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

Natural Language ToolKit

Natural Language Toolkit

- E' una libreria Python che fornisce i moduli per l'elaborazione di testo, classificazione, tokenizing, stemming, tagging e analisi.

- E' molto di più, è una comunità open source creata da due accademici **Steven Bird** e **Edward Loper** nel **Department of Computer and Information Science at the University of Pennsylvania**.

- E' stata creata come supporto per l'insegnamento del ***Natural Language Processing*** e ***Machine Learning***

- Include diverse aree:
  - linguistics
  - cognitive science
  - artificial intelligence
  - information retrieval
  - machine learning

Natural Language Toolkit

- NLTK comprende nelle sue librerie:
  - classification
  - tokenization
  - stemming
  - tagging
  - parsing
  - semantic reasoning functionalities

Natural Language Toolkit

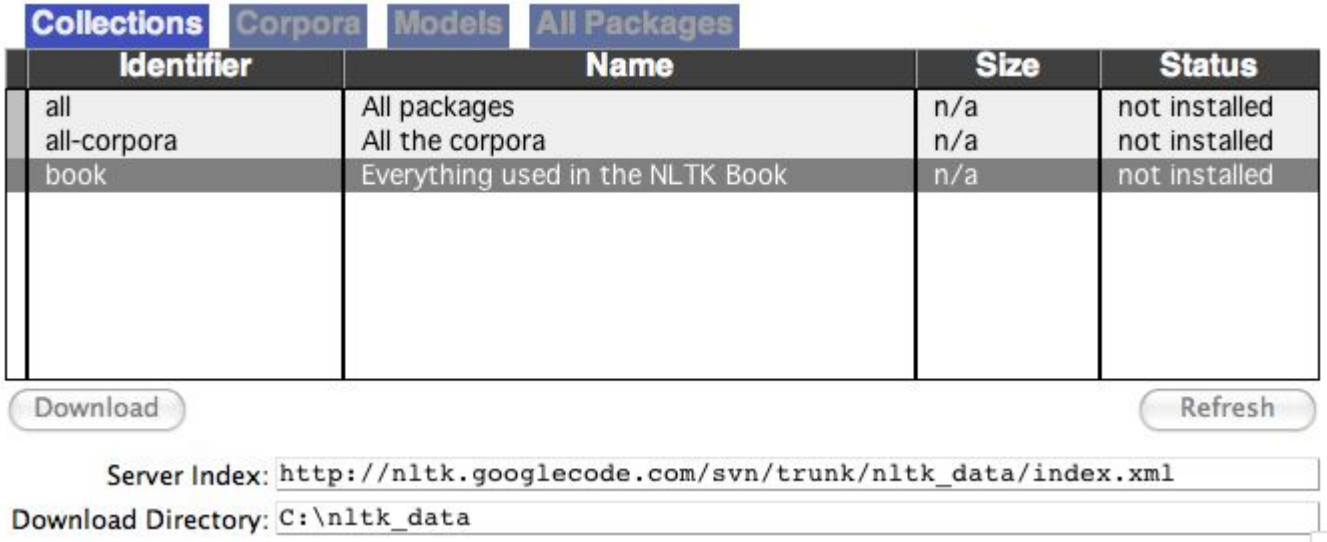
Per seguire le esercitazioni fare il checkout del seguente progetto da GitHub.

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

Seguire le istruzione per la creazione del ***virtualenv*** e fare il checkout del branch ***05-esercitazione.***

Natural Language Toolkit

```
>>> import nltk
>>> nltk.download() # viene aperta la finestra di download
>>> nltk.download('book') # scarica il book corpus da console Python
```



NLTK

Importando tutti i testi disponibili nel modulo nltk.books con la seguente istruzione.

```
>>> from nltk.book import *
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

NLTK

Importando tutti i testi disponibili nel modulo nltk.books, questi diventano disponibili, per esempio le variabili ***text1, text2*** possono essere utilizzare all'interno del nostro modulo.

```
>>> text1
<Text: Moby Dick by Herman Melville 1851>
>>> text2
<Text: Sense and Sensibility by Jane Austen 1811>
```

NLTK: ricerca testuale

Esistono molti modi per esaminare il **contesto** di un testo oltre a leggerlo semplicemente. La ***concordance view*** ci mostra ogni occorrenza di una determinata parola, insieme ad un **contesto**. Qui vediamo la parola ***"monstrous"*** in Moby Dick. Chiamiamo il metodo ***concordance*** sulla variabile ***text1***.

```
>>> text1.concordance("monstrous")
Displaying 11 of 11 matches:
ong the former , one was of a most monstrous size . ... This came towards us ,
ON OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have r
ll over with a heathenish array of monstrous clubs and spears . Some were thick
d as you gazed , and wondered what monstrous cannibal and savage could ever hav
...
```

NLTK: ricerca testuale

Possiamo confrontare l'uso di un termine in diversi testi. Utilizzando il metodo ***similar*** possiamo confrontare la parola ***"monstrous"*** in *Moby Dick* e in *Sense and Sensibility*. Chiamiamo il metodo ***similar*** sulla variabile ***text1*** e ***text2***. Similar restituisce tutte le parole che più comunemente appaiono nello stesso contesto della parola cercata.

```
>>> text1.similar("monstrous")
mean part maddens doleful gamesome subtly uncommon careful untoward
exasperate loving passing mouldy christian few true mystifying
imperial modifies contemptible
>>> text2.similar("monstrous")
very heartily so exceedingly remarkably as vast a great amazingly
extremely good sweet
```

Austen usa questa parola in modo molto diverso da Melville; per lei, il mostruoso ha connotazioni positive, e talvolta funziona come un intensificatore come la parola molto.

NLTK: ricerca testuale

Il metodo ***common_contexts*** ci permette di esaminare solo i contesti che sono condivisi da due o più parole, come ***monstrous*** e ***very***. Il metodo prende in ingresso una lista di termini.

```
>>> text2.common_contexts(["monstrous", "very"])
a_pretty is_pretty am_glad be_glad a_lucky
```

Possiamo rilevare automaticamente che una parola particolare occorre in un testo e visualizzare alcune parole che occorrono nello stesso contesto.

NLTK: ricerca testuale

L'informazione ***posizionale*** di un termine può essere visualizzata utilizzando un ***grafico di dispersione***. È possibile produrre questo grafico come mostrato di seguito.

```
>>> text4.dispersion_plot(["citizens", "democracy", "freedom", "duties",
"America"])
```

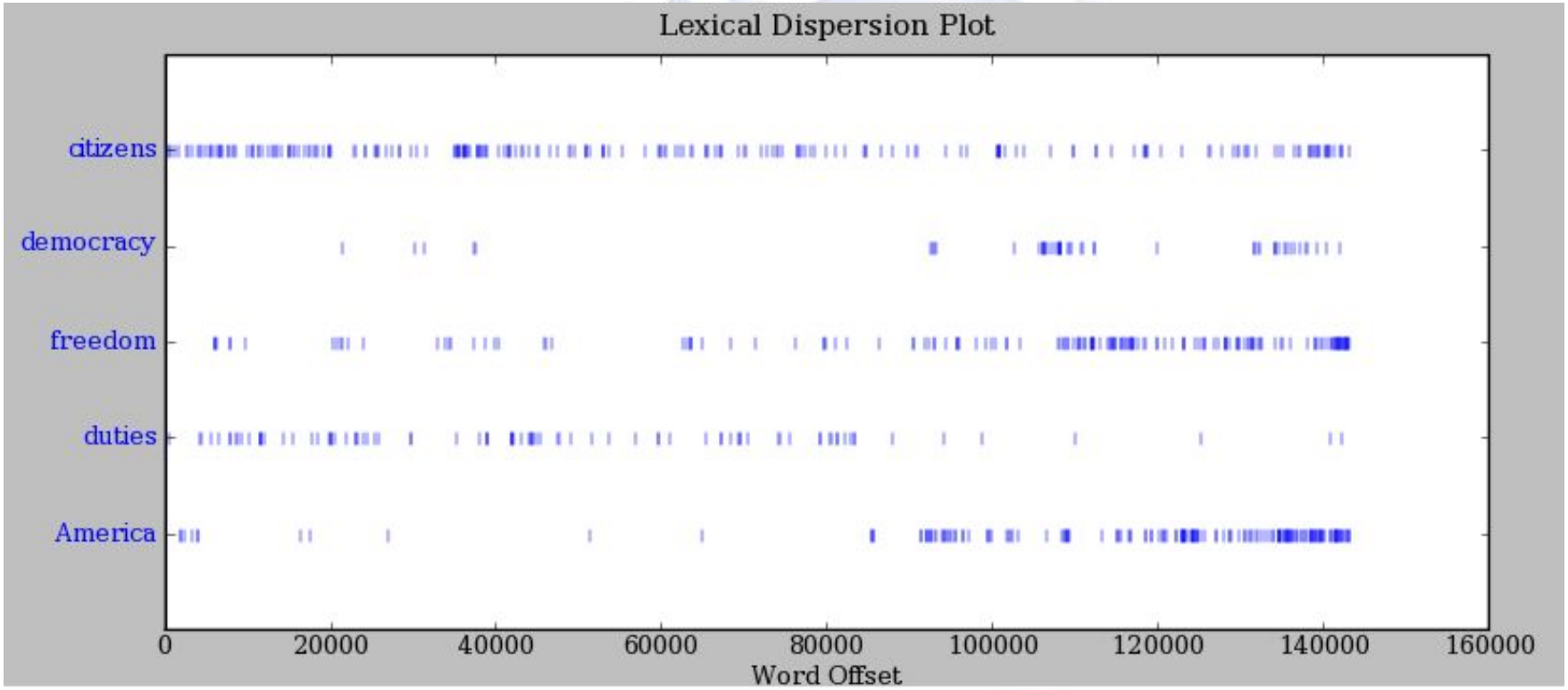
Il metodo ***dispersion_plot*** prende in ingresso una lista di termini.

Il ***text4*** è un testo artificiale costruito unendo i testi del Corpus Inaugural Address.

Vediamo alcuni modelli sorprendenti di uso delle parole negli ultimi 220 anni

NLTK: ricerca testuale

Ogni striscia rappresenta un'istanza di una parola e ogni riga rappresenta l'intero testo.



NLTK: generare un testo

Per fare ciò, creiamo un oggetto ***Text()*** sulla variabile ***textEintein*** che definiamo.

