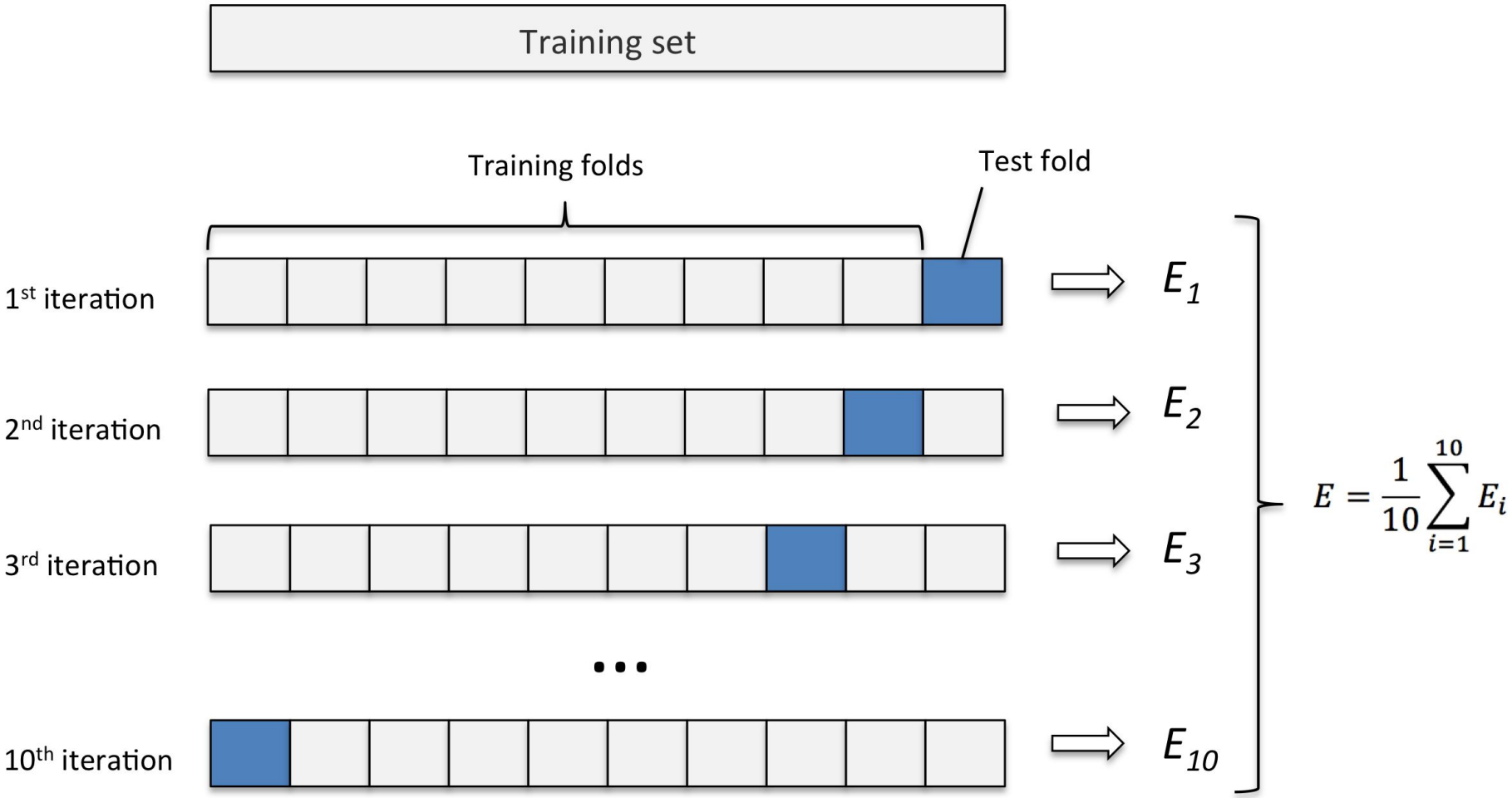
```
classifier = Pipeline([
  ('feature_vect',  TfidfVectorizer(strip_accents='unicode',
                    tokenizer=word_tokenize,
                    stop_words='english',
                    decode_error='ignore',
                    analyzer='word',
                    norm='l2',
                    ngram_range=(1, 2)
                    )),
  ('clf', RandomForestClassifier(n_jobs=2, n_estimators=10))
])
```



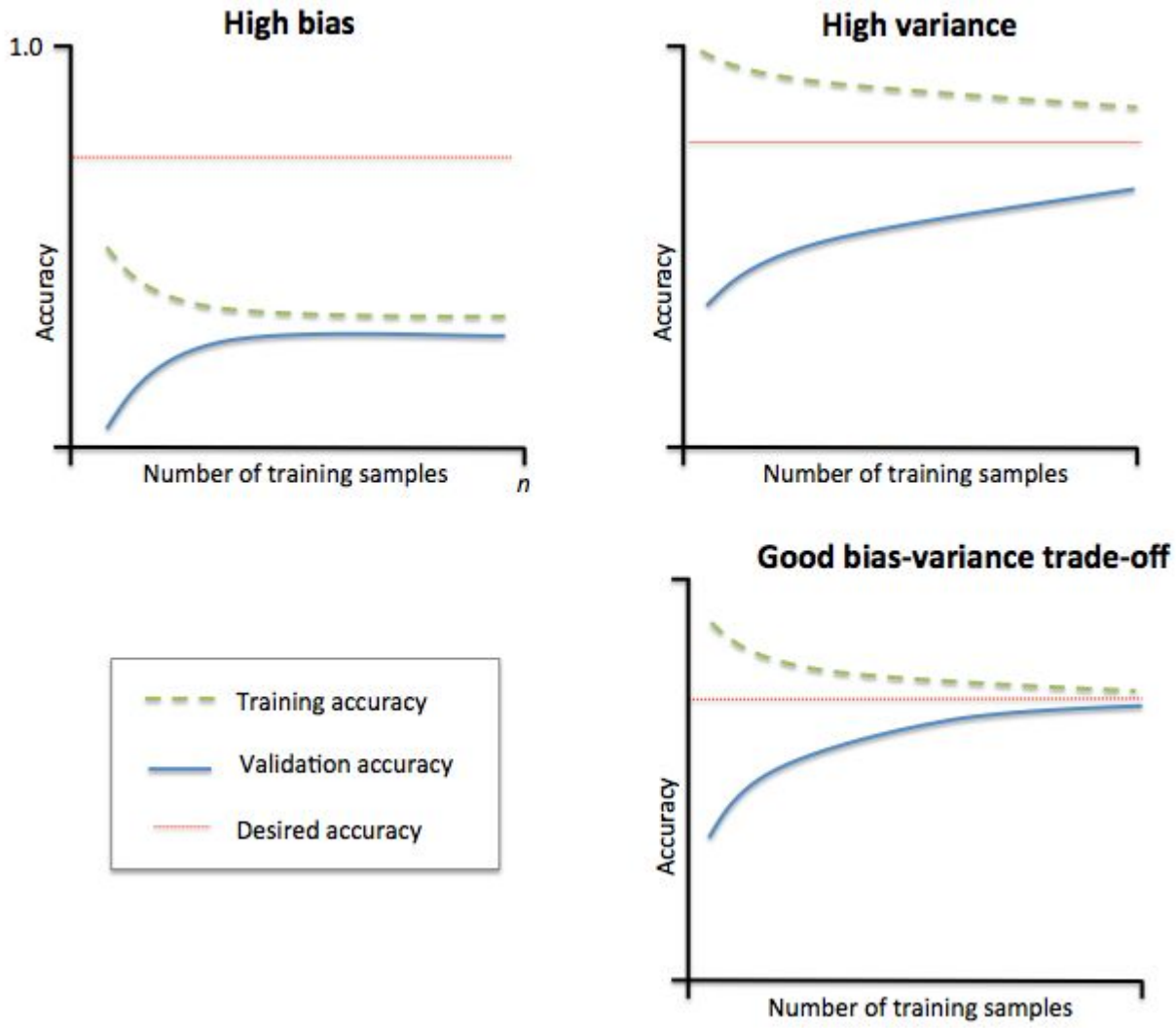
# Scikit-Learn: Ottimizzazione dei parametri



