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

NLTK Text Processing

NLTK: processare html

Gran parte del testo sul Web è sotto forma di documenti HTML.

È possibile utilizzare un browser Web per salvare una pagina, quindi accedere a questo file. Python permette di automatizzare il lavoro.

Usando ***requests.get()*** del modulo requests è possibile accedere ad una pagina Web. Prendiamo alla storia della BBC News chiamata "*Blondes to die out in 200 years*" che è un fatto scientifico consolidato.

```
>>> url = "http://news.bbc.co.uk/2/hi/health/2284783.stm"
>>> html = requests.get(url).content
>>> html[:60]
'<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN'
```

È possibile digitare ***print(html)*** per vedere il contenuto HTML in tutta la sua gloria, compresi meta tag, una mappa immagine, JavaScript, moduli e tabelle.

NLTK: processare html

Per estrarre il testo da HTML usiamo ***BeautifulSoup***, disponibile da

```
>>> from bs4 import BeautifulSoup
>>> from nltk import word_tokenize
>>> import requests, nltk
>>> raw = BeautifulSoup(requests.get(url).content).get_text()
>>> tokens = word_tokenize(raw)
>>> tokens
['BBC', 'NEWS', '|', 'Health', '|', 'Blondes', '"to", 'die', 'out', ...]
```

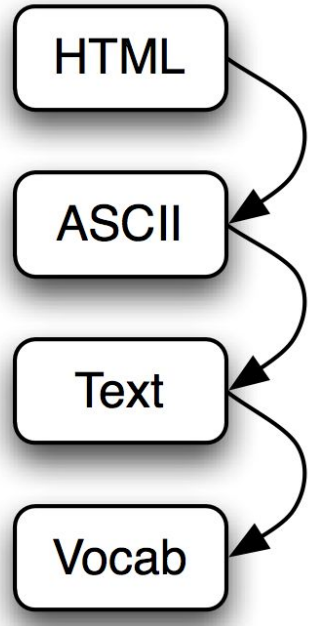
Questo contiene ancora materiale indesiderato riguardante la navigazione del sito e storie correlate.

Con alcuni *trial and error* si può provare a cercare gli indici di inizio e di fine del contenuto e selezionare i token di interesse e inizializzare un testo come prima.

NLTK: processare html

```
>>> tokens = tokens[110:390]
>>> text = nltk.Text(tokens)
>>> text.concordance('gene')
Displaying 5 of 5 matches:
hey say too few people now carry the gene for blondes to last beyond the next
blonde hair is caused by a recessive gene . In order for a child to have blond
have blonde hair , it must have the gene on both sides of the family in the g
ere is a disadvantage of having that gene or by chance . They do n't disappear
des would disappear is if having the gene was a disadvantage and I do not
thin
```

NLTK: processare html




```
html = urlopen(url).read()
raw = nltk.clean_html(html)
raw = raw[750:23506]
```

Download web page, strip HTML if necessary, trim to desired content

```
tokens = nltk.wordpunct_tokenize(raw)
tokens = tokens[20:1834]
text = nltk.Text(tokens)
```

Tokenize the text, select tokens of interest, create an NLTK text

```
words = [w.lower() for w in text]
vocab = sorted(set(words))
```

Normalize the words, build the vocabulary

NLTK: processare csv

Per estrarre agevolmente dati in formato ***csv*** (Comma Separated Values) possiamo utilizzare il modulo ***pandas*** builtin di Python

```
>>> import pandas as pd
>>> rows = rows = pd.read_csv("./corpora/post_comments.csv").to_dict('records')
>>> row[0].keys()
['comment_id', 'comment_text']
>>> tokens = word_tokenize(row[0]['comment_text'])
>>> tokens
['Very', 'nice', ',', 'thanks', '.', 'I', "'ll", 'do', 'more', 'extensive', 'research', 'on', 'that', '.', ':', '-', ')']
```

Il metodo read_csv importa il CSV in un DataFrame pandas che viene poi convertito in una lista di dizionari. Ogni riga del CSV è rappresentata da un dizionario le cui chiavi sono le colonne del CSV

NLTK: processare html

Funzioni disponibili sulle stringhe in Python

Method	Functionality
`s.find(t)`	index of first instance of string `t` inside `s` (`-1` if not found)
`s.rfind(t)`	index of last instance of string `t` inside `s` (`-1` if not found)
`s.index(t)`	like `s.find(t)` except it raises `ValueError` if not found
`s.rindex(t)`	like `s.rfind(t)` except it raises `ValueError` if not found
`s.join(text)`	combine the words of the text into a string using `s` as the glue
`s.split(t)`	split `s` into a list wherever a `t` is found (whitespace by default)
`s.splitlines()`	split `s` into a list of strings, one per line
`s.lower()`	a lowercased version of the string `s`
`s.upper()`	an uppercased version of the string `s`
`s.title()`	a titlecased version of the string `s`
`s.strip()`	a copy of `s` without leading or trailing whitespace
`s.replace(t, u)`	replace instances of `t` with `u` inside `s`

NLTK: Text Processing con Unicode

I nostri programmi avranno spesso bisogno di trattare lingue diverse e set di caratteri diversi. Il concetto di "testo semplice" è una finzione.

Nel Regno Unito si utilizza la codifica **ASCII**, in Italia **ISO-8859-1** (Latin-1), in Polonia **ISO-8859-2** (Latin-2) etc..

In Europa si utilizza uno dei set di caratteri latini estesi, contenente caratteri come "ø" per danese e norvegese, "ő" per ungherese, "ñ" per spagnolo e bretone, e "ň" per ceco e slovacco .