```
>>> text_einstein = """
Solo due cose sono infinite: l'universo e la stupidità umana, riguardo l'universo ho ancora dei dubbi.
La logica vi porterà da A a B. L'immaginazione vi porterà dappertutto.
Non so con quali armi si combatterà la Terza guerra mondiale, ma la Quarta sì: con bastoni e pietre.
Se l'ape scomparisse dalla faccia della terra, all'uomo non resterebbero che quattro anni di vita
Ognuno è un genio. Ma se si giudica un pesce dalla sua abilità di arrampicarsi sugli alberi lui passerà tutta la sua vita a credersi stupido
Io appartengo all'unica razza che conosco, quella umana.
Tutto è determinato… da forze sulle quali non abbiamo alcun controllo. Lo è per l'insetto come per le stelle. Esseri umani, vegetali, o polvere cosmica,
tutti danziamo al ritmo di una musica misteriosa, suonata in lontananza da un pifferaio invisibile.
Tutti sanno che una cosa è impossibile da realizzare, finché arriva uno sprovveduto che non lo sa e la inventa.
"""
>>> text10 = nltk.word_tokenize(text_einstein)
>>> text10 = nltk.Text(text10)
>>> text10.concordance("logica")
Displaying 1 of 1 matches:
' universo ho ancora dei dubbi . La logica vi porterà da A a B. L ' immaginazi
None
```

NLTK: contare

La cosa più ovvia che emerge dagli esempi precedenti è che essi differiscono nel vocabolario che usano.

Vediamo come contare le parole di un testo in una varietà di modi utili.

Iniziamo con la lunghezza di un testo dall'inizio alla fine, in termini di parole e simboli di punteggiatura che appaiono.
Usiamo il metodo ***len*** per ottenere la lunghezza di qualcosa, che applichiamo per esempio al libro della Genesi:

```
>>> len(text3)
44764
```

NLTK: contare

Quindi Genesis ha 44.764 parole e simboli di punteggiatura, o "token".
Un token è il nome tecnico di una sequenza di caratteri - come peloso, suo o :) - che vogliamo trattare come un gruppo.
Quando contiamo il numero di token in un testo, diciamo, la frase “essere o non essere”, stiamo contando le occorrenze di queste sequenze.
Pertanto, nella nostra frase esemplificativa ci sono due occorrenze di essere, e uno ciascuno di o e non. Ma ci sono solo quattro voci di vocabolario distinte in questa frase.
Quante parole **distinte** contiene il libro della Genesi?
Per risolvere questo problema in Python, dobbiamo porre la domanda in modo leggermente diverso.
Il vocabolario di un testo è solo l'insieme di token che utilizza, poiché in un set tutti i duplicati sono compressi insieme.
In Python possiamo ottenere gli elementi di vocabolario di text3 con il comando: ***set (text3)***.

NLTK: contare

```
>>> sorted(set(text3))
['!', '"', '(', ')', ',', ',)', '.', '.)', ':', ';', ';)', '?', '?)',
'A', 'Abel', 'Abelmizraim', 'Abidah', 'Abide', 'Abimael', 'Abimelech',
'Abr', 'Abrah', 'Abraham', 'Abram', 'Accad', 'Achbor', 'Adah', ...]
>>> len(set(text3))
2789
```

Ora, calcoliamo una misura della ***ricchezza lessicale*** del testo.

```
>>> len(set(text3)) / len(text3)
0.0623045304262353 7
```

Vediamo che il numero di parole distinte è solo il **6%** del numero totale di parole, o equivalentemente che ogni parola viene usata **16** volte in media.

NLTK: contare

Successivamente, concentriamoci su determinate parole. Possiamo contare la frequenza con cui una parola si verifica in un testo e calcolare quale percentuale del testo è occupata da una parola specifica:

```
>>> text3.count("smote")
5
>>> 100 * text4.count('a') / len(text4)
1.4643016433938312
```

Esercizio:
Quante volte la parola ***lol*** appare nel **text5**?
Quanto è una percentuale del numero totale di parole in questo testo?

NLTK: contare

Creiamo i metodi "***lexicalDiversity***" o "**percentage**", e associarlo con un blocco di codice.

```
>>> def lexicalDiversity(text):
...     return len(set(text)) / len(text)
>>> def percentage(count, total):
...     return 100 * count / total
```

```
>>> lexicalDiversity(text3)
0.06230453042623537
>>> lexicalDiversity(text5)
0.13477005109975562
>>> percentage(4, 5)
80.0
>>> percentage(text4.count('a'), len(text4))
1.4643016433938312
```

NLTK: testo come lista di parole

Cos'è un testo? Ad un livello, è una sequenza di simboli su una pagina come questa.
Ad un altro livello, è una sequenza di capitoli, composta da una sequenza di sezioni, in cui ogni sezione è una sequenza di paragrafi e così via.
Tuttavia, per i nostri scopi, un testo è nient'altro che una sequenza di parole e punteggiatura.
Ecco come rappresentiamo il testo in Python, in questo caso la frase di apertura di Moby Dick:

```
>>> sent1 = ['Call', 'me', 'Ishmael', '.']
>>> sent1
['Call', 'me', 'Ishmael', '.']
>>> len(sent1)
4
>>> lexical_diversity(sent1)
1.0
```

NLTK: testo come lista di parole

Altre liste sono state predefinite da NLTK, una per la frase di apertura di ciascuno dei nostri testi, ***sent2*** ... ***sent9***.

```
>>> sent2
['The', 'family', 'of', 'Dashwood', 'had', 'long',
'been', 'settled', 'in', 'Sussex', '.']
>>> sent3
['In', 'the', 'beginning', 'God', 'created', 'the',
'heaven', 'and', 'the', 'earth', '.']
```

Le liste possono essere concatenate tra loro:

```
>>> sent4 + sent1
['Fellow', '-', 'Citizens', 'of', 'the', 'Senate', 'and', 'of', 'the',
'House', 'of', 'Representatives', ':', 'Call', 'me', 'Ishmael', '.']
```

NLTK: testo come lista di parole

Con un po 'di pazienza, possiamo scegliere la 1°, la 173° o la 14,278 ° parola in un testo.
Analogamente, possiamo identificare gli elementi di un elenco Python in base al loro ordine di occorrenza nell'elenco. Il numero che rappresenta questa posizione è l'***indice*** dell'oggetto. Istruiamo Python per mostrarci l'oggetto con indice 173 in un testo scrivendo il nome del testo seguito dall'indice all'interno di parentesi quadre:

```
>>> text5[16715:16735]
['U86', 'thats', 'why', 'something', 'like', 'gamefly', 'is', 'so', 'good',
'because', 'you', 'can', 'actually', 'play', 'a', 'full', 'game', 'without',
'buying', 'it']

>>> text6[1600:1625]
['We', """, 're', 'an', 'anarcho', '-', 'syndicalist', 'commune', '.', 'We',
'take', 'it', 'in', 'turns', 'to', 'act', 'as', 'a', 'sort', 'of', 'executive',
'officer', 'for', 'the', 'week']
```

NLTK: Frequency Distributions

Come possiamo identificare automaticamente le parole che sono più informative sull'argomento e il genere del testo?
Troviamo ad esempio le 50 parole più frequenti di un libro.
Un metodo sarebbe quello di mantenere un conteggio per ogni voce del vocabolario, come quella mostrata:

Word Tally

the	𝍸 𝍸 𝍸 ||||
been	𝍸 𝍸 |
message	||||
persevere	|
nation	𝍸 |||

Il conteggio avrebbe bisogno di migliaia di righe, e sarebbe un processo estremamente noioso - così noioso che preferiremmo assegnare il compito a una macchina.

NLTK: Frequency Distributions

La ***frequency distribution*** è una "distribuzione" perché ci dice come il numero totale di 'token' nel testo è distribuito tra gli elementi del vocabolario. Poiché spesso abbiamo bisogno di distribuzioni di frequenza nell'elaborazione del linguaggio, ***NLTK*** fornisce supporto integrato il calcolo.
Usiamo la classe ***FreqDist*** per trovare le 50 parole più frequenti di Moby Dick:

```
>> > fdist1 = FreqDist(text1)
>>> print(fdist1)
< FreqDist  with 19317 samples and 260819 outcomes >
>>> fdist1.most_common(50)
   [(',', 18713), ('the', 13721), ('.', 6862),
('of', 6536), ('and', 6024),  ('a', 4569),
('to', 4542), (';', 4072), ('in', 3916),
('that', 2982), ('now', 646), ('which', 640),
('?', 637), ('me', 627), ('like', 624)]
>>> fdist1['whale']
   906
```

NLTK: Frequency Distributions

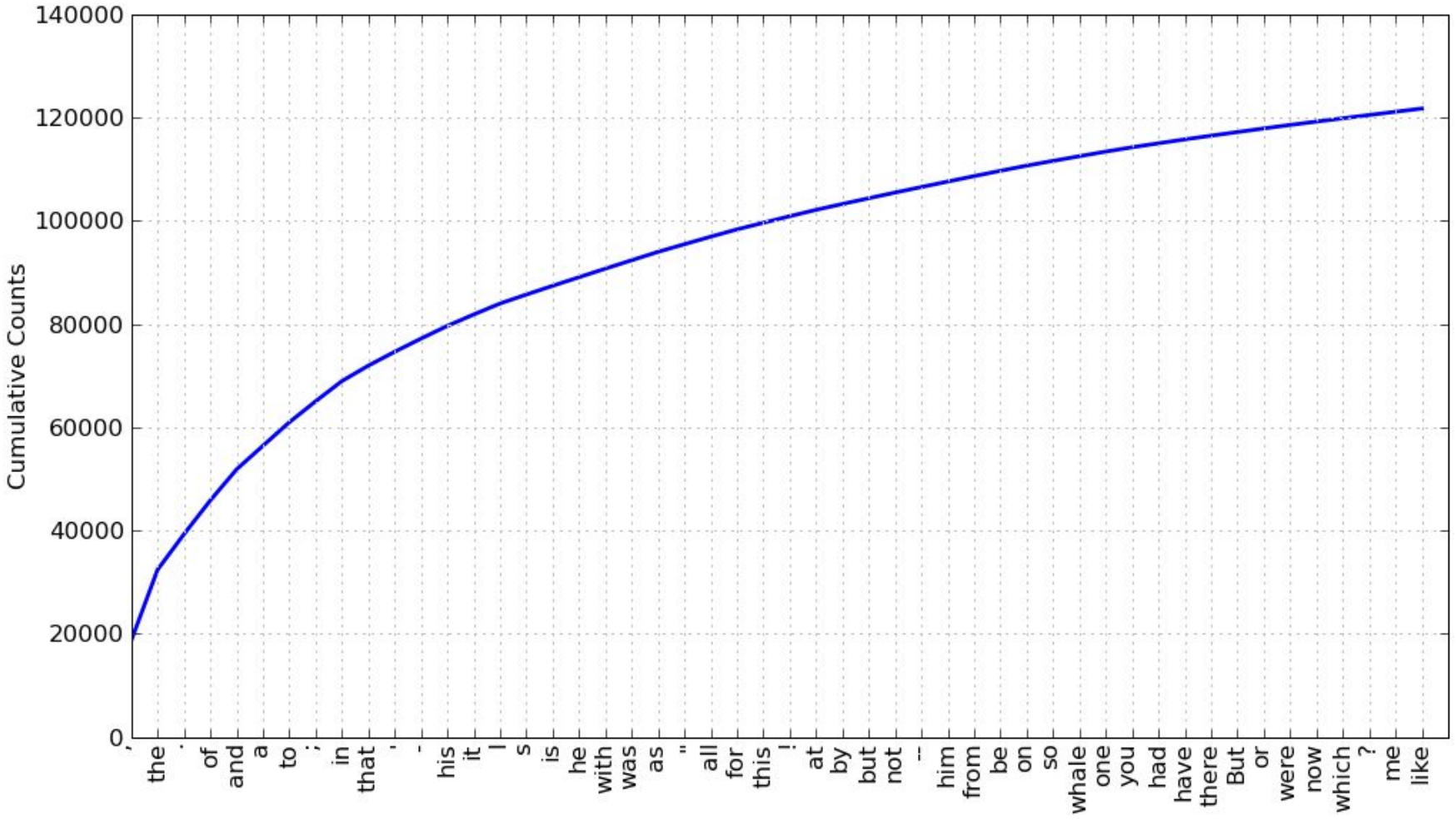
Qualche parola prodotta nell'ultimo esempio ci aiuta a cogliere l'argomento o il genere di questo testo?
Solo una parola, balena, è leggermente informativa! Si verifica oltre 900 volte.

Il resto delle parole non ci dice nulla sul testo; sono solo parole inglesi dette "***plumbing***".

Quale parte del testo è occupata da tali parole?
Possiamo generare un grafico di frequenza cumulativo per queste parole, usando:

```
>>>len(text1)
260819
>> > fdist1.plot (50, cumulative = True)
```

NLTK: Frequency Distributions



Queste 50 parole rappresentano quasi la metà del libro!

NLTK: testo come lista di parole

Se le parole frequenti non ci aiutano, che dire delle parole che si verificano una sola volta, le cosiddette ***hapaxes***?

hapax:

➔ "detto una volta sola" è una forma linguistica (parola o espressione), che compare una sola volta nell'ambito di un testo

Si possono visualizzare digitando ***fdist1.hapaxes()***.

[u'funereal', u'unscientific', u'prefix', u'plaudits', u'woody', u'disobeying', u'Westers', u'DRYDEN', u'Untried', u'superficially', u'Western', u'Spurn', u'treasuries', u'powders', u'tinkerings', u'bolting', u'stabbed', u'elevations', u'ferreting', u'believers']

Questo elenco contiene lessicografo, cetologico, contrabbando, expostulations e circa 9.000 altri. Sembra che ci siano troppe parole rare e, senza vedere il contesto, probabilmente non possiamo indovinare il significato dell'hapax in ogni caso! Dal momento che né le parole frequenti né quelle rari aiutano, dobbiamo provare qualcos'altro.

NLTK: Fine-grained Selection of Words

Esaminiamo le lunghe parole di un testo; forse queste saranno più caratteristiche e informative.

Per questo adottiamo alcune notazioni dalla teoria degli insiemi. Vorremmo trovare le parole del vocabolario del testo di oltre 15 caratteri.

Chiamiamo questa proprietà P, così che P (w) è vera se e solo se w è lunga più di 15 caratteri. Ora possiamo esprimere le parole di interesse usando la notazione dell'insieme matematico.

➔ $\{w \mid w \in V \ \& \ P(w)\}$

➔ [w for w in V if p(w)]

Questo significa "l'insieme di tutti w tale che w è un elemento di V (il vocabolario) e w ha la proprietà P".

NLTK: Fine-grained Selection of Words

L'espressione Python corrispondente data precedentemente è la seguente (osservare quanto simili siano le due notazioni)

```
>>> V = set(text1)
>>> long_words = [w for w in V if len(w) > 15]
>>> sorted(long_words)
['CIRCUMNAVIGATION', 'Physiognomically', 'apprehensiveness', 'cannibalistically',
'characteristically', 'circumnavigating', 'circumnavigation', 'circumnavigations',
'comprehensiveness', 'hermaphroditical', 'indiscriminately', 'indispensableness',
'irresistibleness', 'physiognomically', 'preternaturalness', 'responsibilities',
'simultaneousness', 'subterraneousness', 'supernaturalness', 'superstitiousness',
'uncomfortableness', 'uncompromisedness', 'undiscriminating',
'uninterpenetratingly']
```

NLTK: Fine-grained Selection of Words

Torniamo al nostro compito di trovare parole che caratterizzano un testo. Si noti che le lunghe parole nel ***text4*** riflettono il focus nazionale - costituzionalmente, transcontinentale - mentre quelle del ***text5*** riflettono il suo contenuto informale: boooooooooooglyyyyyy e yuuuuuuuuuuuuummmmmmmmmmm.
Siamo riusciti a estrarre automaticamente le parole che caratterizzano un testo? Bene, queste parole molto lunghe sono spesso hapaxes (vale a dire, uniche) e forse sarebbe meglio trovare le parole **lunghe** che si verificano di **frequente**. Ciò sembra promettente poiché elimina le parole brevi frequenti (ed anche le parole lunghe e poco frequenti ad esempio antifilosofici). Qui ci sono tutte le parole del corpus chat che sono più lunghe di sette caratteri, che si verificano più di sette volte:

```
>>> fdist5 = FreqDist(text5)
>>> sorted([w for w in set(text5) if len(w) > 7 and fdist5[w] > 7])
['#14-19teens', '#talkcity_adults', '((((((((((', '.....', 'Question',
'actually', 'anything', 'computer', 'cute.-ass', 'everyone', 'football',
'innocent', 'listening', 'remember', 'seriously', 'something', 'together',
'tomorrow', 'watching']
```

NLTK: Collocations and Bigrams

Una collocazione è una sequenza di parole che si verificano insieme spesso. Quindi il “vino rosso” è una collocazione, mentre il vino no.

Una caratteristica delle collocazioni è che sono resistenti alla sostituzione con parole che hanno sensi simili; per esempio, il “vino marrone” sembra decisamente strano.

Per ottenere un controllo sulle collocazioni, iniziamo estraendo da un testo un elenco di coppie di parole, note anche come bigram. Questo è facilmente realizzabile con la funzione bigrams ():

```
>>> list(bigrams(['more', 'is', 'said', 'than', 'done']))
[('more', 'is'), ('is', 'said'), ('said', 'than'), ('than', 'done')]
```

NLTK: Collocations and Bigrams

La coppia di parole than-done è un bigram, e lo scriviamo in Python come ('than', 'done').
Le collocazioni sono essenzialmente solo dei bigram frequenti, tranne per il fatto che vogliamo prestare maggiore attenzione ai casi che riguardano parole rare.
In particolare, vogliamo trovare i bigram che si verificano più spesso di quanto ci aspetteremmo in base alla frequenza delle singole parole.
La funzione ***collocations()*** fa esattamente questo. Vedremo come funzionerà in seguito.

```
>>> text4.collocations()
United States; fellow citizens; four years; years ago; Federal
Government; General Government; American people; Vice President; Old
World; Almighty God; Fellow citizens; Chief Magistrate; Chief Justice;
God bless; every citizen; Indian tribes; public debt; one another;
>>> text8.collocations()
would like; medium build; social drinker; quiet nights; non smoker;
long term; age open; Would like; easy going; financially secure; fun
times; similar interests; Age open; weekends away; poss rship; well
```

NLTK: Collocations and Bigrams

Le collocazioni che emergono sono molto specifiche per il genere dei testi. Per trovare il “vino rosso” come collocazione, avremmo bisogno di elaborare un corpo di testo molto più ampio.

```
>>> text4.collocations()
United States; fellow citizens; four years; years ago; Federal
Government; General Government; American people; Vice President; Old
World; Almighty God; Fellow citizens; Chief Magistrate; Chief Justice;
God bless; every citizen; Indian tribes; public debt; one another;

>>> text8.collocations()
would like; medium build; social drinker; quiet nights; non smoker;
long term; age open; Would like; easy going; financially secure; fun
times; similar interests; Age open; weekends away; poss rship; well
```

NLTK: Collocations and Bigrams

Il conteggio delle parole è utile, ma possiamo contare anche altre cose. Ad esempio, possiamo esaminare la distribuzione delle lunghezze delle parole in un testo, creando un ***FreqDist*** da una lunga lista di numeri, dove ogni numero è la lunghezza della parola corrispondente nel testo:

```
>>> [len(w) for w in text1]
[1, 4, 4, 2, 6, 8, 4, 1, 9, 1, 1, 8, 2, 1, 4, 11, 5, 2, 1, 7, 6, 1, 3, 4, 5, 2, ...]
>>> fdist = FreqDist([len(w) for w in text1] )
>>> print(fdist)
<FreqDist with 19 samples and 260819 outcomes>
>>> fdist
FreqDist({3: 50223, 1: 47933, 4: 42345, 2: 38513, 5: 26597, 6: 17111, 7: 14399,
  8: 9966, 9: 6428, 10: 3528, ...})
```

Derivando un elenco delle lunghezze delle parole in text1, e ***FreqDist*** conta il numero di volte in cui ciascuna di queste si verifica. Il risultato è una distribuzione contenente un quarto di un milione di elementi, ognuno dei quali è un numero corrispondente a un token di parole nel testo.

NLTK: Collocations and Bigrams

Ma al massimo si contano solo 20 voci distinte, i numeri da 1 a 20, perché ci sono solo 20 diverse lunghezze di parole. Ad esempio, ci sono parole composte da un solo carattere, due caratteri, ..., venti caratteri, ma nessuno con ventuno o più caratteri. Ci si potrebbe chiedere quanto siano frequenti le diverse lunghezze della parola (ad es. Quante parole di lunghezza quattro compaiono nel testo, ci sono più parole di lunghezza cinque che lunghezza quattro, ecc.). Possiamo farlo come segue:

```
>>> fdist.most_common()
[(3, 50223), (1, 47933), (4, 42345), (2, 38513), (5, 26597), (6, 17111), (7, 14399),
(8, 9966), (9, 6428), (10, 3528), (11, 1873), (12, 1053), (13, 567), (14, 177),
(15, 70), (16, 22), (17, 12), (18, 1), (20, 1)]
>>> fdist.max()
3
>>> fdist[3]
50223
>>> fdist.freq(3)
0.192558824319
```

NLTK: Collocations and Bigrams

Da questo vediamo che la lunghezza più frequente è 3, e che le parole di lunghezza 3 rappresentano circa 50.000 (o 20%) delle parole che compongono il libro.
Un'ulteriore analisi della lunghezza delle parole potrebbe aiutarci a capire le differenze tra autori, generi o lingue.

```
>>> fdist.most_common()
[(3, 50223), (1, 47933), (4, 42345), (2, 38513), (5, 26597), (6, 17111), (7, 14399),
(8, 9966), (9, 6428), (10, 3528), (11, 1873), (12, 1053), (13, 567), (14, 177),
(15, 70), (16, 22), (17, 12), (18, 1), (20, 1)]
>>> fdist.max()
3
>>> fdist[3]
50223
>>> fdist.freq(3)
0.192558824319
```

NLTK: utilities

La seguente tabella riassume le funzioni definite nelle distribuzioni di frequenza.

Example	Description
`fdist = FreqDist(samples)`	create a frequency distribution containing the given samples
`fdist[sample] += 1`	increment the count for this sample
`fdist['monstrous']`	count of the number of times a given sample occurred
`fdist.freq('monstrous')`	frequency of a given sample
`fdist.N()`	total number of samples
`fdist.most_common(n)`	the n most common samples and their frequencies
`for sample in fdist:`	iterate over the samples
`fdist.max()`	sample with the greatest count
`fdist.tabulate()`	tabulate the frequency distribution
`fdist.plot()`	graphical plot of the frequency distribution
`fdist.plot(cumulative=True)`	cumulative plot of the frequency distribution
`fdist1 |= fdist2`	update `fdist1` with counts from `fdist2`
`fdist1 < fdist2`	test if samples in `fdist1` occur less frequently than in `fdist2`

NLTK: utilities

La seguente tabella riassume le funzioni definite per le stringhe.

Function	Meaning
`s.startswith(t)`	test if s starts with t
`s.endswith(t)`	test if s ends with t
`t in s`	test if t is a substring of s
`s.islower()`	test if s contains cased characters and all are lowercase
`s.isupper()`	test if s contains cased characters and all are uppercase
`s.isalpha()`	test if s is non-empty and all characters in s are alphabetic
`s.isalnum()`	test if s is non-empty and all characters in s are alphanumeric
`s.isdigit()`	test if s is non-empty and all characters in s are digits
`s.istitle()`	test if s contains cased characters and is titlecased (i.e. all words in s have initial capitals)

NLTK: utilities

Ecco alcuni esempi di questi operatori usati per selezionare parole dai nostri testi: parole che terminano con ***-ableness***; parole contenenti ***gnt***; parole che hanno un capitale iniziale; e parole composte interamente da cifre.

```
>>> sorted(w for w in set(text1) if w.endswith('ableness'))
['comfortableness', 'honourableness', 'immutableness', 'indispensableness',
...]
>>> sorted(term for term in set(text4) if 'gnt' in term)
['Sovereignty', 'sovereignties', 'sovereignty']
>>> sorted(item for item in set(text6) if item.istitle())
['A', 'Aaaaaaaaah', 'Aaaaaaaah', 'Aaaaaah', 'Aaaah', 'Aaaaugh', 'Aaagh', ...]
>>> sorted(item for item in set(sent7) if item.isdigit())
['29', '61']
```

NLTK: utilities

Esercizio:

Eseguire i seguenti esempi e provare a spiegare cosa fa ognuno di essi. Quindi, provare a creare alcune condizioni personali.