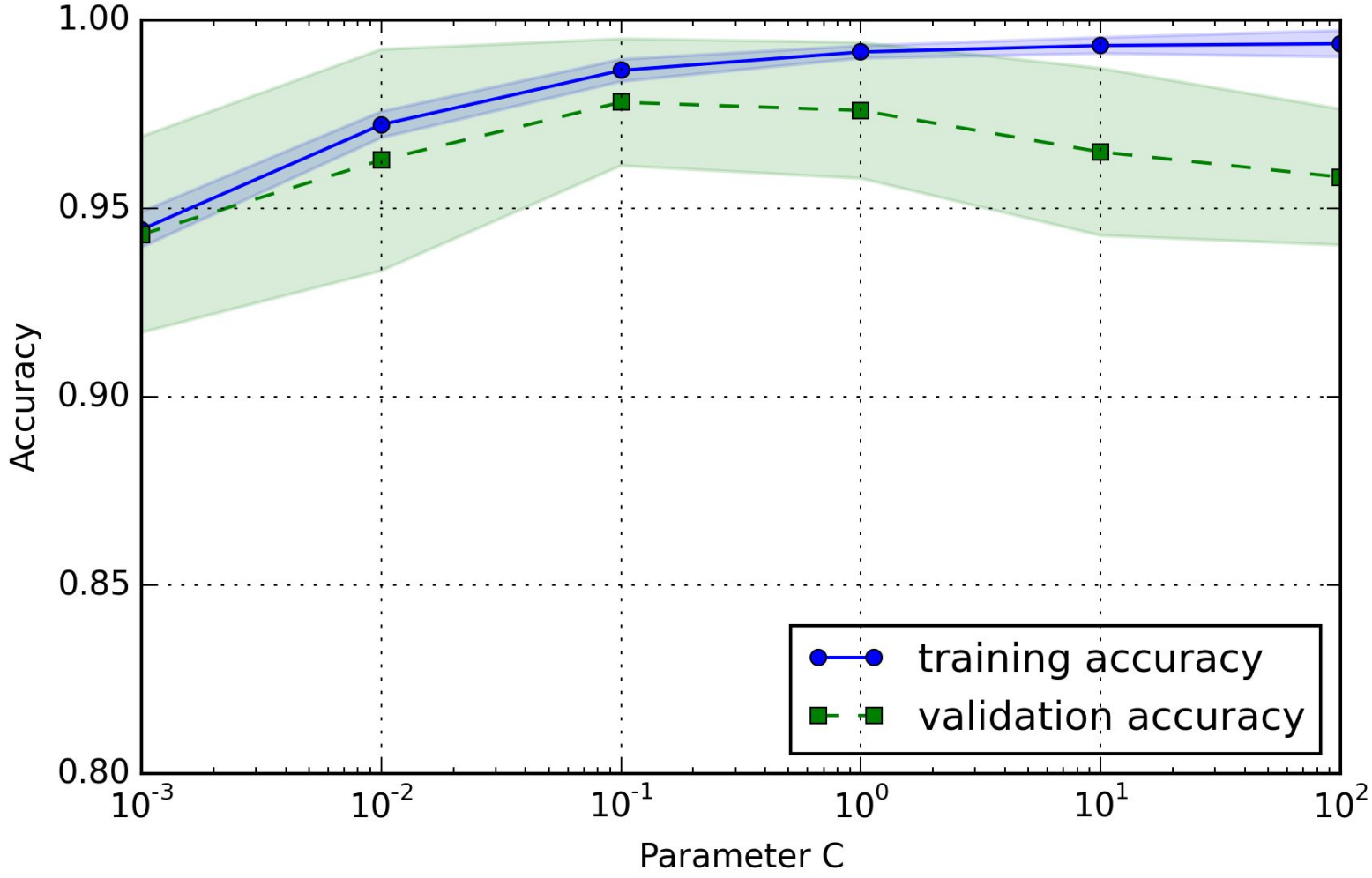
# Scikit-Learn: Ottimizzazione dei parametri