Vediamo una panoramica su come utilizzare Unicode per l'elaborazione di testi che utilizzano set di caratteri non ASCII.

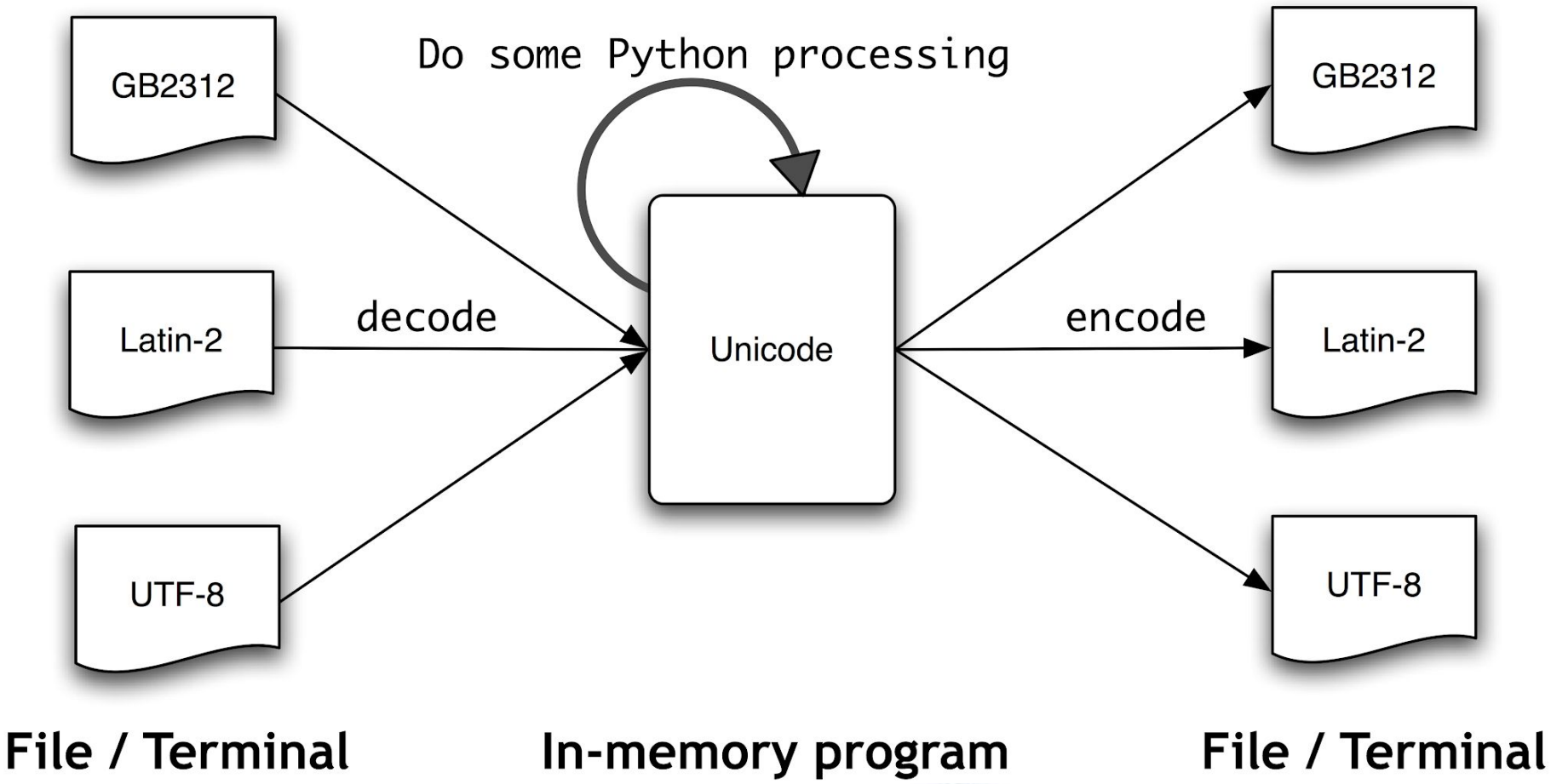
NLTK: Text Processing con Unicode

Unicode supporta oltre un milione di caratteri. A ogni carattere è assegnato un numero unico, chiamato appunto di ***unicode***.
In Python, gli unicode sono scritti nel modulo \ uXXXX, dove XXXX è il numero in formato esadecimale a 4 cifre.

All'interno di un programma, possiamo manipolare le stringhe Unicode proprio come le normali stringhe. Tuttavia, quando i caratteri Unicode vengono memorizzati in file o visualizzati su un terminale, devono essere codificati come un flusso di byte.

Alcune codifiche (come ASCII e Latin-2) utilizzano un singolo byte per ogni codice, quindi possono supportare solo un piccolo sottoinsieme di Unicode, sufficiente per una singola lingua.
Altre codifiche (come UTF-8) utilizzano più byte e possono rappresentare l'intera gamma di caratteri Unicode.

NLTK: Text Processing con Unicode



NLTK: Text Processing con Unicode

Il testo nei file sarà in una particolare codifica, quindi abbiamo bisogno di un meccanismo per tradurlo in Unicode - la traduzione in Unicode è chiamata decodifica.

Al contrario, per scrivere Unicode in un file o in un terminale, dobbiamo prima tradurlo in una codifica adatta: questa traduzione di Unicode è chiamata codifica.

Da una prospettiva Unicode, i caratteri sono entità astratte che possono essere realizzate come uno o più ***glyphs*** (glifi).

Solo i ***glyphs*** possono apparire su uno schermo o essere stampati su carta. Un ***font*** è una ***mappatura*** da ***caratteri*** a ***glyphs***.