```
>>> sorted(w for w in set(text7) if '-' in w and 'index' in w)
>>> sorted(w for wd in set(text3) if w.istitle() and len(w) > 10)
>>> sorted(w for w in set(sent7) if not w.islower())
>>> sorted(t for t in set(text2) if 'cie' in t or 'cei' in t)
```

NLTK: Accesso ai *Corpora*

Come appena accennato, un corpus di testo è un grande insieme di testi.
Molti corpora sono progettati per contenere un attento equilibrio di materiale in uno o più generi. Abbiamo esaminato alcune piccole raccolte di testi, come gli interventi noti come gli discorsi inaugurali presidenziali degli Stati Uniti.

Questo corpus particolare contiene in realtà dozzine di singoli testi - uno per discorso - ma per praticità li abbiamo incollati e trattati come un singolo testo.
Abbiamo anche utilizzato vari testi predefiniti a cui abbiamo avuto accesso digitando da ***nltk.book import** **.

Tuttavia, dal momento che vogliamo essere in grado di lavorare con altri testi, esamineremo una varietà di corpora testuali. Vedremo come selezionare singoli testi e come lavorare con loro.

NLTK: Gutenberg Corpus

NLTK include una piccola selezione di testi dall'archivio di testo elettronico di Project Gutenberg, che contiene circa 25.000 libri elettronici gratuiti, ospitati su http://www.gutenberg.org/.

Iniziamo chiedendo a Python di caricare il pacchetto NLTK, poi chiediamo di vedere ***nltk.corpus.gutenberg.fileids()***, gli identificatori di file in questo corpus.

```
>>> import nltk
>>> nltk.corpus.gutenberg.fileids()
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt',
'blake-poems.txt', 'bryant-stories.txt', 'burgess-busterbrown.txt',
'carroll-alice.txt', 'chesterton-ball.txt', 'chesterton-brown.txt',
'chesterton-thursday.txt', 'edgeworth-parents.txt', 'melville-moby_dick.txt',
'milton-paradise.txt', 'shakespeare-caesar.txt', 'shakespeare-hamlet.txt',
'shakespeare-macbeth.txt', 'whitman-leaves.txt']
```

NLTK: Gutenberg Corpus

Scegliamo il primo di questi testi - Emma di Jane Austen - e diamo un nome breve, emma e vediamo quante parole contiene:

```
>>> emma = nltk.corpus.gutenberg.words('austen-emma.txt')
>>> len(emma)
192427
```

Possiamo ora trasformare il testo precedente utilizzando la class ***nltk.Text()***.

```
>>> emma = nltk.Text(nltk.corpus.gutenberg.words('austen-emma.txt'))
>>> emma.concordance("surprize")
er father , was sometimes taken by surprize at his being still able to pity `
hem do the other any good ." " You surprize me ! Emma must do Harriet good :
Knightley actually looked red with surprize and displeasure , as he stood up ,
r . Elton , and found to his great surprize , that Mr . Elton was actually on
```

NLTK: Gutenberg Corpus

Scriviamo un breve programma per visualizzare altre informazioni su ogni testo, eseguendo il ***for*** su tutti i valori di ***fileId*** corrispondenti agli identificatori di file ***gutenberg*** elencati in precedenza e quindi calcolando le statistiche per ogni testo. Arrotondiamo ciascun numero al numero intero più vicino, usando ***round()***.

```
>>> for fileid in gutenberg.fileids():
...     num_chars = len(gutenberg.raw(fileid))
...     num_words = len(gutenberg.words(fileid))
...     num_sents = len(gutenberg.sents(fileid))
...     num_vocab = len(set(w.lower() for w in gutenberg.words(fileid)))
...     print(round(num_chars/num_words), round(num_words/num_sents),
round(num_words/num_vocab), fileid)
5 25 26 austen-emma.txt
5 26 17 austen-persuasion.txt
5 28 22 austen-sense.txt
4 34 79 bible-kjv.txt
5 19 5 blake-poems.txt
4 19 14 bryant-stories.txt
4 18 12 burgess-busterbrown.txt
```

NLTK: Gutenberg Corpus

Questo programma visualizza tre statistiche per ciascun testo.

- ➔ lunghezza media delle parole,
- ➔ lunghezza media delle frasi e
- ➔ diversità lessicale.

La lunghezza media delle parole sembra essere una **proprietà** generale dell'inglese, poiché ha un valore ricorrente di 4. (In realtà, la lunghezza media delle parole è 3 e non 4, poiché la variabile ***num_chars*** conta i caratteri dello spazio.)

Per contro la lunghezza media delle frasi e la diversità lessicale sembrano essere caratteristiche di autori particolari.

NLTK: Gutenberg Corpus

La funzione ***raw()*** ci fornisce il contenuto del file senza alcuna elaborazione linguistica. Quindi, ad esempio, ***len(gutenberg.raw('blake-poems.txt'))*** ci dice quante lettere ci sono nel testo, compresi gli spazi tra le parole. La funzione **sents()** divide il testo nelle sue frasi, dove ogni frase è una lista di parole:

```
>>> macbeth_sentences = gutenberg.sents('shakespeare-macbeth.txt')
>>> macbeth_sentences
[['[', 'The', 'Tragedie', 'of', 'Macbeth', 'by', 'William', 'Shakespeare',
'1603', ']'], ['Actus', 'Primus', '.'], ...]
>>> macbeth_sentences[1116]
['Double', ',', 'double', ',', 'toile', 'and', 'trouble', ';',
'Fire', 'burne', ',', 'and', 'Cauldron', 'bubble']
>>> longest_len = max([len(s) for s in macbeth_sentences])
>>> [s for s in macbeth_sentences if len(s) == longest_len]
[['Doubtfull', 'it', 'stood', ',', 'As', 'two', 'spent', 'Swimmers', ',', 'that',
'doe', 'cling', 'together', ',', 'And', 'choake', 'their', 'Art', ':', 'The',
'mercilesse', 'Macdonwald', ...]]
```

NLTK: Web and Chat Text

Sebbene Project Gutenberg contenga migliaia di libri, rappresenta una letteratura consolidata.
È importante considerare anche un linguaggio meno formale.
La piccola raccolta di testi web di NLTK include i contenuti di un forum di discussione di ***Firefox***, conversazioni ascoltate a ***New York***, la sceneggiatura di ***Pirati dei Caraibi***, ***annunci personali*** e ***recensioni di vini***:

```
>>> from nltk.corpus import webtext
>>> for fileid in webtext.fileids():
...     print(fileid, webtext.raw(fileid)[:65], '...')
...
firefox.txt Cookie Manager: "Don't allow sites that set removed cookies to se...
grail.txt SCENE 1: [wind] [clop clop clop] KING ARTHUR: Whoa there!  [clop...
overheard.txt White guy: So, do you have any plans for this evening? Asian girl...
pirates.txt PIRATES OF THE CARRIBEAN: DEAD MAN'S CHEST, by Ted Elliott & Terr...
singles.txt 25 SEXY MALE, seeks attrac older single lady, for discreet encoun...
wine.txt Lovely delicate, fragrant Rhone wine. Polished leather and strawb...
```

NLTK: Web and Chat Text

Esiste anche un corpus di sessioni di chat di messaggistica istantanea, originariamente raccolte dalla ***Naval Postgraduate School*** per la ricerca sul rilevamento automatico dei predatori di Internet.
Il corpus contiene oltre 10.000 post, resi anonimi sostituendo nomi utente con nomi generici del modulo *"UserNNN"* e modificati manualmente per rimuovere qualsiasi altra informazione di identificazione. Il corpus è organizzato in 15 file, in cui ogni file contiene diverse centinaia di post raccolti in una determinata data, per una chat room specifica per età (adolescenti, 20, 30, 40, oltre a una chat room generica per adulti).
Il nome file contiene la data, la chat e il numero di post; ad esempio, 10-19-20s_706posts.xml contiene 706 post raccolti dalla chat room del 19/10/2006.

```
>>> from nltk.corpus import nps_chat
>>> chatroom = nps_chat.posts('10-19-20s_706posts.xml')
>>> chatroom[123]
['i', 'do', "n't", 'want', 'hot', 'pics', 'of', 'a', 'female', ',',
'I', 'can', 'look', 'in', 'a', 'mirror', '.']
```

NLTK: Brown Corpus

Il Brown Corpus era il primo corpus elettronico di milioni di parole inglesi, creato nel 1961 alla Brown University. Questo corpus contiene testo da 500 fonti e le fonti sono state categorizzate per genere, come notizie, editoriali e così via(per un elenco completo, vedi http://icame.uib.no/brown/bcm-los.html).

Il Brown Corpus è una comoda risorsa per studiare le differenze sistematiche tra i generi, una sorta di indagine linguistica nota come stilistica.
Confrontiamo i generi nel loro uso dei verbi modali. Il primo passo è produrre i conteggi per un particolare genere. Ricordarsi di importare nltk prima di fare quanto segue:

```
>>> from nltk.corpus import brown
>>> news_text = brown.words(categories='news')
>>> fdist = nltk.FreqDist(w.lower() for w in news_text)
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> for m in modals:
...     print(m + ':', fdist[m])
can: 94 could: 87 may: 93 might: 38 must: 53 will: 389
```

NLTK: Brown Corpus

Successivamente, dobbiamo ottenere i conteggi per ogni genere di interesse.
Useremo il supporto di NLTK per le distribuzioni di ***frequenza condizionale***.
Per il momento, ignoriamo i dettagli e concentriamoci solo sull'output.

```
>>> cfd = nltk.ConditionalFreqDist(
...         (genre, word)
...         for genre in brown.categories()
...         for word in brown.words(categories=genre))
>>> genres = ['news', 'religion', 'hobbies', 'science_fiction', 'romance', 'humor']
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> cfd.tabulate(conditions=genres, samples=modals)
                 can could  may might must will
        news   93   86   66   38   50  389
    religion   82   59   78   12   54   71
     hobbies  268   58  131   22   83  264
science_fiction   16   49    4   12    8   16
     romance   74  193   11   51   45    4
```

Il verbo modale più frequente nel genere delle ***news*** è ***will***, mentre la modale più frequente nel genere ***romance could***.

NLTK: Reuters Corpus

Il Corpus Reuters contiene 10.788 documenti di notizie per un totale di 1,3 milioni di parole. I documenti sono stati classificati in 90 argomenti e raggruppati in due gruppi, denominati "***training***" e "***test***"; quindi, il testo con fileid '***test/14826***' è un documento tratto dal set di test.
Questa suddivisione è per gli algoritmi di training e test che rilevano automaticamente l'argomento di un documento.