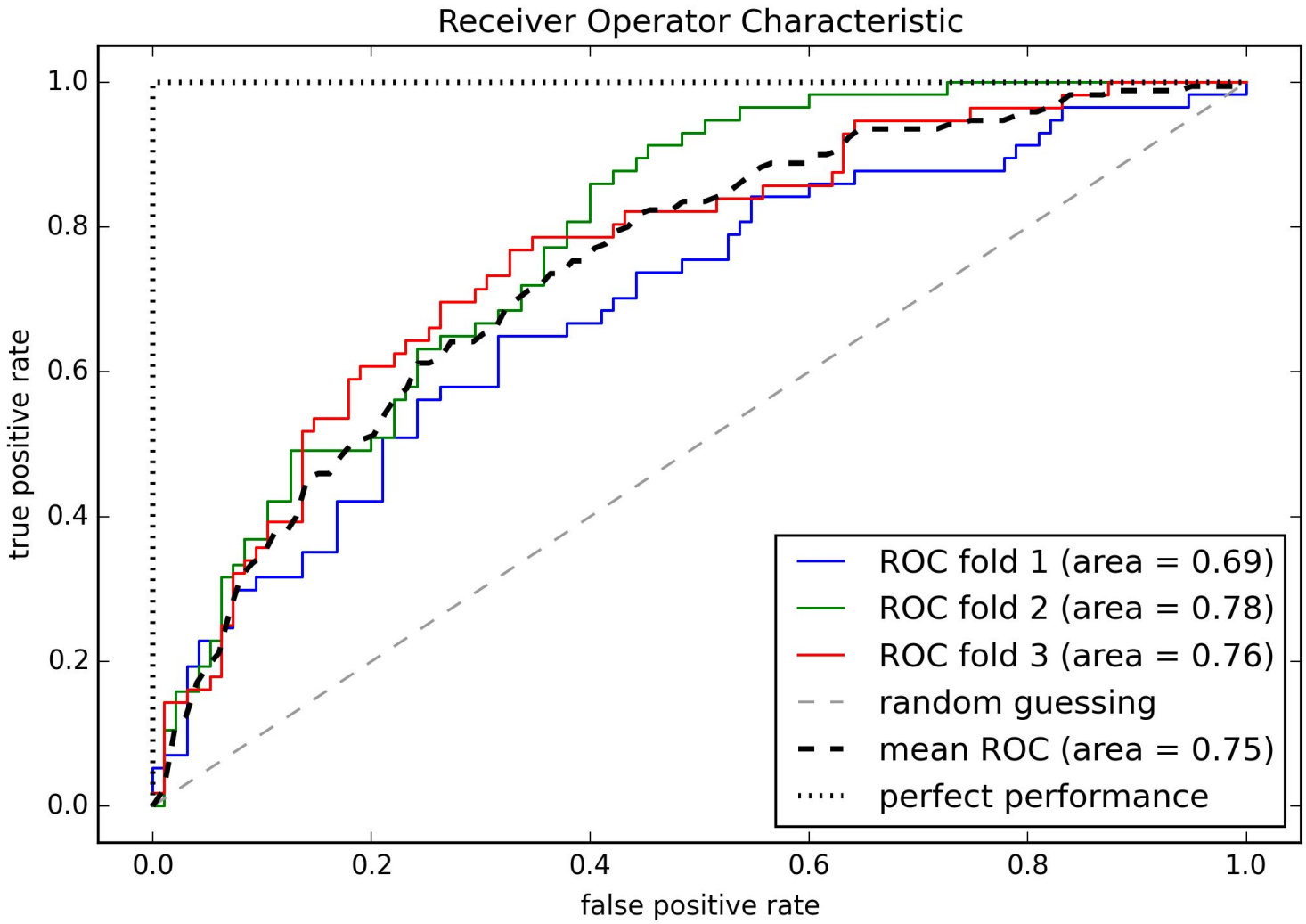
# Scikit-Learn: Overfitting vs Underfitting



# Scikit-Learn: Ottimizzazione dei parametri



# Scikit-Learn: Ottimizzazione dei parametri



# Scikit-Learn: Grid Search

I modelli di apprendimento automatico sono parametrizzati in modo che il loro comportamento possa essere regolato per un determinato problema.

I modelli possono avere molti parametri e trovare la migliore combinazione di parametri può essere trattato come un problema di ricerca operativa (ottimizzazione lineare).

L'ottimizzazione dei parametri è un passo finale nel processo di apprendimento automatico applicato prima di presentare i risultati.

Talvolta viene definita ***Hyperparameter optimization*** in cui i parametri dell'algoritmo vengono definiti iperparametri mentre i coefficienti rilevati dall'algoritmo di apprendimento automatico vengono definiti parametri.

# Scikit-Learn: Grid Search

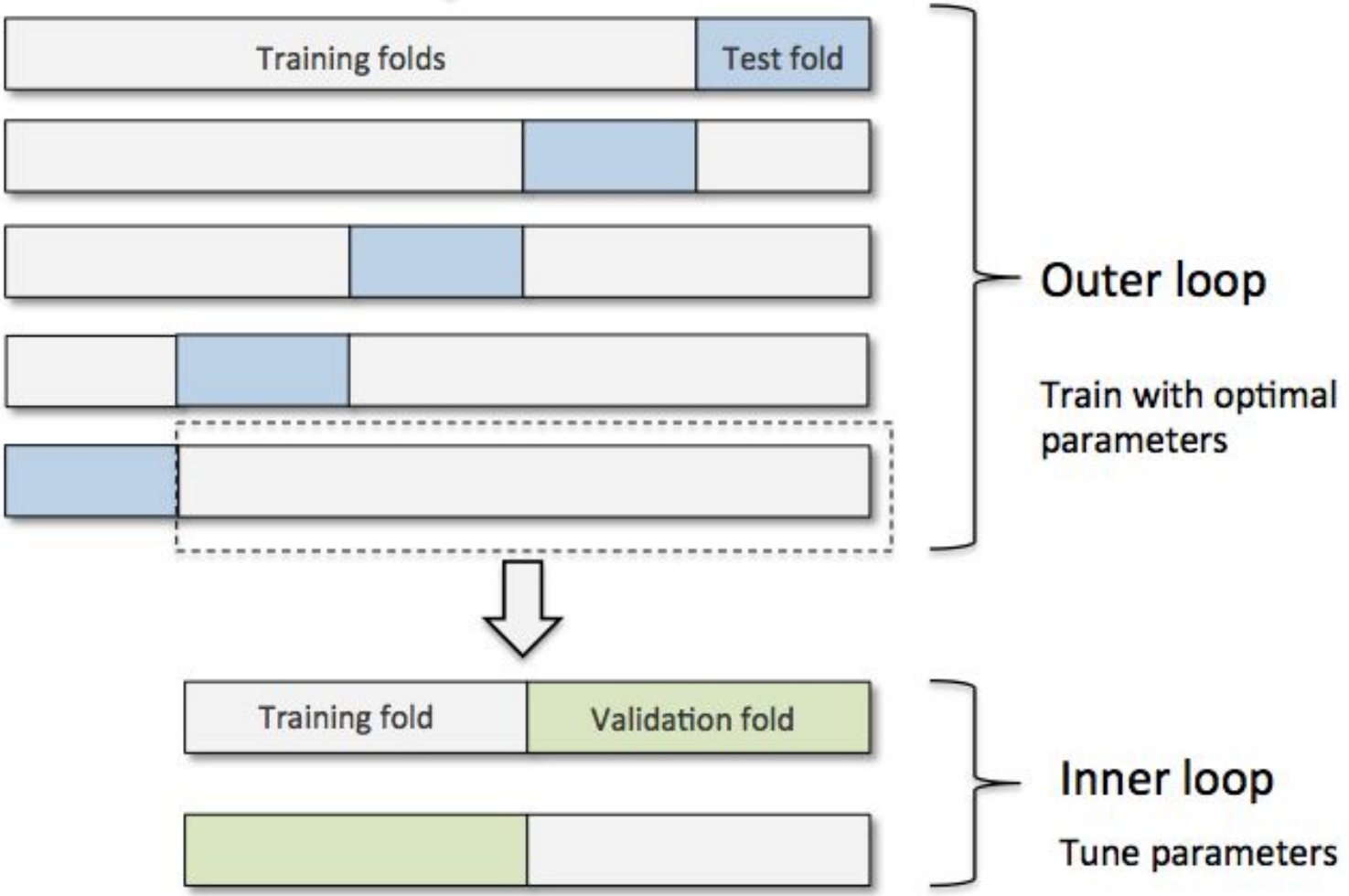
Definito come problema di ricerca, è possibile utilizzare diverse strategie di ricerca per trovare un parametro valido o affidabile o un insieme di parametri per un algoritmo su un dato problema.