NLTK: Text Processing con Unicode

Supponiamo di avere un piccolo file di testo e di sapere come è codificato. Ad esempio, polish-lat2.txt, come suggerisce il nome, è uno snippet di testo polacco (dalla Wikipedia polacca, vedi http://pl.wikipedia.org/wiki/Biblioteka_Pruska).

Questo file è codificato come Latin-2, noto anche come ISO-8859-2. La funzione nltk.data.find () individua il file per noi.

```
>>> import codecs
>>> nltk.download("all-corpora") # NB!! solo se non lo avete già fatto
>>> path = nltk.data.find('corpora/unicode_samples/polish-lat2.txt')
>>> f = codecs.open(path, encoding='latin2')
>>> for line in f:
...     line = line.strip()
...     print(line)
Pruska Biblioteka Państwowa. Jej dawne zbiory znane pod nazwą
"Berlinka" to skarb kultury i sztuki niemieckiej. Przewiezione przez
```

NLTK: Text Processing con Unicode

Se questo non viene visualizzato correttamente sul tuo terminale, o se vogliamo vedere i valori numerici sottostanti (o "codepoints") dei caratteri, allora possiamo convertire tutti i caratteri non ASCII nelle loro due cifre \ xXX e quattro cifre \ uXXXX rappresentazioni:

```
>>> import codecs
>>> f = codecs.open(path, encoding='latin2')
>>> for line in f:
...     line = line.strip()
...     print(line.encode('unicode_escape'))
b'Pruska Biblioteka Pa\\u0144stwowa. Jej dawne zbiory znane pod nazw\\u0105'
b'"Berlinka" to skarb kultury i sztuki niemieckiej. Przewiezione przez'
b'Niemc\\xf3w pod koniec II wojny \\u015bwiatowej na Dolny \\u015al\\u0105sk,
zosta\\u0142y'
```

NLTK: Text Processing con Unicode

La prima riga illustra una stringa di escape Unicode preceduta dalla stringa di escape \ u, ovvero \ u0144. Il carattere Unicode rilevante verrà visualizzato sullo schermo come il glifo ñ. Nella terza riga dell'esempio precedente, vediamo \ xf3, che corrisponde al glifo ó, ed è compreso nell'intervallo 128-255.

In Python 3, il codice sorgente viene codificato utilizzando UTF-8 per impostazione predefinita e puoi includere caratteri Unicode nelle stringhe se stai utilizzando IDLE o un altro editor di programma che supporta Unicode.

I caratteri Unicode arbitrari possono essere inclusi utilizzando la sequenza di escape \ uXXXX. Troviamo il numero intero ordinale di un personaggio usando ord (). Per esempio:

```
>>> ord('ń') # Solo in Python 3
324
```

NLTK: Text Processing con Unicode

I tokenizer NLTK consentono le stringhe Unicode come input e producono corrispondentemente stringhe Unicode come output.

```
>>> path = nltk.data.find('corpora/unicode_samples/polish-lat2.txt')
>>> lines = codecs.open(path, encoding='latin2').readlines()
>>> for line in lines:
...         for token in nltk.word_tokenize(line):
...             print token
Pruska
Biblioteka
Państwowa
```

Se si lavora con i caratteri in una particolare codifica locale, probabilmente si vuole essere in grado di usare i metodi standard per inserire e modificare le stringhe in un file Python. Per fare ciò, includere la stringa ***'# - * - coding: <coding> - * -'*** come prima o seconda riga del tuo file. Nota che ***<coding>*** deve essere una stringa come ***"latin-1", "big5" o "utf-8"***

NLTK: Normalization -> Stemming

Nei precedenti esempi di programma abbiamo spesso convertito il testo in minuscolo prima di fare qualsiasi cosa con le parole, ad es. set (w.lower () per w nel testo). Usando ***lower()***, abbiamo normalizzato il testo in lettere minuscole in modo che la distinzione tra ***The*** e ***the*** sia ignorata. Spesso vogliamo andare oltre questo e rimuovere tutti gli affissi, un compito noto come "***stemming***". Un ulteriore passo è quello di assicurarsi che la forma risultante sia una parola conosciuta in un dizionario, un'attività nota come ***lemmatizzazione***. Discutiamo ognuno di questi a turno. Innanzitutto, dobbiamo definire i dati che useremo in questa sezione:

```
>>> raw = """DENNIS: Listen, strange women lying in ponds distributing swords
... is no basis for a system of government.  Supreme executive power derives from
... a mandate from the masses, not from some farcical aquatic ceremony."""
>>> tokens = word_tokenize(raw)
```

NLTK: Normalization -> Stemming

NLTK include diversi stemmer pronti all'uso e, se hai mai bisogno di uno stemmer, dovresti usare uno di questi in preferenza per creare il tuo usando espressioni regolari, poiché gestiscono una vasta gamma di casi irregolari. Gli stemmer Porter e Lancaster seguono le proprie regole per rimuovere gli affissi. Osservare che lo stemmer di Porter gestisce correttamente la parola ***lying*** (mappandola per ***lie***), mentre lo stemmer di Lancaster no.

```
>>> porter = nltk.PorterStemmer()
>>> lancaster = nltk.LancasterStemmer()
>>> [porter.stem(t) for t in tokens]
['DENNI', ':', 'Listen', ',', 'strang', 'women', 'lie', 'in', 'pond',
'distribut', 'sword', 'is', 'no', 'basi', 'for', 'a', 'system', 'of', 'govern',
'.', 'Suprem', 'execut', 'power', 'deriv', 'from', 'a', 'mandat', 'from',
'the', 'mass', ',', 'not', 'from', 'some', 'farcic', 'aquat', 'ceremoni', '.']
>>> [lancaster.stem(t) for t in tokens]
['den', ':', 'list', ',', 'strange', 'wom', 'lying', 'in', 'pond', 'distribut',
'sword', 'is', 'no', 'bas', 'for', 'a', 'system', 'of', 'govern', '.', 'suprem',
'execut', 'pow', 'der', 'from', 'a', 'mand', 'from', 'the', 'mass', ',', 'not',
'from', 'som', 'farc', 'aqu', 'ceremony', '.']
```