```
>>> from nltk.corpus import reuters
>>> reuters.fileids()
['test/14826', 'test/14828', 'test/14829', 'test/14832', ...]
>>> reuters.categories()
['acq', 'alum', 'barley', 'bop', 'carcass', 'castor-oil', 'cocoa',
'coconut', 'coconut-oil', 'coffee', 'copper', 'copra-cake', 'corn',
'cotton', 'cotton-oil', 'cpi', 'cpu', 'crude', 'dfl', 'dlr', ...]
```

NLTK: Inaugural Address Corpus

Il corpus è in realtà una raccolta di 55 testi, uno per ciascun discorso inauguarle presidenziale.
Una proprietà interessante di questa collezione è la sua dimensione temporale:

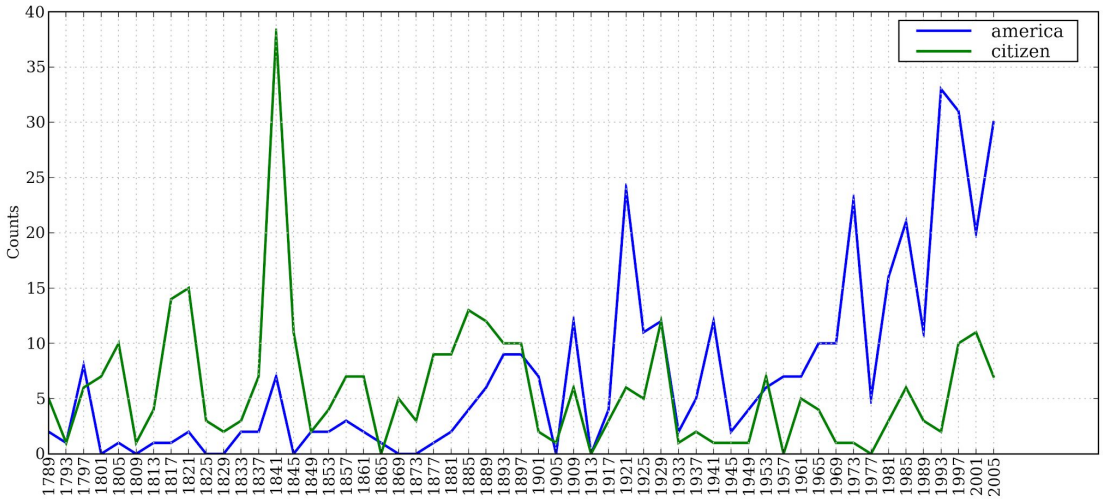
```
>>> from nltk.corpus import inaugural
>>> inaugural.fileids()
['1789-Washington.txt', '1793-Washington.txt', '1797-Adams.txt', ...]
>>> [fileid[:4] for fileid in inaugural.fileids()]
['1789', '1793', '1797', '1801', '1805', '1809', '1813', '1817', '1821', ...]
```

NLTK: Inaugural Address Corpus

Si noti che l'anno di ciascun testo appare nel suo nome file. Per ottenere l'anno dal nome del file, abbiamo estratto i primi quattro caratteri, usando fileid [: 4].

Scopriamo come le parole America e cittadino sono usate nel tempo. Il seguente codice converte le parole del corpus in minuscolo usando ***w.lower()***, quindi controlla se iniziano con uno dei "***target***" **america** o **citizen** usando ***startswith()***. Quindi conterà parole come American e Citizens.
Utilizziamo poi la distribuzione di frequenze condizionata creare il grafico.

```
>>> cfd = nltk.ConditionalFreqDist(
...         (target, fileid[:4])
...         for fileid in inaugural.fileids()
...         for w in inaugural.words(fileid)
...         for target in ['america', 'citizen']
...         if w.lower().startswith(target))
>>> cfd.plot()
```



NLTK: Corpora altri linguaggi

NLTK viene fornito con corpora per molte lingue, sebbene in alcuni casi sia necessario imparare come manipolare le ***codifiche dei caratteri*** in Python prima di utilizzare questi corpora.

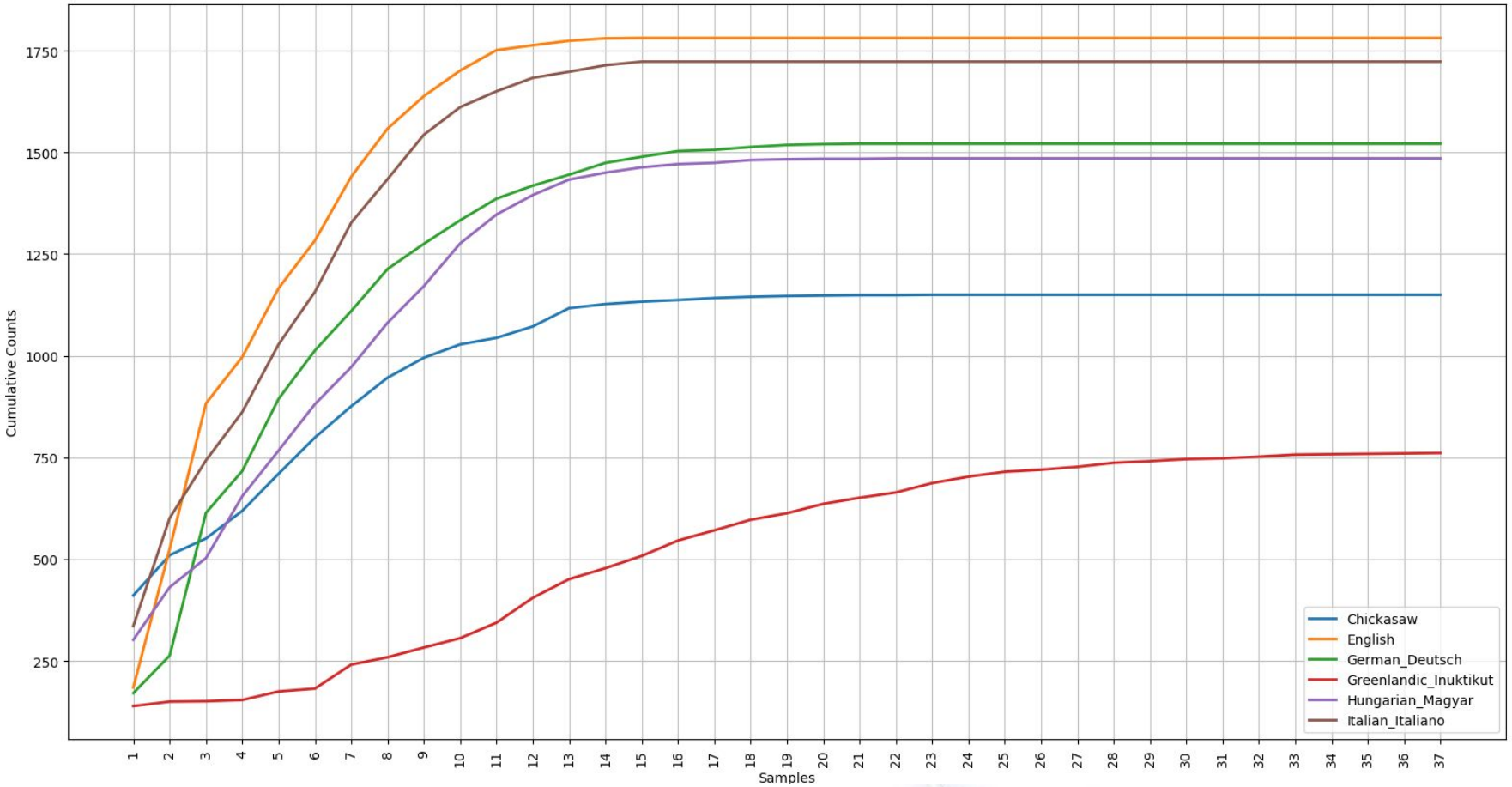
```
>>> nltk.corpus.cess_esp.words()
['El', 'grupo', 'estatal', 'Electricit\xe9_de_France', ...]
>>> nltk.corpus.floresta.words()
['Um', 'revivalismo', 'refrescante', 'O', '7_e_Meio', ...]
>>> nltk.corpus.indian.words('hindi.pos')
['पूर्ण', 'प्रतिबंध', 'हटाओ', ':', 'इराक', 'संयुक्त', ...]
>>> nltk.corpus.udhr.fileids()
['Abkhaz-Cyrillic+Abkh', 'Abkhaz-UTF8', 'Achehnese-Latin1', 'Achuar-Shiwiar-Latin1',
'Adja-UTF8', 'Afaan_Oromo_Oromiffa-Latin1', 'Afrikaans-Latin1', 'Aguaruna-Latin1',
'Akuapem_Twi-UTF8', 'Albanian_Shqip-Latin1', 'Amahuaca', 'Amahuaca-Latin1', ...]
>>> nltk.corpus.udhr.words('Javanese-Latin1')[11:]
['Saben', 'umat', 'manungsa', 'lair', 'kanthi', 'hak', ...]
```

NLTK: Corpora altri linguaggi

L'ultimo di questi corpora, udhr, contiene la Dichiarazione Universale dei Diritti Umani in oltre 300 lingue. I ***fileids*** per questo corpus includono informazioni sulla codifica dei caratteri utilizzata nel file, come ***UTF8*** o ***Latin1***.
Usiamo una distribuzione di frequenza condizionale per esaminare le differenze nelle lunghezze delle parole per una selezione di lingue incluse nel corpus udhr.

```
>>> from nltk.corpus import udhr
>>> languages = ['Chickasaw', 'English', 'German_Deutsch',
...    'Greenlandic_Inuktikut', 'Hungarian_Magyar', 'Italian_Italiano']
>>> cfd = nltk.ConditionalFreqDist(
...           (lang, len(word))
...           for lang in languages
...           for word in udhr.words(lang + '-Latin1'))
>>> cfd.plot(cumulative=True)
```

NLTK: Corpora altri linguaggi



NLTK: Conditional Frequency Distributions

Condition: News

the	𝍸 𝍸 𝍸 ||||
cute	
Monday	𝍸 ||||
could	|
will	𝍸 |||

Condition: Romance

the	𝍸 𝍸 |||
cute	|||
Monday	|
could	𝍸 𝍸 𝍸
will	||||

NLTK: Conditional Frequency Distributions

Abbiamo introdotto distribuzioni di frequenza e abbiamo visto che, dato un elenco di parole o altri elementi, ***FreqDist (mylist)*** calcola il numero di occorrenze di ciascun elemento nell'elenco. Generalizziamo questa idea.

Quando i testi di un corpus sono suddivisi in diverse categorie, per genere, argomento, autore, ecc., possiamo mantenere distribuzioni di frequenza separate per ogni categoria.

Questo ci permetterà di studiare le differenze sistematiche tra le categorie. Abbiamo già ottenuto questo risultato utilizzando la classe ***ConditionalFreqDist*** di NLTK.

Una distribuzione di frequenza condizionale è una raccolta di distribuzioni di frequenza, ciascuna per una ***"condizione"*** diversa.

NLTK: Conditional Frequency Distributions

Condizioni ed Eventi

Una distribuzione di frequenza conta eventi osservabili, come l'occorrenza di parole in un testo.

Una distribuzione di frequenza condizionale deve associare ogni evento a una condizione. Quindi, invece di elaborare una sequenza di parole, dobbiamo elaborare una sequenza di tuple:

```
>>> text = ['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
>>> pairs = [('news', 'The'), ('news', 'Fulton'), ('news', 'County'), ...]
```

NLTK: Conditional Frequency Distributions

Contare le parole per genere

Abbiamo visto una distribuzione di frequenza condizionale in cui la condizione era la sezione del Brown corpus, e per ogni condizione abbiamo contato le parole. Mentre ***FreqDist()*** prende un semplice elenco come input, ***ConditionalFreqDist()*** prende un elenco di coppie.

```
>>> from nltk.corpus import brown
>>> cfd = nltk.ConditionalFreqDist(
...         (genre, word)
...         for genre in brown.categories()
...         for word in brown.words(categories=genre))
```

NLTK: Conditional Frequency Distributions

Contare le parole per genere

Vediamo per semplicità solo due generi, notizie e romanticismo.

Per ogni genere [2], eseguiamo il ***for*** su ogni parola di quel genere [3], producendo coppie costituite dal genere e la parola [1]:

```
>>> genre_word = [(genre, word) #(1)
...              for genre in ['news', 'romance']  #(2)
...              for word in brown.words(categories=genre)] #(3)
>>> len(genre_word)
170576
```

Quindi, come possiamo vedere qui sotto, le coppie all'inizio della lista ***genre_word*** saranno della forma ***('news', word)*** [1], mentre quelle alla fine saranno della forma ***('romance', word)*** [2].

NLTK: Conditional Frequency Distributions

Contare le parole per genere

Ora possiamo usare questa lista di coppie per creare un ***ConditionalFreqDist*** e salvarlo in una variabile ***cfd***. Come al solito, possiamo digitare il nome della variabile per controllarlo e verificare che abbia due condizioni:

```
>>> genre_word[:4]
[('news', 'The'), ('news', 'Fulton'), ('news', 'County'), ('news', 'Grand')] #
[_start-genre]
>>> genre_word[-4:]
[('romance', 'afraid'), ('romance', 'not'), ('romance', """), ('romance', '.')] #
[_end-genre]
>>> cfd = nltk.ConditionalFreqDist(genre_word)
>>> cfd.conditions()
['news', 'romance'] # [_conditions-cfd]
```

NLTK: Conditional Frequency Distributions

Contare le parole per genere

Accediamo alle due condizioni e ci accertiamo che ciascuna sia solo una distribuzione di frequenza:

```
>>> print(cfd['news'])
<FreqDist with 14394 samples and 100554 outcomes>
>>> print(cfd['romance'])
<FreqDist with 8452 samples and 70022 outcomes>
>>> cfd['romance'].most_common(20)
[(',', 3899), ('.', 3736), ('the', 2758), ('and', 1776), ('to', 1502),
('a', 1335), ('of', 1186), ('``', 1045), ("''", 1044), ('was', 993),
('I', 951), ('in', 875), ('he', 702), ('had', 692), ('?', 690),
('her', 651), ('that', 583), ('it', 573), ('his', 559), ('she', 496)]
>>> cfd['romance']['could']
193
```

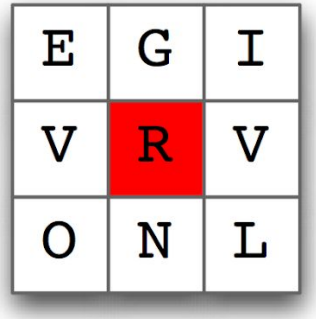
NLTK: Wordlist Corpora

NLTK include alcuni corpora che non sono altro che liste di parole.