Due semplici e facili strategie di ricerca sono la ***grid search*** e la ***ricerca casuale***. Scikit-learn fornisce questi due metodi per la regolazione dei parametri dell'algoritmo.

La ***grid search*** è un approccio alla regolazione dei parametri che costruisce e valuta metodicamente un modello per ciascuna combinazione di parametri dell'algoritmo specificati in una griglia (grid appunto).

Il seguente esempio valuta i diversi valori del parametro ***C*** di un classificatore SVM sul set di dati standard del diabete.

# Scikit-Learn: Grid Search



# Scikit-Learn: Grid Search

```
import numpy as np

from sklearn import datasets

from sklearn.svm import SVC

from sklearn.model_selection import GridSearchCV

dataset = datasets.load_diabetes()

cParams = np.array([1,2,3,4,5])

# create and fit a ridge regression model, testing each alpha

model = SVC()

grid = GridSearchCV(estimator=model, param_grid=dict(C=cParams))

grid.fit(dataset.data, dataset.target)

print(grid)

# summarize the results of the grid search

print(grid.best_score_)

print(grid.best_estimator_.C)
```

# Scikit-Learn: Clustering

Il **clustering** fa parte degli algoritmi di apprendimento ***non*** supervisionato e ha come obiettivo quello di raggruppare i modelli simili tra loro.

Il ***KMeans*** è un algoritmo di clustering basato sul concetto di centroide, ovvero una particolare combinazione di features rappresentativa di un determinato gruppo di campioni. Il KMeans ha bisogno di un numero iniziale di raggruppamenti.

Nell'analisi testuale si ha a che fare con numeri di features molto elevati ($10^3$~$10^5$) e la visualizzazione grafica del clustering diventa impossibile. Per poter visualizzare i raggruppamenti ottenuti con il KMeans bisogna prima ridurre lo spazio da *n* dimensioni a 2 e poi proiettare ogni campione in questo spazio, ad esempio con una ***PCA.***

# Scikit-Learn: Clustering

Consideriamo nuovamente il Brown corpus e prendiamo i documenti delle prime 5 categorie e limitiamo il numero di documenti a 300.

```
class BrownDataset(object):
  def __init__(self, categories=3, maxFeatures=100):
    documents = [(brown.raw(fileid), category)
                 for category in brown.categories()[:categories]
                 for fileid in brown.fileids(category)]
    random.shuffle(documents)
    documents = documents[:300]
    vectorizer = TfidfVectorizer(strip_accents='unicode', max_features=maxFeatures, tokenizer=word_tokenize,
                                 stop_words='english',decode_error='ignore',  analyzer='word', norm='l2' )
    vectorizer.fit([d[0] for d in documents])
    self.data = vectorizer.transform([d[0] for d in documents]).toarray()
    self.target = [d[1] for d in documents]
    self.targetNames = list(set(self.target))
```

# Scikit-Learn: Clustering

Utilizzando l'oggetto BrownDataset.

```
brownDataset = BrownDataset(5)
data = scale(brownDataset.data)
samples, features = data.shape
clusters = len(brownDataset.targetNames)
labels = brownDataset.target
```

Applicacchiamo la ***PCA*** al nostro dataset e poi usiamo il KMeans per ottenere i clusters in 2D.

```
reduced_data = PCA(n_components=2).fit_transform(data)
kmeans = KMeans(init='k-means++', n_clusters=clusters)
kmeans.fit(reduced_data)
print kmeans.cluster_centers_
```

# Scikit-Learn: Clustering

Prepariamo i dati per la visualizzazione.

```
# Step size of the mesh. Decrease to increase the quality of the VQ.
h = .02  # point in the mesh [x_min, x_max]x[y_min, y_max].
# Plot the decision boundary. For that, we will assign a color to each
x_min, x_max = reduced_data[:, 0].min() - 1, reduced_data[:, 0].max() + 1
y_min, y_max = reduced_data[:, 1].min() - 1, reduced_data[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
# Obtain labels for each point in mesh. Use last trained model.
Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])
# Put the result into a color plot
Z = Z.reshape(xx.shape)
```