NLTK: Normalization -> Lemmatization

Il lemmatizzatore WordNet rimuove solo gli affissi se la parola risultante è nel suo dizionario. Questo ulteriore processo di verifica rende il ***lemmatizer*** più lento rispetto agli stemmer sopra indicati. Si noti che non gestisce la menzogna, ma converte le women in woman.

```
>>> wnl = nltk.WordNetLemmatizer()
>>> [wnl.lemmatize(t) for t in tokens]
['DENNIS', ':', 'Listen', ',', 'strange', 'woman', 'lying', 'in', 'pond',
'distributing', 'sword', 'is', 'no', 'basis', 'for', 'a', 'system', 'of',
'government', '.', 'Supreme', 'executive', 'power', 'derives', 'from', 'a',
'mandate', 'from', 'the', 'mass', ',', 'not', 'from', 'some', 'farcical',
'aquatic', 'ceremony', '.']
```

Il lemmatizer WordNet è una buona scelta se si vuole compilare il vocabolario di alcuni testi e un elenco di lemmi validi (o parole chiave lessicali).

NLTK: Segmentation

La tokenizzazione è un'istanza di un problema più generale di segmentazione. Vedremo altre due istanze di questo problema, che utilizzano tecniche radicalmente diverse da quelle che abbiamo visto finora in questo capitolo.

Manipolare i testi a livello di singole parole spesso presuppone la possibilità di dividere un testo in singole frasi. Come abbiamo visto, alcuni corpora forniscono già l'accesso a livello di frase. Nell'esempio seguente, calcoliamo il numero medio di parole per frase nel Corpus Brown:

```
>>> len(nltk.corpus.brown.words()) / len(nltk.corpus.brown.sents())
20.250994070456922
```

NLTK: Segmentation

La segmentazione per quanto riguarda il NLP viene generalmente identificata nella seguente gerarchia.

- ➔ Corpus
- ➔ Document
- ➔ Sentence
- ➔ Clause
- ➔ Word

} ***Tokens***

A seconda del problema specifico la trasformazione del testo in tokens sarà a grana più o meno fine. Per esempio nella ***sentiment analysis*** si usa spesso segmentare un testo in clausole, dove ogni clausola è una parte di una frase che esprime una opinione.

NLTK: Segmentation

In altri casi, il testo è disponibile solo come flusso di caratteri. Prima di tokenizzare il testo in parole, dobbiamo segmentarlo in frasi. NLTK facilita questo includendo il segmento di frase Punkt (Kiss & Strunk, 2006). Ecco un esempio del suo uso nel segmentare il testo di un romanzo. (Si noti che se i dati interni del segmenter sono stati aggiornati al momento della lettura, verrà visualizzato un output diverso):

```
>>> text = nltk.corpus.gutenberg.raw('chesterton-thursday.txt')
>>> sents = nltk.sent_tokenize(text)
>>> pprint.pprint(sents[79:89])
['"Nonsense!"',
 'said Gregory, who was very rational when anyone else\nattempted paradox.',
 '"Why do all the clerks and navvies in the\n'
 'railway trains look so sad and tired, so very sad and tired?',
 'I will\ntell you.',
 'It is because they know that the train is going right.',
 'It\n'
 ...]
```

NLTK: Segmentation

Si noti che questo esempio è in realtà una singola frase, riportando il discorso di Lucian Gregory. Tuttavia, il discorso citato contiene diverse frasi, e queste sono state divise in singole stringhe. Questo è un comportamento ragionevole per la maggior parte delle applicazioni.

La segmentazione della frase è difficile perché il periodo viene utilizzato per contrassegnare le abbreviazioni e alcuni punti contrassegnano contemporaneamente un'abbreviazione e terminano una frase, come spesso accade con gli acronimi come gli U.S.A.

NLTK: TF-IDF

Il modulo ***nltk.text*** fornisce una classe apposita per la definizione di un collezione di documenti, ***TextCollection***, attraverso la quale è possibile ottenere le misure di ***term frequency***, ***inverse document*** frequency e ***tf-idf***.

```
>>> from nltk.book import text1, text2, text3
>>> from nltk.text import TextCollection
>>> mytexts = TextCollection([text1, text2, text3])
>>> mytexts.tf('interested', text1)
7.66815301033e-06
>>> mytexts.idf('interested')
0.405465108108
>>> mytexts.tf_idf('interested', text1)
3.10916848932e-06
```