```
>>> from nltk.corpus import stopwords
>>> stopwords.words('english')
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your', 'yours',
'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', 'her', 'hers',
'herself', 'it', 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves',
'what', 'which', 'who', 'whom', 'this', 'that', 'these', 'those', 'am', 'is', 'are',
'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does',
'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until',
'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into',
'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here',
'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more',
'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so',
'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', 'should', 'now']
```

NLTK: Wordlist Corpora

Una lista di parole è utile per risolvere i puzzle di parole. Il nostro programma scorre ogni parola e, per ognuno, controlla se soddisfa le condizioni.



How many words of four letters or more can you make from those shown here? Each letter may be used once per word. Each word must contain the center letter and there must be at least one nine-letter word. No plurals ending in "s"; no foreign words; no proper names. 21 words, good; 32 words, very good; 42 words, excellent.

NLTK: Wordlist Corpora

È facile controllare la lettera obbligatoria [2] e i vincoli di lunghezza [1] (e cercheremo solo parole con sei o più lettere qui). È più complicato verificare che le soluzioni candidate utilizzino solo combinazioni delle lettere fornite, soprattutto perché alcune delle lettere fornite appaiono due volte (qui, la lettera v). Il metodo di confronto ***FreqDist*** [3] ci consente di verificare che la frequenza di ogni lettera nella parola candidata sia inferiore o uguale alla frequenza della lettera corrispondente nel puzzle.

```
>>> puzzle_letters = nltk.FreqDist('egivrvonl')
>>> obligatory = 'r'
>>> wordlist = nltk.corpus.words.words()
>>> [w for w in wordlist if len(w) >= 4 #(1)
...                 and obligatory in w  #(2)
...                 and nltk.FreqDist(w) <= puzzle_letters] #(3)
['glover', 'gorlin', 'govern', 'grovel', 'ignore', 'involver', 'lienor',
'linger', 'longer', 'lovering', 'noiler', 'overling', 'region', 'renvoi',
'revolving', 'ringle', 'roving', 'violer', 'virole']
```

NLTK: Wordlist Corpora

Un altro corpus di liste di parole è il corpus di Nomi, contenente 8.000 nomi per categoria.

I nomi maschili e femminili sono memorizzati in file separati.

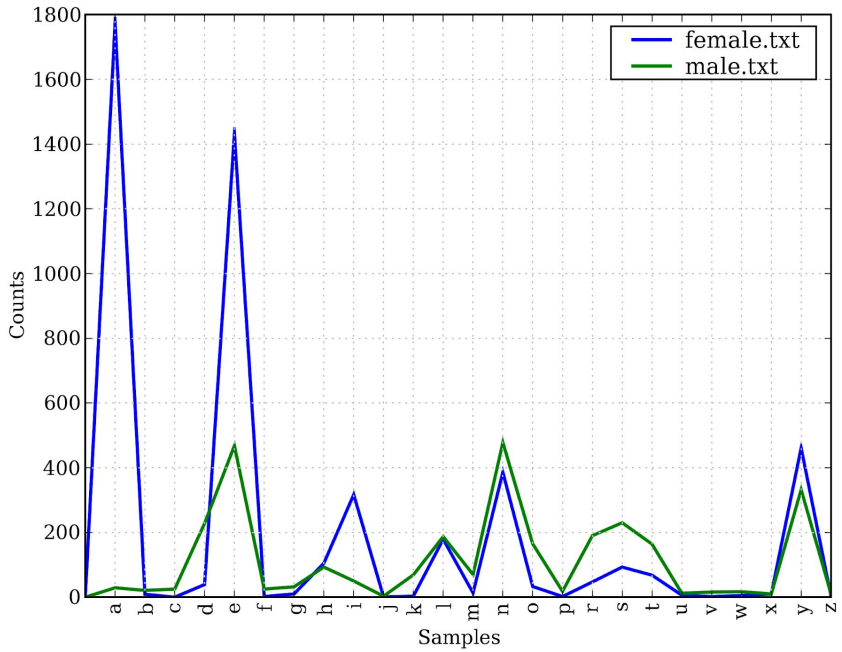
Scopriamo i nomi che appaiono in entrambi i file, cioè i nomi ambigui per genere:

```
>>> names = nltk.corpus.names
>>> names.fileids()
['female.txt', 'male.txt']
>>> male_names = set(names.words('male.txt'))
>>> female_names = set(names.words('female.txt'))
>>> sorted(list(male_names.intersection(female_names)))
['Abbey', 'Abbie', 'Abby', 'Addie', 'Adrian', 'Adrien', 'Ajay', 'Alex', 'Alexis',
'Alfie', 'Ali', 'Alix', 'Allie', 'Allyn', 'Andie', 'Andrea', 'Andy', 'Angel',
'Angie', 'Ariel', 'Ashley', 'Aubrey', 'Augustine', 'Austin', 'Averil', ...]
```

NLTK: Wordlist Corpora

È noto che i nomi che terminano con la lettera a sono quasi sempre femminili. Possiamo vedere questo e alcuni altri pattern nel grafico prodotto dal seguente codice. Ricorda che il nome [-1] è l'ultima lettera del nome.

```
>>> cfd = nltk.ConditionalFreqDist(
...         (fileid, name[-1])
...         for fileid in names.fileids()
...         for name in names.words(fileid))
>>> cfd.plot()
```



NLTK: WordNet

WordNet è un dizionario semantico della lingua inglese, simile a un thesaurus tradizionale ma con una struttura molto più ricca.

NLTK include il WordNet in inglese, con 155.287 parole e 117.659 gruppi di sinonimi.

- ➔ Benz is credited with the invention of the motorcar.
- ➔ Benz is credited with the invention of the automobile.

Poiché tutto il resto della frase è rimasto invariato, possiamo concludere che le parole ***motorcar*** e ***automobile*** hanno lo stesso significato, cioè sono sinonimi. Possiamo esplorare queste parole con l'aiuto di WordNet:

```
>>> from nltk.corpus import wordnet as wn
>>> wn.synsets('motorcar')
[Synset('car.n.01')]
```

NLTK: WordNet

Quindi ***motorcar*** ha un solo significato possibile ed è identificata come ***car.n.01***, il primo senso del senso dell'automobile. L'entità ***car.n.01*** è chiamata ***synset***, o "insieme sinonimo", una raccolta di parole sinonimi (o "lemmi"):

```
>>> wn.synset('car.n.01').lemma_names()
['car', 'auto', 'automobile', 'machine', 'motorcar']
```

Ogni parola di un synset può avere diversi significati, ad esempio l'auto può anche indicare una carrozza del treno, una gondola o un'auto dell'ascensore.

Tuttavia, siamo interessati al solo significato comune a tutte le parole del synset di cui sopra.

NLTK: WordNet

I synset hanno anche con una definizione e alcune frasi di esempio:

```
>>> wn.synset('car.n.01').definition()
'a motor vehicle with four wheels; usually propelled by an internal combustion engine'
>>> wn.synset('car.n.01').examples()
['he needs a car to get to work']
```

NLTK: WordNet

Sebbene le definizioni aiutino gli esseri umani a comprendere il significato previsto di un synset, le parole del synset sono spesso più utili per i nostri programmi.

Per eliminare l'ambiguità, identificheremo queste parole come ***car.n.01.automobile***, ***car.n.01.motorcar***, e così via. Questo abbinamento di un synset con una parola è chiamato ***lemma***. Possiamo ottenere tutti i lemmi per un dato synset [1], cercare un particolare lemma [2], ottenere il synset corrispondente a un lemma [3], e ottenere il "nome" di un lemma [4]:

```
>>> wn.synset('car.n.01').lemmas() #(1)
[Lemma('car.n.01.car'), Lemma('car.n.01.auto'), Lemma('car.n.01.automobile'),
Lemma('car.n.01.machine'), Lemma('car.n.01.motorcar')]
>>> wn.lemma('car.n.01.automobile') #(2)
Lemma('car.n.01.automobile')
>>> wn.lemma('car.n.01.automobile').synset() #(3)
Synset('car.n.01')
>>> wn.lemma('car.n.01.automobile').name() #(4)
'automobile
```

NLTK: WordNet

A differenza della parola ***automobile***, che non ha ambiguità e ha un synset, la parola ***car*** è ambigua, con cinque synset:

```
>>> wn.synsets('car')
[Synset('car.n.01'), Synset('car.n.02'), Synset('car.n.03'), Synset('car.n.04'),
Synset('cable_car.n.01')]
>>> for synset in wn.synsets('car'):
...     print(synset.lemma_names())
['car', 'auto', 'automobile', 'machine', 'motorcar']
['car', 'railcar', 'railway_car', 'railroad_car']
['car', 'gondola']
['car', 'elevator_car']
['cable_car', 'car']
```

NLTK: WordNet

Per comodità, possiamo accedere a tutti i lemmi che coinvolgono la parola auto come segue.

```
>>> wn.lemmas('car')
[Lemma('car.n.01.car'), Lemma('car.n.02.car'), Lemma('car.n.03.car'),
Lemma('car.n.04.car'), Lemma('cable_car.n.01.car')]
```

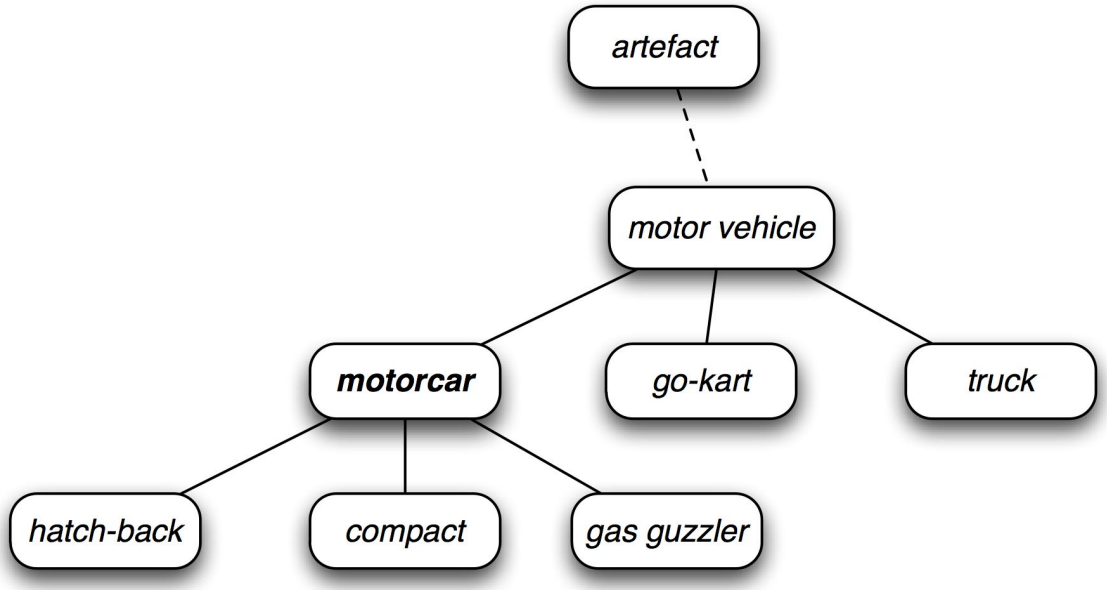
Esercizio:

Scrivere tutti i sensi della parola "***dish***" vengono in mente.

Ora, esplorare questa parola con l'aiuto di WordNet, usando le stesse operazioni che abbiamo usato sopra.

NLTK: WordNet

I synset di WordNet corrispondono a concetti astratti e non sempre hanno parole corrispondenti in inglese. Questi concetti sono collegati tra loro in una gerarchia. Alcuni concetti sono molto generali, come Entità, Stato, Evento - questi sono chiamati ***root synets***. Altri, come benzinaio e hatchback, sono molto più specifici.



NLTK: WordNet

WordNet semplifica la navigazione tra i concetti. Ad esempio, dato un concetto come l'automobile, possiamo guardare i concetti che sono più specifici; gli ***iponimi*** (immediati).