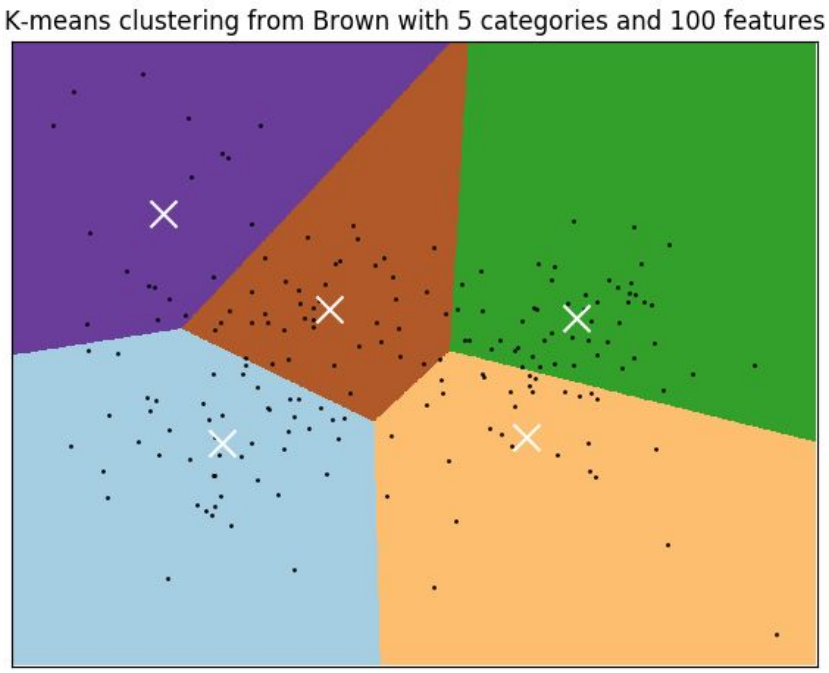
# Scikit-Learn: Clustering

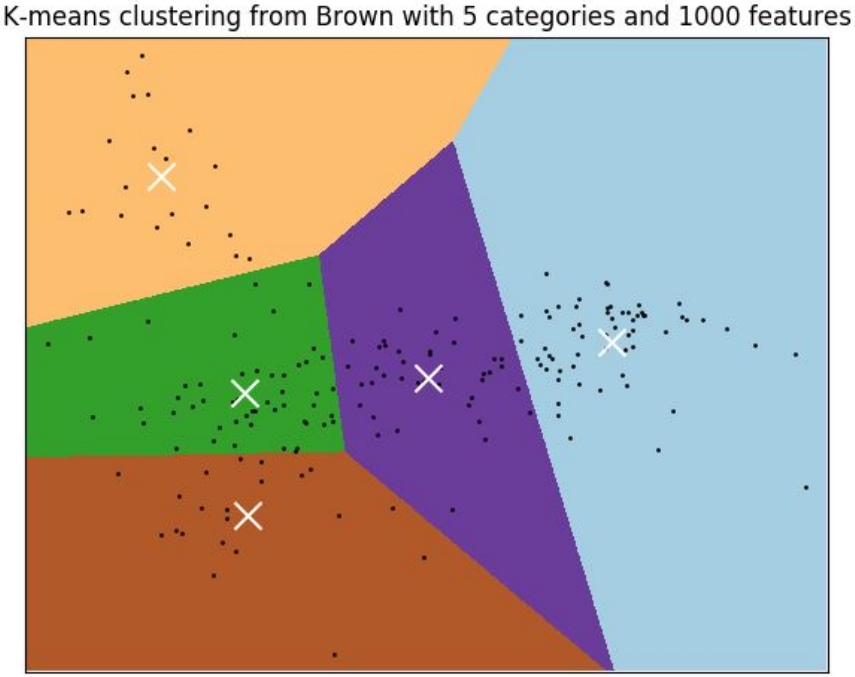
Visualizziamo i gruppi con sovrapposta una immagine a colori dei vari raggruppamenti.

```
Z = Z.reshape(xx.shape)

plt.figure(1)

plt.clf()

plt.imshow(Z, interpolation='nearest',  extent=(xx.min(), xx.max(), yy.min(), yy.max()),

        cmap=plt.cm.Paired, aspect='auto',  origin='lower')

plt.plot(reduced_data[:, 0], reduced_data[:, 1], 'k.', markersize=2)

centroids = kmeans.cluster_centers_

plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', s=169, linewidths=3, color='w', zorder=10)

plt.title('K-means clustering on the categories from Brown corpus dataset (PCA-reduced data) Centroids are marked with white cross')

plt.xlim(x_min, x_max)

plt.ylim(y_min, y_max)

plt.xticks(())

plt.yticks(())

plt.show()
```

# Scikit-Learn: Clustering

Utilizzando le top 100 e 1000 features con *TF-Idf* maggiore questo è il risultato che si ottiene.



# Scikit-Learn: K-Means

Per ottenere una stima del K migliore da utilizzare come numero di clusters del K-Means si può utilizzare il così detto metodo ***Elbow***.

Immaginiamo di ottenere una trasformazione TF-IDF data una lista di testi ***textList***.

```
tfidfVectorizer = TfidfVectorizer(max_features=100,
                                  stop_words='english',
                                  use_idf=True,
                                  tokenizer=word_tokenize,
                                  ngram_range=(1, 2))
tfidfMatrix = tfidfVectorizer.fit_transform(textList)
```

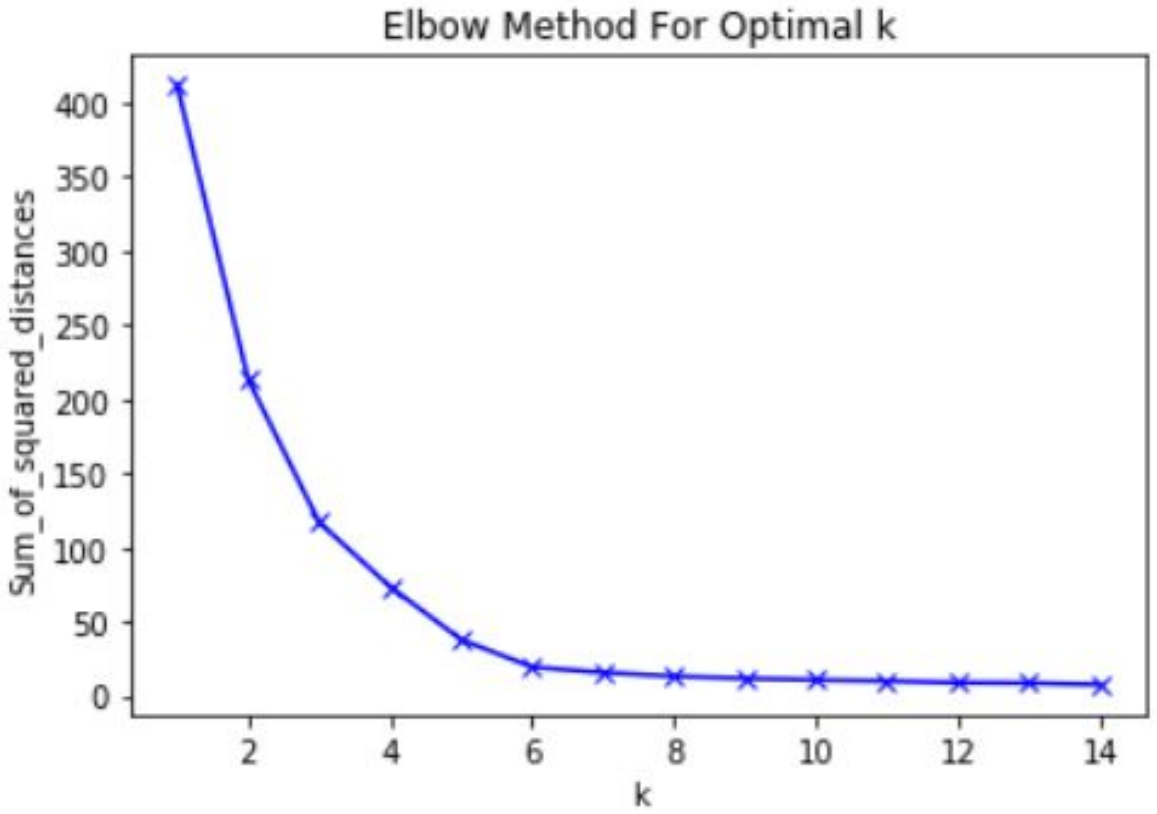
# Scikit-Learn: K-Means

Definiamo una funzione che calcola la somma dei quadrati delle distanza più vicine ad ogni cluster ad ogni iterazione (parametro ***inertia_*** del kmeans).

```
def getBestKForKMeans(tfidfMatrix, maxClusters=15):
    sumOfSquaredDistances = []
    K = range(1, maxClusters)
    for k in K:
        km = KMeans(n_clusters=k)
        km = km.fit(tfidfMatrix)
        # inertia_: Sum of squared distances of samples to their closest cluster center.
        sumOfSquaredDistances.append(km.inertia_)
    print sumOfSquaredDistances
    plt.plot(K, sumOfSquaredDistances, 'bx-')
    plt.xlabel('k')
    plt.ylabel('Sum of Squared Distances')
    plt.title('Elbow Method For Optimal k')
    plt.show()
```