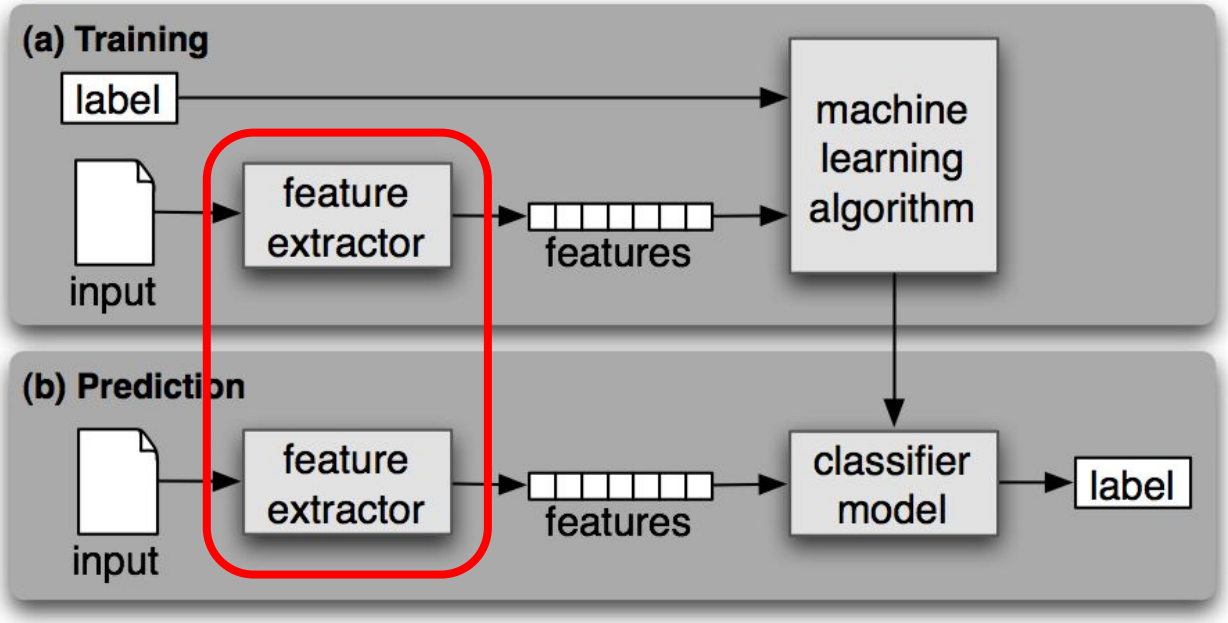
NLTK: Machine Learning

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

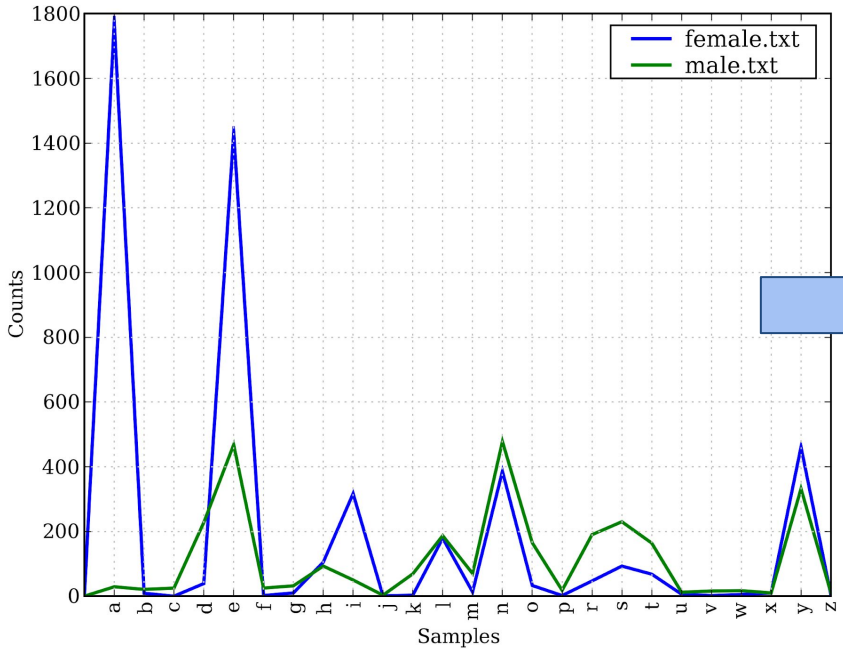
NLTK: Supervised Classification



La classificazione ha il compito di scegliere l'etichetta (classe) corretta da associare ad un determinato input. Nelle attività di classificazione supervisionata, ogni input viene considerato indipendentemente da tutti gli altri, e il set di etichette iniziale (classi) viene definito in anticipo. L'esempio tipico è il filtro anti-spam che associa l'etichetta ***spam***, ***non-spam*** alle email di un account di posta.

NLTK: Supervised Classification

Prendiamo l'esempio di gender classification, cioè dato il nome di una persona vogliamo indovinare il genere. Abbiamo già visto che alcuni nomi maschili e femminili hanno una maggiore probabilità (frequenza) di terminare con una determinata lettera.



La lettera finale diventa la nostra unica feature.

NLTK: Supervised Classification

Creiamo una funzione per estrarre la nostra unica feature.

```
>>> def gender_features(word):
...     return {'last_letter': word[-1]}
>>> gender_features('Shrek')
{'last_letter': 'k'}
```

Creiamo poi il nostro ***dataset*** sfruttando il corpus dei nomi.

```
>>> from nltk.corpus import names
>>> labeled_names = ([(name, 'male') for name in names.words('male.txt')] +
... [(name, 'female') for name in names.words('female.txt')])
>>> import random
>>> random.shuffle(labeled_names)
```

NLTK: Supervised Classification

Creiamo ora i nostri training e testing sets e addestriamo il ***NaiveBayesCassifier***.

```
>>> featuresets = [(gender_features(n), gender) for (n, gender) in
labeled_names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
```