```
>>> motorcar = wn.synset('car.n.01')
>>> types_of_motorcar = motorcar.hyponyms()
>>> types_of_motorcar[0]
Synset('ambulance.n.01')
>>> sorted(lemma.name() for synset in types_of_motorcar for lemma in synset.lemmas())
['Model_T', 'S.U.V.', 'SUV', 'Stanley_Steamer', 'ambulance', 'beach_waggon',
'beach_wagon', 'bus', 'cab', 'compact', 'compact_car', 'convertible',
'coupe', 'cruiser', 'electric', 'electric_automobile', 'electric_car',
'estate_car', 'gas_guzzler', 'hack', 'hardtop', 'hatchback', 'heap',
'horseless_carriage', 'hot-rod', 'hot_rod', 'jalopy', 'jeep', 'landrover',
'limo', 'limousine', 'loaner', 'minicar', 'minivan', 'pace_car', 'patrol_car',
'phaeton', 'police_car', 'police_cruiser', 'prowl_car', 'race_car', 'racer',
'racing_car', 'roadster', 'runabout', 'saloon', 'secondhand_car', 'sedan',
'sport_car', 'sport_utility', 'sport_utility_vehicle', 'sports_car', 'squad_car',
'station_waggon', 'station_wagon', 'stock_car', 'subcompact', 'subcompact_car',
'taxi', 'taxicab', 'tourer', 'touring_car', 'two-seater', 'used-car', 'waggon',
'wagon']
```

NLTK: WordNet

Possiamo anche navigare su per la gerarchia visitando ***iperonimi***. Alcune parole hanno più percorsi, perché possono essere classificati in più di un modo. Esistono due percorsi tra ***car.n.01*** e ***entity.n.01*** poiché ***wheeled_vehicle.n.01*** può essere classificato sia come veicolo sia come contenitore.

```
>>> motorcar.hypernyms()
[Synset('motor_vehicle.n.01')]
>>> paths = motorcar.hypernym_paths()
>>> len(paths)
2
>>> [synset.name() for synset in paths[0]]
['entity.n.01', 'physical_entity.n.01', 'object.n.01', 'whole.n.02', 'artifact.n.01',
'instrumentality.n.03', 'container.n.01', 'wheeled_vehicle.n.01',
'self-propelled_vehicle.n.01', 'motor_vehicle.n.01', 'car.n.01']
>>> [synset.name() for synset in paths[1]]
['entity.n.01', 'physical_entity.n.01', 'object.n.01', 'whole.n.02', 'artifact.n.01',
'instrumentality.n.03', 'conveyance.n.03', 'vehicle.n.01', 'wheeled_vehicle.n.01',
'self-propelled_vehicle.n.01', 'motor_vehicle.n.01', 'car.n.01']
```

NLTK: WordNet

Possiamo ottenere gli iperinomi più generali (o ipernomi radice) di un synset come segue:

```
>>> motorcar.root_hypernyms()
[Synset('entity.n.01')]
```

Iperonimi e iponimi sono chiamati relazioni lessicali perché mettono in relazione un synset con un altro.

Queste due relazioni navigano su e giù per la gerarchia "***is-a***". Un altro modo importante per navigare nella rete ***WordNet*** è considerando gli elementi componenti (meronimi) o gli elementi contenuti (olonimi). Ad esempio, le parti di un albero sono il suo tronco, la sua corona e così via; il ***part_meronyms***(). La sostanza di cui è composto un albero comprende durame e alburno; the ***substance_meronyms*** (). Una collezione di alberi forma una foresta; the ***member_holonyms*** ():

NLTK: WordNet

```
>>> wn.synset('tree.n.01').part_meronyms()
[Synset('burl.n.02'), Synset('crown.n.07'), Synset('limb.n.02'),
Synset('stump.n.01'), Synset('trunk.n.01')]
>>> wn.synset('tree.n.01').substance_meronyms()
[Synset('heartwood.n.01'), Synset('sapwood.n.01')]
>>> wn.synset('tree.n.01').member_holonyms()
[Synset('forest.n.01')]
```

Per vedere quanto possono essere intricate le cose,prendiamo in considerazione la parola menta, che ha molti sensi strettamente correlati.
Possiamo vedere che ***mint.n.04*** è parte di ***mint.n.02*** e la sostanza da cui è prodotto ***mint.n.05***.

NLTK: WordNet

```
>>> for synset in wn.synsets('mint', wn.NOUN):
...     print(synset.name() + ':', synset.definition())
batch.n.02: (often followed by `of') a large number or amount or extent
mint.n.02: any north temperate plant of the genus Mentha with aromatic
leaves and
          small mauve flowers
mint.n.03: any member of the mint family of plants
mint.n.04: the leaves of a mint plant used fresh or candied
mint.n.05: a candy that is flavored with a mint oil
mint.n.06: a plant where money is coined by authority of the government
>>> wn.synset('mint.n.04').part_holonyms()
[Synset('mint.n.02')]
>>> wn.synset('mint.n.04').substance_holonyms()
[Synset('mint.n.05')]
```

NLTK: WordNet and Semantic Similarity

Abbiamo visto che i synset sono collegati da una complessa rete di relazioni lessicali. Dato un particolare synset, possiamo attraversare la rete WordNet per trovare synset con significati correlati.

Sapere quali parole sono correlate semanticamente è utile per l'indicizzazione di una raccolta di testi, in modo che la ricerca di un termine generale come veicolo corrisponda a documenti contenenti termini specifici come la limousine.

Ogni synset ha uno o più percorsi hypernym che lo collegano a un hypernym di root come ***entity.n.01***. Due synset collegati alla stessa radice possono avere diversi iperonimi in comune. Se due synset condividono un iperonimo molto specifico - uno che è in basso nella gerarchia iperonimi - devono essere strettamente correlati.

NLTK: WordNet and Semantic Similarity

```
>>> right = wn.synset('right_whale.n.01')
>>> orca = wn.synset('orca.n.01')
>>> minke = wn.synset('minke_whale.n.01')
>>> tortoise = wn.synset('tortoise.n.01')
>>> novel = wn.synset('novel.n.01')
>>> right.lowest_common_hypernyms(minke)
[Synset('baleen_whale.n.01')]
>>> right.lowest_common_hypernyms(orca)
[Synset('whale.n.02')]
>>> right.lowest_common_hypernyms(tortoise)
[Synset('vertebrate.n.01')]
>>> right.lowest_common_hypernyms(novel)
[Synset('entity.n.01')]
```

NLTK: WordNet and Semantic Similarity

Ovviamente sappiamo che la balena è molto specifica (e che i fanoni di balena lo sono ancora di più), mentre i vertebrati sono più generici e l'entità è completamente generale.
Possiamo quantificare questo concetto di generalità osservando la profondità di ogni synset:

```
>>> wn.synset('baleen_whale.n.01').min_depth()
14
>>> wn.synset('whale.n.02').min_depth()
13
>>> wn.synset('vertebrate.n.01').min_depth()
8
>>> wn.synset('entity.n.01').min_depth()
0
```

NLTK: WordNet and Semantic Similarity

Le misure di similarità sono state definite sulla collezione di synset di WordNet che incorporano l'intuizione di cui sopra. Ad esempio, ***path_similarity*** assegna un punteggio nell'intervallo 0-1 in base al percorso più breve che collega i concetti nella gerarchia hypernym (-1 viene restituito in quei casi in cui non è possibile trovare un percorso). Considera i seguenti punteggi di similarità, che riguardano la balena franca, la balena minke, l'orca, la tartaruga e il romanzo. Anche se i numeri non significano molto, diminuiscono man mano che ci spostiamo dallo spazio semantico delle creature marine a oggetti inanimati.

```
>>> right.path_similarity(minke)
0.25
>>> right.path_similarity(orca)
0.16666666666666666
>>> right.path_similarity(tortoise)
0.07692307692307693
>>> right.path_similarity(novel)
0.043478260869565216
```

NLTK: WordNet Visualization

Utilizzando la libreria è possibile manipolare strutture dati composte da nodi e archi, cioè delle reti complesse. Possiamo per esempio visualizzare la rete di relazioni semantiche di un dato synset presente in Wordnet.

```
import networkx as nx
import matplotlib
from nltk.corpus import wordnet as wn

def traverse(graph, start, node):
    graph.depth[node.name] = node.shortest_path_distance(start)
    for child in node.hyponyms():
        graph.add_edge(node.name, child.name)
        traverse(graph, start, child)
```

trasverse è una funzione ***ricorsiva*** che dato un nodo di partenza, esamina tutti i suoi iperonimi.

NLTK: WordNet Visualization

hyponym_graph inizializza e restituisce un grafo creato con la precedente funzione ***trasverse.***

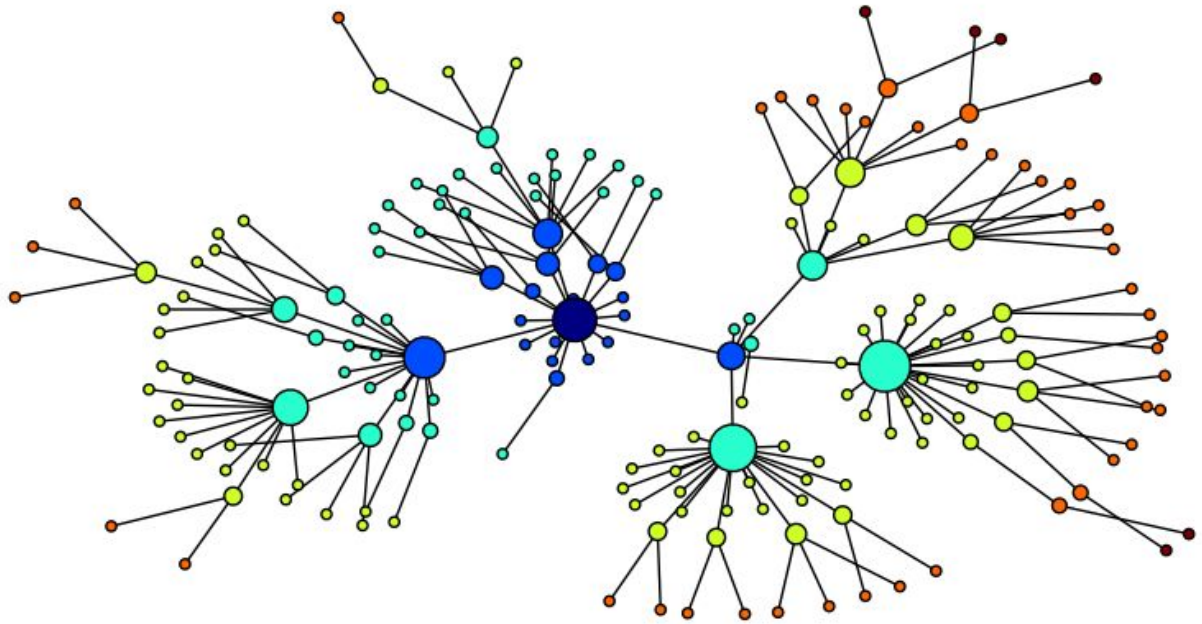
```
def hyponym_graph(start):
    G = nx.Graph()
    G.depth = {}
    traverse(G, start, start)
    return G

def graph_draw(graph):
    nx.draw_networkx(graph,
        node_size = [16 * graph.degree(n) for n in graph],
        node_color = [graph.depth[n] for n in graph],
        with_labels = False)
    matplotlib.pyplot.show()
```

NLTK: WordNet Visualization

Per visualizzare la rete è sufficiente selezionare un ***synset*** da ***Wordnet*** e richiamare le funzioni definite precedentemente.