# Scikit-Learn: K-Means

Si ottiene una curva al variare di K. Se la curva ha la forma di un gomito (Elbow appunto) allora il valore di K ottimale si trova in prossimità del gomito.



# Scikit-Learn: Topic Modeling

Nel Natural Language Processing (NLP) e più in generale nell'apprendimento statistico gli algoritmi di ***topic modeling*** sono modelli statistici che cercano di associare un ***argomento*** ad un documento appartenente ad una collezione di documenti.

Il concetto di base è che documenti che trattano lo stesso argomento hanno una probabilità maggiore di contenere termini simili rispetto a documenti che trattano altri argomenti.

La topic modeling consente quindi di trovare i termini che compongono un particolare topic e quindi poi di poter raggruppare in maniera automatica documenti con lo stesso topic.

# Scikit-Learn: Topic Modeling

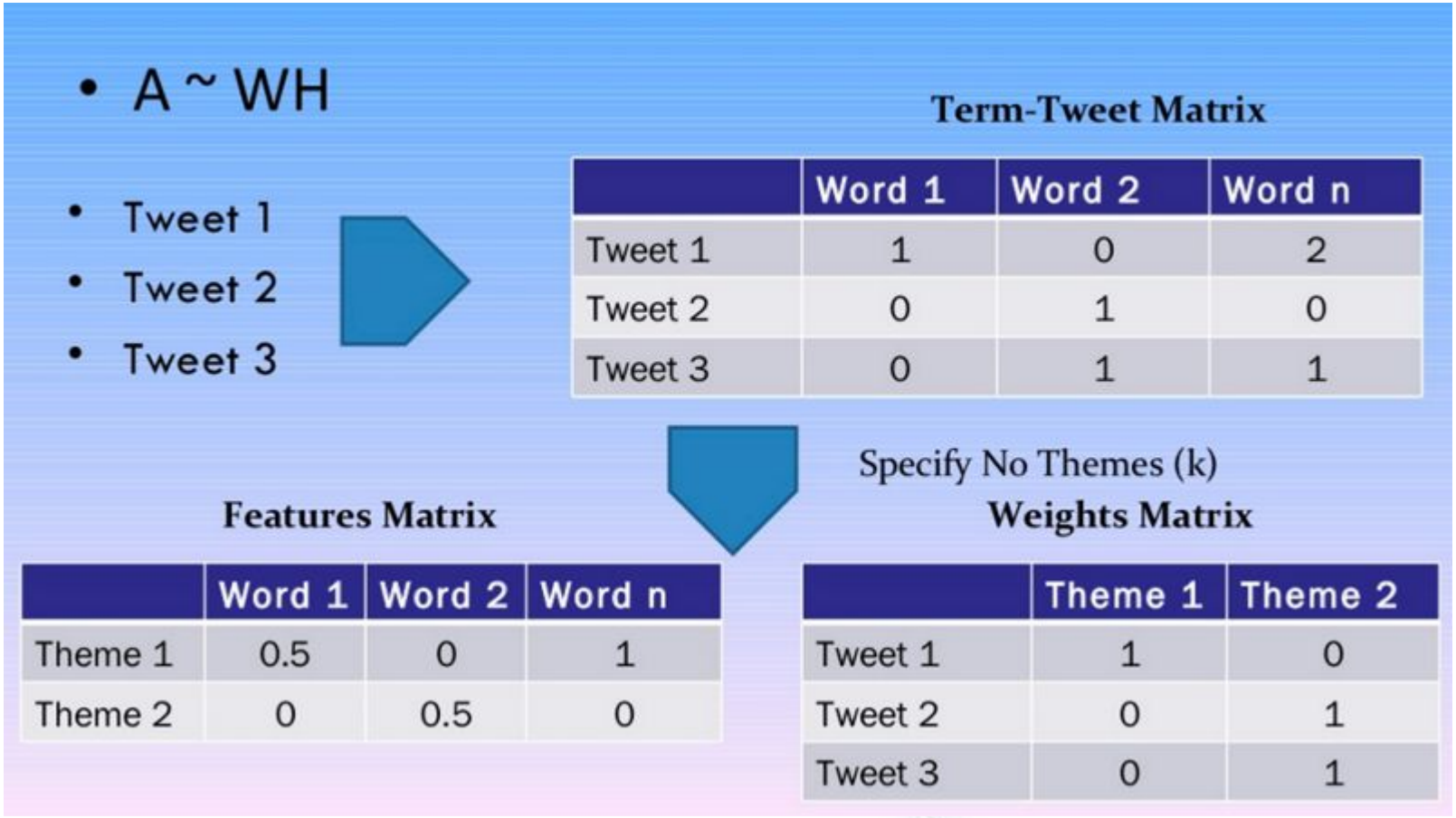
***Latent Dirichlet Allocation (LDA)*** e ***Non-Negative Matrix Factorization*** (***NMF***) sono algoritmi utilizzati per determinare gli argomenti (topics) che sono presenti in un corpus.

La base matematica alla base di NMF è molto diversa dalla LDA. NMF è stato incluso in Scikit Learn per un po 'di tempo ma LDA è stata inclusa solo di recente (alla fine del 2015).

La cosa grandiosa dell'utilizzo di Scikit Learn è il fatto che costituisce una API (interfaccia) coerente che rende quasi banale eseguire il Topic Modeling utilizzando sia LDA che NMF.

Scikit Learn include anche le opzioni di seeding per NMF che aiutano notevolmente con la convergenza degli algoritmi e offre sia le varianti online che batch di LDA.