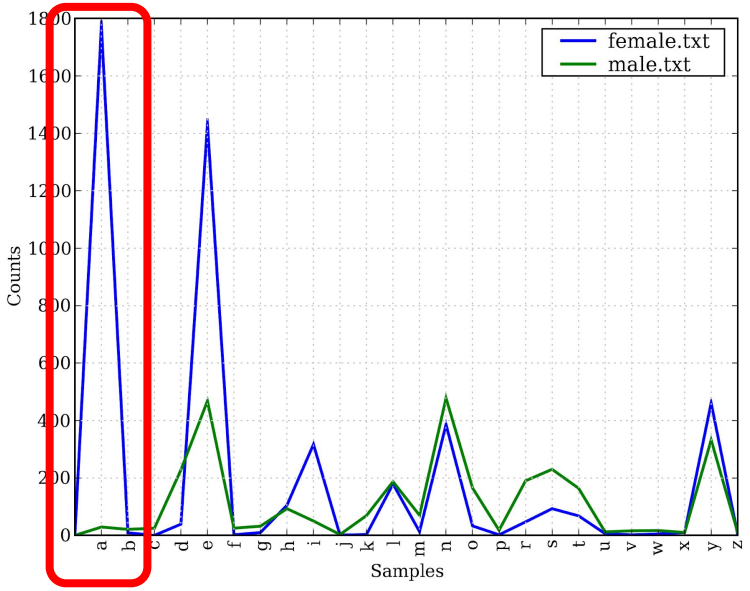
Adesso possiamo classificare un nome trasformandolo nel vettore di feature.

```
>>> classifier.classify(gender_features('Neo'))
'male'
>>> classifier.classify(gender_features('Trinity'))
'female'
>>> print(nltk.classify.accuracy(classifier, test_set))
0.77
```

NLTK: Supervised Classification

Vediamo quali sono le features più sono utili per discriminare tra le due classi.

```
>>> classifier.show_most_informative_features(3)
Most Informative Features
             last_letter = 'a'            female : male   =     33.2 : 1.0
             last_letter = 'k'              male : female =     32.6 : 1.0
             last_letter = 'p'              male : female =     19.7 : 1.0
```



NLTK: Supervised Classification

Esercizio

Ripetere l'esempio precedente utilizzando il ***nltk.DecisionTreeClassifier*** come classificatore.

Verificare l'***accuracy***, quale dei due classificatori è il migliore?

Utilizzare il metodo ***pseudocode*** della classe ***DecisionTreeClassifier*** per visualizzare le regole di scelta del classificatore.

NLTK: Supervised Classification

Selezionare le caratteristiche rilevanti e decidere come codificarle per un metodo di apprendimento può avere un impatto enorme sulla capacità del metodo di apprendimento di estrarre un buon modello.

Gran parte del lavoro interessante nella costruzione di un classificatore sta nel decidere quali features potrebbero essere rilevanti e come possiamo rappresentarle. Sebbene sia spesso possibile ottenere prestazioni decenti utilizzando un set di funzionalità abbastanza semplice, di solito si ottengono guadagni significativi utilizzando caratteristiche costruite con cura basate su una conoscenza approfondita del dominio in questione.

Tipicamente, gli estrattori di feature sono costruiti attraverso un processo trial-and-error, guidato da intuizioni su quali informazioni sono rilevanti per il problema.
È normale iniziare con un approccio "kitchen sink", che include tutte le funzionalità a cui è possibile pensare e quindi controllare per vedere quali funzionalità sono effettivamente utili

NLTK: Supervised Classification

```
def gender_features2(name):
    features = {}
    features["first_letter"] = name[0].lower()
    features["last_letter"] = name[-1].lower()
    for letter in 'abcdefghijklmnopqrstuvwxyz':
        features["count({})".format(letter)] = name.lower().count(letter)
        features["has({})".format(letter)] = (letter in name.lower())
    return features
```

Ripetiamo l'esercizio precedente includendo tutte le nuove feature che ci vengono in mente.

- ➔ Numero di occorrenze di ogni lettere
- ➔ Presenza di una lettera dell'alfabeto

NLTK: Supervised Classification

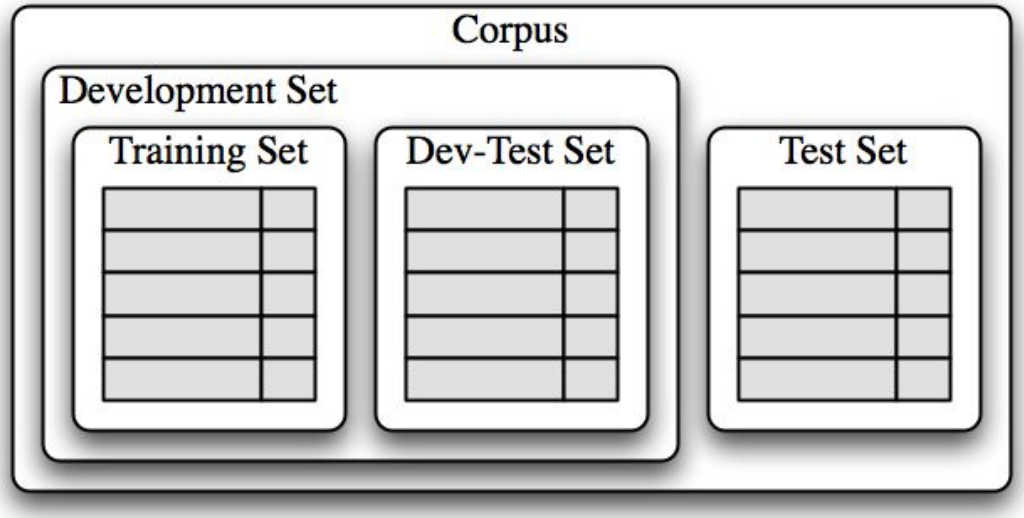
Tuttavia, di solito ci sono dei limiti al numero di features da usare con un dato algoritmo di addestramento (se si forniscono troppe features, allora l'algoritmo avrà una maggiore possibilità di basarsi su idiosincrasie del training set invece di generalizzare).