```
>>> dog = wn.synset('dog.n.01')
>>> graph = hyponym_graph(dog)
>>> graph_draw(graph)
```



NLTK: Comprensione automatica del linguaggio naturale

Abbiamo esplorato il linguaggio dal basso verso l'alto, con l'aiuto di testi e il linguaggio di programmazione Python.
Tuttavia, siamo anche interessati a sfruttare la nostra conoscenza del linguaggio e della computazione costruendo tecnologie linguistiche utili. Ora cogliamo l'occasione per fare un passo indietro rispetto al nocciolo del codice al fine di dipingere un quadro più ampio dell'elaborazione del linguaggio naturale.

A un livello puramente pratico, tutti abbiamo bisogno di aiuto per navigare nell'universo di informazioni rinchiuso nel testo sul Web.
I motori di ricerca sono stati cruciali per la crescita e la popolarità del Web, ma hanno alcuni difetti.

Ci vogliono abilità, conoscenza e un po 'di fortuna per estrarre le risposte a domande come: Quali siti turistici posso visitare tra Philadelphia e Pittsburgh con un budget limitato? Cosa dicono gli esperti delle fotocamere reflex digitali? Quali previsioni sul mercato dell'acciaio sono state fatte da commentatori credibili nella scorsa settimana?

NLTK: Comprensione automatica del linguaggio naturale

Realizzare un computer per rispondere automaticamente comporta una serie di attività di elaborazione del linguaggio, tra cui l'estrazione di informazioni, l'inferenza e il riepilogo, e dovrebbe essere eseguito su una scala e con un livello di robustezza che è ancora al di là delle nostre attuali capacità.

A un livello più filosofico, una sfida di vecchia data nell'intelligenza artificiale è stata quella di costruire macchine intelligenti, e una parte importante del comportamento intelligente è la comprensione del linguaggio.

Per molti anni questo obiettivo è stato visto come troppo difficile.
Tuttavia, man mano che le tecnologie PNL diventano più mature e i metodi più efficaci per analizzare il testo senza restrizioni diventano più diffusi, la prospettiva della comprensione del linguaggio naturale è riemersa come un obiettivo plausibile.

NLTK: Word Sense Disambiguation

Con **word sense disambiguation** vogliamo capire quale senso di una parola è inteso in un determinato contesto. Consideriamo le parole ***serve*** and ***dish***:

- ➔ serve: help with food or drink; hold an office; put ball into play
- ➔ dish: plate; course of a meal; communications device

In una frase contenente la frase: "he served the dish", è possibile rilevare che sia il servizio che il piatto vengono usati con i loro significati alimentari.
È improbabile che l'argomento della discussione si spostasse dallo sport alle stoviglie nello spazio di tre parole.
Questo ti costringerebbe ad inventare immagini bizzarre, come un professionista del tennis che tira fuori le sue frustrazioni su un servizio da tè in porcellana disposto accanto alla corte.
In altre parole, disambiguiamo automaticamente le parole usando il contesto, sfruttando il semplice fatto che le parole vicine hanno significati strettamente correlati.

NLTK: Word Sense Disambiguation

Come altro esempio di questo “effetto contestuale”, considera la parola “ by”, che ha diversi significati, per esempio il libro di Chesterton (agente - Chesterton era l'autore del libro); la tazza vicino alla stufa (locativo - la stufa è dove è la tazza); e inviare entro venerdì (temporale - il venerdì è l'ora della presentazione). Osservare in che il significato della parola in corsivo ci aiuta a interpretare il significato di.

- ➔ The lost children were found by the ***searchers*** (agentive)
- ➔ The lost children were found by the ***mountain*** (locative)
- ➔ The lost children were found by the ***afternoon*** (temporal)

NLTK: Spoken Dialog Systems

Nella storia dell'intelligenza artificiale, la principale misura dell'intelligenza è stata quella linguistica, in particolare il **test di Turing**: un sistema di dialogo, rispondente al testo di un utente, funziona in modo così naturale che non possiamo distinguerlo da una risposta generata dall'uomo.
Al contrario, i sistemi di dialogo commerciale di oggi sono molto limitati, ma svolgono comunque funzioni utili in domini a definizione ristretta, come vediamo qui:

S: Come posso aiutarti?
U: Quando verrà proiettato “Salvate il soldato Ryan”?
S: in quale cinema?
U: Il cinema Paramount.
S: Salvare il soldato Ryan non è proiettato al al cinema Paramount, ma al cinema Madison alle 3:00, 5:30, 8:00 e 10:30.

NLTK: Spoken Dialog Systems

Non è possibile chiedere a questo sistema di fornire istruzioni di guida o dettagli sui ristoranti nelle vicinanze a meno che le informazioni richieste non siano già state memorizzate e siano state incorporate coppie di domande e risposte adeguate nel sistema di elaborazione della lingua.

Questo sistema sembra comprendere gli obiettivi dell'utente: l'utente chiede quando viene mostrato un film e il sistema determina correttamente da questo che l'utente vuole vedere il film.

Questa deduzione sembra così ovvia che probabilmente non ti sei accorto che è stato ***creato***, tuttavia un sistema di linguaggio naturale deve essere dotato di questa capacità per interagire in modo naturale.

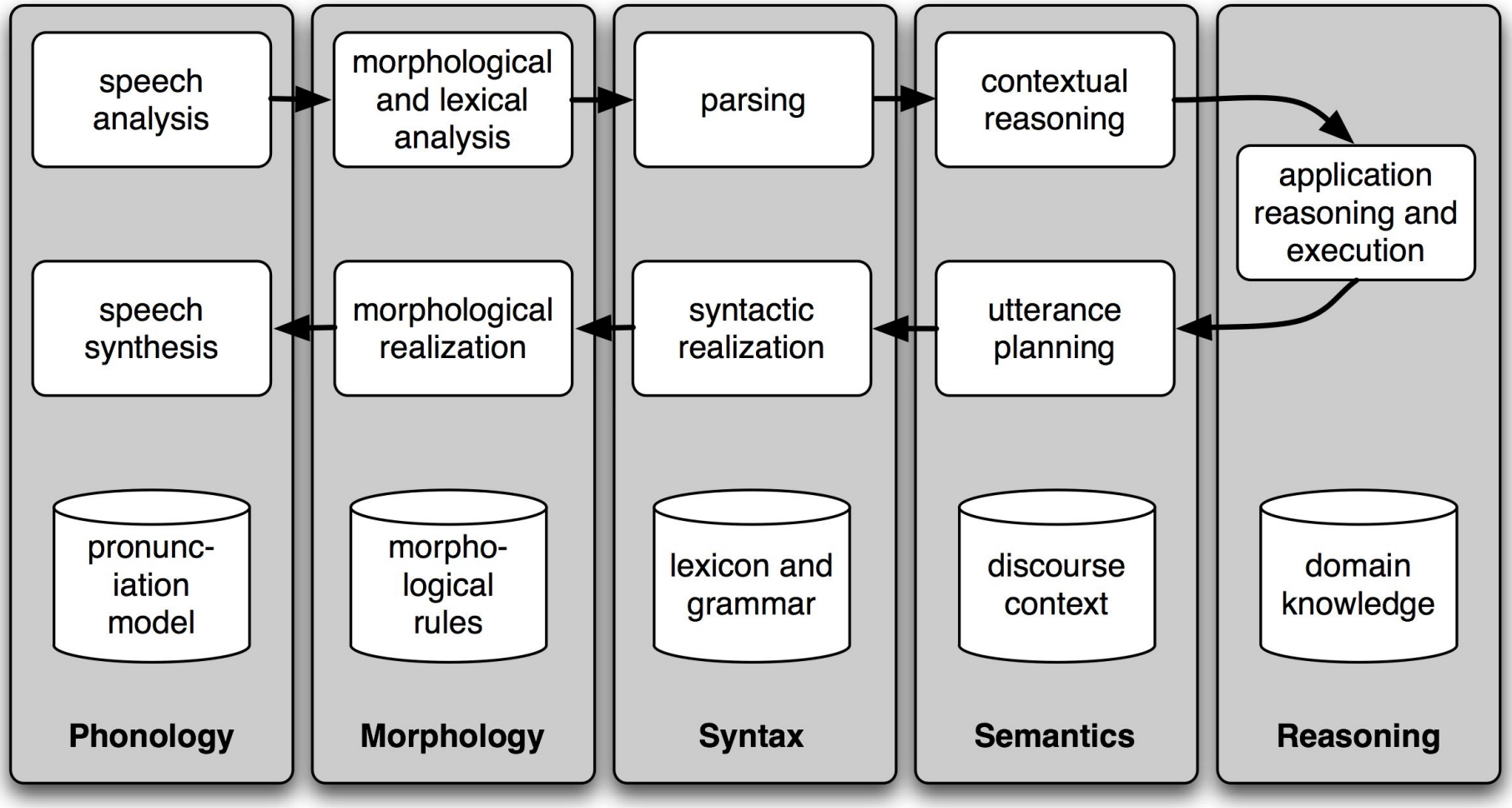
NLTK: Spoken Dialog Systems

Senza di esso, quando richiesto “***Sai*** quando è in proiezione il salvataggio di Ryan privato ?” un sistema potrebbe rispondere in modo non corretto con un freddo “Sì”.

Tuttavia, gli sviluppatori di sistemi di dialogo commerciale utilizzano assunzioni contestuali e logica aziendale per garantire che i diversi modi in cui un utente possa esprimere richieste o fornire informazioni siano gestiti in modo ragionevole per la particolare applicazione.

Quindi, se digiti When is ..., o voglio sapere quando ..., o Puoi dirmi quando ..., le regole semplici porteranno sempre alle date di proiezione. Questo è sufficiente per il sistema per fornire un servizio utile.

NLTK: Spoken Dialog Systems



NLTK: Spoken Dialog Systems

I sistemi di dialogo ci danno l'opportunità di menzionare la pipeline comunemente utilizzata per la PNL. Il precedente schema mostra l'architettura di un semplice sistema di dialogo.

Lungo la parte superiore del diagramma, spostandosi da sinistra a destra, c'è una "pipe" di alcuni componenti di comprensione del linguaggio. Queste mappe derivano dall'inserimento vocale tramite l'analisi sintattica di un qualche tipo di rappresentazione del significato.

Spostandosi da destra a sinistra, si trova la pipeline inversa di componenti per convertire i concetti in parlato. Questi componenti costituiscono gli aspetti dinamici del sistema.
Nella parte inferiore del diagramma vi sono alcuni corpi rappresentativi di informazioni statiche: i repository di dati relativi alla lingua che i componenti di elaborazione attingono per svolgere il proprio lavoro.

NLTK: riferimenti

- http://www.nltk.org/book/ch01.html
- http://www.nltk.org/book/ch02.html
- Natural Language Processing with Python(ISBN-13: 978-0596516499)