# Scikit-Learn: Topic Modeling



# Scikit-Learn: Topic Modeling

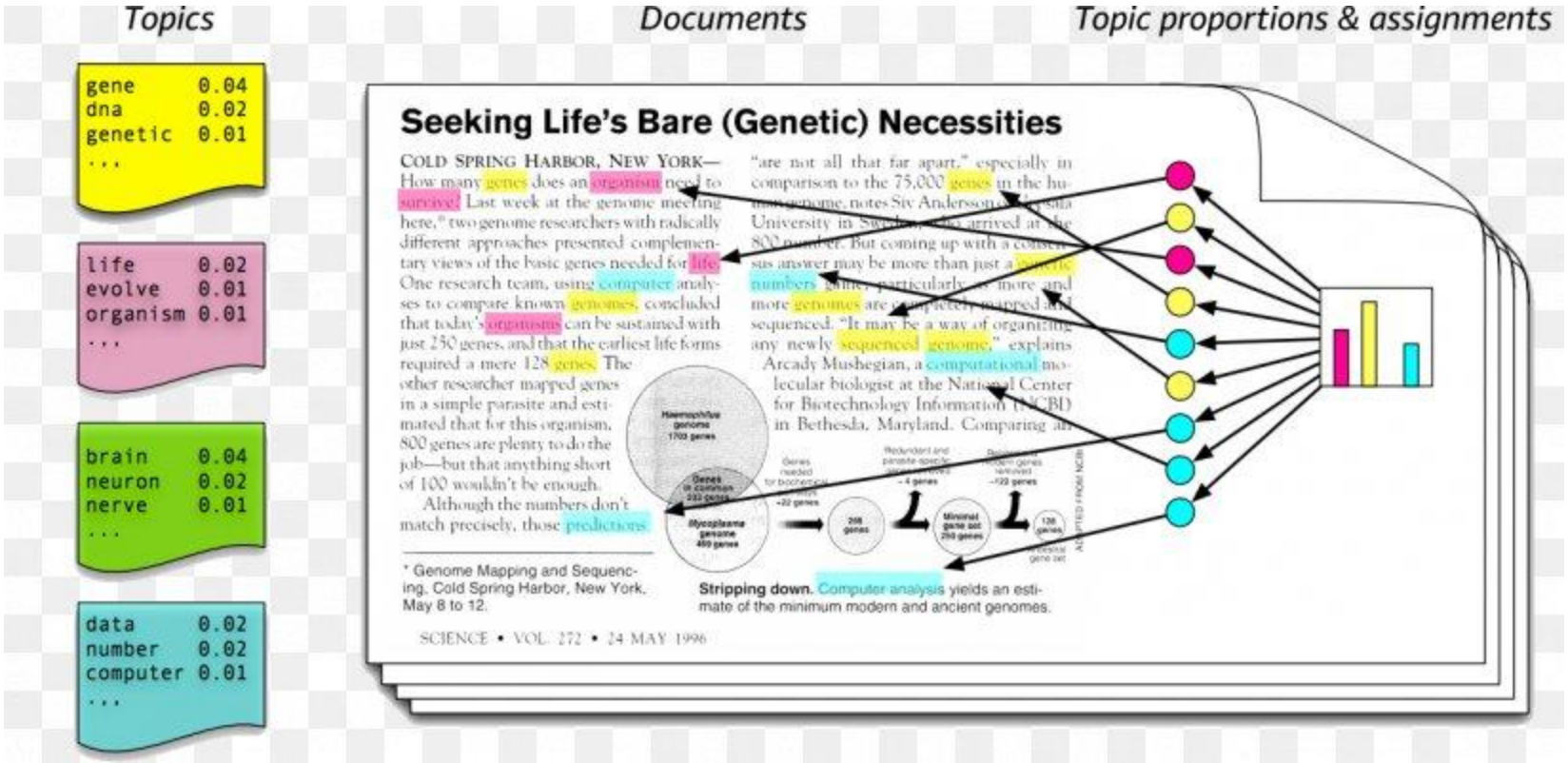
Sebbene LDA e NMF abbiano una base matematica diversa, entrambi gli algoritmi sono in grado di restituire i documenti che appartengono a un argomento in un corpus e le parole che appartengono a un argomento.

LDA si basa sulla modellizzazione grafica probabilistica mentre l'NMF si basa sull'algebra lineare.

Entrambi gli algoritmi prendono come input una matrice (vale a dire, ogni documento rappresentato come una riga, con ogni colonna contenente la metrica associata al token nel corpus).

Lo scopo di ciascun algoritmo è quindi quello di produrre 2 matrici più piccole; un documento in matrice di argomenti e una matrice di token in argomento che quando moltiplicati insieme riproducono una matrice di token con l'errore più basso.

# Scikit-Learn: Topic Modeling



# Scikit-Learn: Topic Modeling

Prepariamo il nostro dataset.

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.decomposition import NMF, LatentDirichletAllocation
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer

dataset = fetch_20newsgroups(shuffle=True, random_state=1, remove=('headers', 'footers', 'quotes'))
documents = dataset.data
no_features = 1000
```

Limitiamo il numero delle features a 1000 per rapidità di calcolo.

# Scikit-Learn: Topic Modeling

Processiamo il corpus di documenti

```
# NMF è basato sulla TF-IDF

tfidf_vectorizer = TfidfVectorizer(max_df=0.95, min_df=2, max_features=no_features, stop_words='english')

tfidf = tfidf_vectorizer.fit_transform(documents)

tfidf_feature_names = tfidf_vectorizer.get_feature_names()

# LDA usa solo il conteggio dei token in quanto basato un modello probabilistico

tf_vectorizer = CountVectorizer(max_df=0.95, min_df=2, max_features=no_features, stop_words='english')

tf = tf_vectorizer.fit_transform(documents)

tf_feature_names = tf_vectorizer.get_feature_names()
```

Limitiamo il numero delle features a 1000 per rapidità di calcolo.

# Scikit-Learn: Topic Modeling

Limitiamo il numero di topics a 10 e calcoliamo i topics con i due metodi.