Questo problema è noto come ***overfitting*** e può essere particolarmente problematico quando si lavora con training set di piccole dimensioni.
Ad esempio, se addestriamo un classificatore di Bayes usando le nuove features, si utilizza un training set relativamente piccolo, risultando in un sistema la cui accuratezza è inferiore di circa 1% rispetto all'accuratezza di un classificatore che presta solo attenzione al lettera finale di ciascun nome.

```
>>> featuresets = [(gender_features2(n), gender) for (n, gender) in labeled_names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print(nltk.classify.accuracy(classifier, test_set))
0.768
```

NLTK: Supervised Classification

Una volta scelto un set iniziale di features, un metodo molto produttivo per perfezionare il set di features è l'***error analysis***. Innanzitutto, selezioniamo un ***development set***, contenente i dati del corpus per la creazione del modello. Questo set di documenti viene quindi suddiviso in ***training*** e ***test set***.




```
>>> train_names = labeled_names[1500:]
>>> devtest_names = labeled_names[500:1500]
>>> test_names = labeled_names[:500]
```

NLTK: Supervised Classification

Il training set viene utilizzato per addestrare il modello e il test set di sviluppo viene utilizzato per eseguire l'***error analysis***.
Il test set serve per la valutazione finale del modello. Per i motivi discussi di seguito, è importante utilizzare un test set separato per l'analisi degli errori, piuttosto che utilizzare semplicemente il set di test.

```
>>> train_set = [(gender_features(n), gender) for (n, gender) in train_names]
>>> devtest_set = [(gender_features(n), gender) for (n, gender) in
devtest_names]
>>> test_set = [(gender_features(n), gender) for (n, gender) in test_names]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print(nltk.classify.accuracy(classifier, devtest_set))
0.75
```

NLTK: Supervised Classification

Usando il set di test di sviluppo, possiamo generare una lista degli errori che il classificatore fa quando si predice i sessi dei nomi:

```
>>> errors = []
>>> for (name, tag) in devtest_names:
...     guess = classifier.classify(gender_features(name))
...     if guess != tag:
...         errors.append( (tag, guess, name) )
```

Possiamo quindi esaminare singoli casi di errore in cui il modello ha predetto l'etichetta sbagliata e cercare di determinare quali ulteriori informazioni potrebbero consentirgli di prendere la decisione giusta (o quali informazioni esistenti lo stanno ingannando nel prendere la decisione sbagliata). Il set di funzioni può quindi essere regolato di conseguenza. Il classificatore dei nomi che abbiamo creato genera circa 100 errori sul corpus di test di sviluppo:

NLTK: Supervised Classification

```
>>> for (tag, guess, name) in sorted(errors):
...     print('correct={:<8} guess={:<8s} name={:<30}'.format(tag, guess, name))
correct=female   guess=male     name=Abigail
  ...
correct=female   guess=male     name=Cindelyn
  ...
correct=female   guess=male     name=Katheryn
correct=female   guess=male     name=Kathryn
  ...
correct=male     guess=female   name=Aldrich
  ...
correct=male     guess=female   name=Mitch
  ...
correct=male     guess=female   name=Rich
```

NLTK: Supervised Classification

Guardando attraverso questa lista di errori si rende chiaro che alcuni suffissi con più di una lettera possono essere indicativi dei sessi dei nomi.

Ad esempio, i nomi che terminano in yn sembrano essere prevalentemente femminili, nonostante il fatto che i nomi che terminano in n tendano ad essere maschili; e i nomi che terminano in ch sono di solito maschi, anche se i nomi che finiscono in h tendono ad essere femminili.

Pertanto modifichiamo il nostro estrattore di funzioni per includere funzionalità per i suffissi di due lettere:

```
>>> def gender_features(word):
...     return {'suffix1': word[-1:],
...             'suffix2': word[-2:]}
```

NLTK: Supervised Classification

Ricostruendo il classificatore con il nuovo estrattore di feature, vediamo che le prestazioni del set di dati di sviluppo dei test migliorano di quasi 2 punti percentuali (dal 76,5% al 78,2%):

```
>>> train_set = [(gender_features(n), gender) for (n, gender) in train_names]
>>> devtest_set = [(gender_features(n), gender) for (n, gender) in
devtest_names]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print(nltk.classify.accuracy(classifier, devtest_set))
0.782
```

NLTK: Supervised Classification

Questa procedura di ***error analysis*** può quindi essere ripetuta, controllando gli errori di classificazione del classificatore raffinato per poi includere le migliorie nella iterazione successiva.

Ogni volta che viene ripetuta la procedura di ***error analysis***, dovremmo selezionare una divisione training/test di sviluppo differente, per garantire che il classificatore non inizi a riflettere le idiosincrasie nel test set di sviluppo e quindi vada in ***overfitting***.

Ma una volta che abbiamo utilizzato il test set di sviluppo per aiutarci a sviluppare il modello, non possiamo più fidarci del fatto che ci fornirà un'idea precisa di come il modello si comporterebbe correttamente con i nuovi dati.

È quindi importante mantenere il test set separato e inutilizzato fino al completamento dello sviluppo del modello.

A quel punto, possiamo usare il test set per valutare ***l'efficacia*** del nostro modello sui nuovi valori di input.

NLTK: Document Classification

Abbiamo visto diversi esempi di corpora in cui i documenti sono stati etichettati con categorie.
Usando questi corpora, possiamo costruire classificatori che taggano automaticamente nuovi documenti con etichette di categorie appropriate. Innanzitutto, costruiamo un elenco di documenti, etichettati con le categorie appropriate. Per questo esempio, abbiamo scelto il Movie Reviews Corpus, che classifica ogni recensione come positiva o negativa.

```
>>> from nltk.corpus import movie_reviews
>>> documents = [(list(movie_reviews.words(fileid)), category)
...              for category in movie_reviews.categories()
...              for fileid in movie_reviews.fileids(category)]
>>> random.shuffle(documents)
```

NLTK: Document Classification

Definiamo un ***feature extractor*** per i documenti, il classificatore saprà a quali caratteristiche dei dati deve prestare attenzione. Per l'identificazione del ***document topic***, possiamo definire una funzione per ogni parola, indicando se il documento contiene quella parola. Per limitare il numero di features che il classificatore deve elaborare, iniziamo costruendo un elenco delle 2000 parole più frequenti nel corpus generale. Possiamo quindi definire il **feature extractor** che controlla semplicemente se ciascuna di queste parole sia presente in un determinato documento.

```
def document_features(document):
  all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
  word_features = list(all_words)[:2000]
  document_words = set(document)
  features = {}
   for word in word_features:
      features['contains({})'.format(word)] = (word in document_words)
   return features
```

NLTK: Document Classification

```
>>> print(document_features(movie_reviews.words('pos/cv957_8737.txt')))
{'contains(waste)': False, 'contains(lot)': False, ...}
```

Addestriamo un classificatore per etichettare le nuove recensioni di film. Per verificare l'affidabilità del classificatore risultante, calcoliamo la sua accuracy sul test set.

```
>>> featuresets = [(document_features(d), c) for (d,c) in documents]
>>> train_set, test_set = featuresets[100:], featuresets[:100]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print(nltk.classify.accuracy(classifier, test_set))
0.81
```

NLTK: Document Classification

E ancora una volta, possiamo usare ***show_most_informative_features***() per scoprire quali features il classificatore ha trovato più utili.

```
>>> classifier.show_most_informative_features(5)
Most Informative Features
  contains(outstanding) = True              pos : neg    =     11.1 : 1.0
       contains(seagal) = True              neg : pos    =      7.7 : 1.0
  contains(wonderfully) = True              pos : neg    =      6.8 : 1.0
        contains(damon) = True              pos : neg    =      5.9 : 1.0
       contains(wasted) = True              neg : pos    =      5.8 : 1.0
```

Apparentemente in questo corpus, una recensione che menziona "Seagal" ha quasi 8 volte più probabilità di essere negativa che positiva, mentre una recensione che menziona "Damon" ha circa 6 volte più probabilità di essere positiva.

NLTK: Part-of-Speech Tagging

Universal Part-of-Speech Tagset è un set di ***tags*** standard con cui è possibile taggare una determinata parola.

Tag	Meaning	English Examples
ADJ	adjective	*new, good, high, special, big, local*
ADP	adposition	*on, of, at, with, by, into, under*
ADV	adverb	*really, already, still, early, now*
CONJ	conjunction	*and, or, but, if, while, although*
DET	determiner, article	*the, a, some, most, every, no, which*
NOUN	noun	*year, home, costs, time, Africa*
NUM	numeral	*twenty-four, fourth, 1991, 14:24*
PRT	particle	*at, on, out, over per, that, up, with*
PRON	pronoun	*he, their, her, its, my, I, us*
VERB	verb	*is, say, told, given, playing, would*
.	punctuation marks	*. , ; !*
X	other	*ersatz, esprit, dunno, gr8, univeristy*

NLTK: Part-of-Speech Tagging

Il Brown's Corpus contiene un parte di documenti taggati utilizzando lo ***universal tagset***.

```
>>> from nltk.corpus import brown
>>> brown_news_tagged = brown.tagged_words(categories='news',
tagset='universal')
>>> tag_fd = nltk.FreqDist(tag for (word, tag) in brown_news_tagged)
>>> tag_fd.most_common()
[('NOUN', 30640), ('VERB', 14399), ('ADP', 12355), ('.', 11928), ('DET', 11389),
 ('ADJ', 6706), ('ADV', 3349), ('CONJ', 2717), ('PRON', 2535), ('PRT', 2264),
 ('NUM', 2166), ('X', 106)]
```

NLTK: Part-of-Speech Tagging

Gli ***Unigram Tagger*** si basano su un semplice algoritmo statistico: per ogni token, assegna il tag che è più probabile per quel particolare token. Ad esempio, assegnerà il tag JJ a qualsiasi occorrenza della parola ***frequent***, poiché ***frequent*** è usato come aggettivo (ad esempio una parola ***frequent***) più spesso di quanto sia usato come verbo (ad esempio, frequento questo caffè).

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
>>> unigram_tagger = nltk.UnigramTagger(brown_tagged_sents)
>>> unigram_tagger.tag(brown_sents[2007])
[('Various', 'JJ'), ('of', 'IN'), ('the', 'AT'), ('apartments', 'NNS'),
('are', 'BER'), ('of', 'IN'), ('the', 'AT'), ('terrace', 'NN'), ('type', 'NN'),
(',', ','), ('being', 'BEG'), ('on', 'IN'), ('the', 'AT'), ('ground', 'NN'),
('floor', 'NN'), ('so', 'QL'), ('that', 'CS'), ('entrance', 'NN'), ('is', 'BEZ'),
('direct', 'JJ'), ('.', '.')]
>>> unigram_tagger.evaluate(brown_tagged_sents)
0.9349006503968017
```

NLTK: Esercitazione 5

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

Fare il checkout del branch 05-esercitazione.

NLTK: riferimenti

- http://www.nltk.org/book/ch03.html
- http://www.nltk.org/book/ch04.html
- http://www.nltk.org/book/ch05.html
- http://www.nltk.org/book/ch06.html
- Natural Language Processing with Python(ISBN-13: 978-0596516499)