```
no_topics = 10

# Run NMF

nmf = NMF(n_components=no_topics, random_state=1, alpha=.1, l1_ratio=.5, init='nndsvd').fit(tfidf)

# Run LDA

lda = LatentDirichletAllocation(n_components=no_topics, max_iter=5,
                                learning_method='online', learning_offset=50.,
                                random_state=0).fit(tf)

display_topics(nmf, tfidf_feature_names, 5)

display_topics(lda, tf_feature_names, 5)
```

# Scikit-Learn: Topic Modeling

Definiamo una funzione di supporto per visualizzare i topics:

```
def display_topics(model, feature_names, no_top_words):
    for topic_idx, topic in enumerate(model.components_):
        print "Topic %d:" % (topic_idx)
        print " ".join([feature_names[i]
                        for i in topic.argsort()[:-no_top_words - 1:-1]])
```

# Scikit-Learn: Topic Modeling

***Topic 0:***

don just people think like know time good right ve

***Topic 1:***

card video monitor drivers cards bus vga driver color ram

***Topic 2:***

god jesus bible christ faith believe christian christians church sin

***Topic 3:***

game team year games season players play hockey win player

***Topic 4:***

car new 00 sale 10 price offer condition shipping 20

***Topic 5:***

thanks does know advance mail hi anybody info looking help

***Topic 6:***

windows file use files dos window program using problem running

***Topic 7:***

edu soon cs university com email internet article ftp send

***Topic 8:***

key chip encryption clipper keys government escrow public use algorithm

***Topic 9:***

drive scsi drives hard disk ide controller floppy cd mac

# Scikit-Learn: Topic Modeling

***Topic 0:***

people gun armenian armenians war turkish states israel said children

***Topic 1:***

government people law mr use president don think right public

***Topic 2:***

space program output entry data nasa use science research build

***Topic 3:***

key car chip used keys bike use bit clipper number

***Topic 4:***

edu file com available mail ftp files information image send

***Topic 5:***

god people does jesus say think believe don know just

***Topic 6:***

windows use drive thanks does problem know card like using

***Topic 7:***

ax max b8f g9v a86 pl 145 1d9 0t 34u

***Topic 8:***

just don like think know good time ve people said

***Topic 9:***

10 00 25 15 12 20 11 14 17 16

# Scikit-Learn: Topic Modeling

Abbiamo utilizzato diversi parametri per configurare la ***LatentDirichletAllocation***, nel dettaglio:

- ***n_components***: definisce il numero di di topics (default=10)
- ***max_iter***: definisce il numero massimo di iterazioni dell'algoritmo
- ***learning_method***: definisce il metodo di aggiornamento delle ***components_*** che rappresentano i vettori dei token, può essere:
  - *Batch:* usa tutto il dataset per calcolare le componenti dei topics ad ogni iterazione.
  - *Online:* aggiorna ***components_*** in maniera incrementale ad ogni iterazione usando solo porzioni del dataset totale. Il *tasso di apprendimento (learning rate)* è governato dal parametro *learning_decay (default=0.7).*
- ***learning_decay***: controlla il tasso di apprendimento della LDA e garantisce la convergenza del metodo quando è settato nel range [0.5,1].
- ***learning_offset***: è un numero positivo che diminuisce il peso degli aggiornamenti delle componenti dei topics per limitare le prime iterazioni.

# Scikit-Learn: Topic Modeling

Analizziamo le performance della LDA: ***Likelihood*** e ***Perplexity***.

```
no_topics = 10

lda = LatentDirichletAllocation(n_components=no_topics, max_iter=5,
                                learning_method='online', learning_offset=50.).fit(tf)

# Likelyhood: Higher the better  “Latent Dirichlet Allocation

# David M. Blei, Andrew Y. Ng, Michael I. Jordan; 3(Jan):993-1022, 2003.”

print("Log Likelihood: ", lda.score(tf))

# Perplexity: Lower the better. Perplexity = exp(-1. * log-likelihood per word)

print("Perplexity: ", lda.perplexity(tf))

# ('Log Likelihood: ', -3064601.6245813086)

# ('Perplexity: ', 262.08843745431557)

# See model parameters

pprint(lda.get_params())
```

# Scikit-Learn: Topic Modeling

Analizziamo il modello LDA per ottenere il *dominant topic* dato uno specifico documento.

```
print len(lda.components_) # the number of topics' components equal to # topics
print lda.components_[0] # the first topic weights
print lda.components_[0].argsort() # argsort returns the sorted indexes of a numpy array
# we need last 10 indexes which are the words with highest weights for the first topic
topNTopicWordIndexes = lda.components_[0].argsort()[:-11:-1]
print topNTopicWordIndexes
# given the top 10 words indexes, the words name are contained in the feature names list
print " ".join([tfFeatureNames[wordIndex] for wordIndex in topNTopicWordIndexes])
```

# Scikit-Learn: Topic Modeling

Analizziamo il modello LDA per ottenere il *dominant topic* dato uno specifico documento.

```
# transform method returns the topic vector for the first document doc0
doc0Topics = lda.transform(dataVectorized[0])[0]
print doc0Topics
# the dominant topic is the one with the highest value, argmax return the index of max in a numpy array
doc0DominantTopic = np.array(doc0Topics).argmax()
print doc0DominantTopic
# again top 10 words for the dominant topic of doc0
top10TopicWordIndexesOfDoc0 = lda.components_[doc0DominantTopic].argsort()[:-11:-1]
print " ".join([tfFeatureNames[wordIndex] for wordIndex in top10TopicWordIndexesOfDoc0])
```

# Scikit-Learn: Topic Modeling

Possiamo ottimizzare i parametri della LDA utilizzando la grid search ed in particolare la classe GridSearchCV già vista in precedenza.

```
no_topics = 10

lda = LatentDirichletAllocation()

searchParams = {'n_components': [10, 15, 20, 25, 30], 'learning_decay': [.5, .7, .9]}

model = GridSearchCV(lda, param_grid=searchParams)

model.fit(tf)

print(model)

print(model.best_score_)
```

# Scikit-Learn: Topic Modeling

Topic per ogni documento:

```
no_topics = 10

lda = LatentDirichletAllocation()

searchParams = {'n_components': [10, 15, 20, 25, 30], 'learning_decay': [.5, .7, .9]}

model = GridSearchCV(lda, param_grid=searchParams)

model.fit(tf)

ldaOutput = model.transform(dataVectorized)

topicnames = ["Topic" + str(i) for i in range(no_topics)]

docnames = ["Doc" + str(i) for i in range(len(documents))]

dfDocumentTopic = pd.DataFrame(np.round(ldaOutput, 2),
                                        columns=topicnames,
                                        index=docnames)

print dfDocumentTopic.to_string()
```

# Scikit-Learn: Topic Modeling

Utilizzando la grid search otteniamo il numero di topics pari a 10 e un leaning decay pari a 9.

```
Best Model's Params:  {'learning_decay': 0.9, 'n_topics': 10}
Best Log Likelyhood Score:  -3417650.82946
  Model Perplexity:  2028.79038336
```

Per capire meglio come variano le performance della LDA al variare di questi due parametri possiamo vedere l'andamento della likelihood al variare del learning_decay e del numero di topics per esempio.

# NLTK: riferimenti

- http://scikit-learn.org/stable/tutorial/index.html

- Machine Learning con Python: costruire algoritmi per generare conoscenza (Data Science Vol. 2) (ISBN-13: 978-88-503-3397-4) materiale disponibile qui.

- Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems (ISBN-13:  978-1491962299)

- Esempi: https://github.com/marcoortu/WAAT-2020